ANIMAL-FARM: An Extensible Framework for Algorithm Visualization

Vom Fachbereich Elektrotechnik und Informatik
der Universität Siegen
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

von

Diplom-Informatiker Guido Rößling

geboren in Darmstadt

1. Gutachter: Prof. Dr. Bernd Freisleben
2. Gutachter: Prof. Dr. Mira Mezini

Tag der mündlichen Prüfung: 11. April 2002

Acknowledgments

The material presented in this thesis is based on my research during my time as a research assistant in the Parallel Systems Research Group at the Department of Electrical Engineering and Computer Science at the University of Siegen, Germany.
I would like to thank all collaboration partners who made this research enjoyable and productive. Apart from many discussion partners at conferences who offered insights or sparked ideas, I want to thank some persons individually.
First of all, I want to thank my academic mentor Bernd Freisleben who gave me the freedom needed for developing this thesis. He also supported the incorporation of algorithm visualizations in the Introduction to Computer Science course. Apart from giving hints on academic research techniques and assisting me in writing papers, he has also graciously invested much financial support into the stressful year of 2000, when I was presenting my research status at six conferences.
I also wish to thank my second advisor Mira Mezini for her advice in software engineering aspects. Her insights on developing adaptable software systems gave me a starting point for my exploration of dynamically configurable and extensible software.
Several members at the department have helped developing ANIMAL-FARM and the ANIMAL system. Markus Schüler helped to design and implement the first ANIMAL prototype as his Master's Thesis. Jens Brodowski and André Flöper adapted parts of the graphical editing facilities to changed Swing interfaces. Matthew Smith and Patrick Ahlbrecht also contributed to later stages of the development process by implementing add-ons. André Berten and Frank Katritzke proof-read drafts of this thesis.
Several researchers around the globe contributed ideas to the thesis. First of all, I want to thank Tom Naps for many discussions and the productive cooperation in adapting algorithm visualization systems to pedagogical purposes. Stuart Hansen provided several animations generated from student assignments, as well as hints on improving ANIMAL. Nils Faltin offered ideas on adapting algorithm visualizations to improve the success of teaching the underlying content.
I thank my parents for their continuous support, without which I would not have been able to come this far. Finally, grateful thanks go to my wife Annika and our son Fabian for their support during the development of this thesis and especially the finishing phase. Without them and their inspiration, I would have found it much harder to take a step back during the writing process and allow phases of rest that helped firming up the content of the thesis.

Zusammenfassung

Die Visualisierung und Animation von Algorithmen ist in den letzten Jahren sowohl bei Lehrkräften als auch bei Lernenden auf weltweit zunehmendes Interesse gestoßen. Dieses Interesse wird belegt durch die wachsende Anzahl an wissenschaftlichen Publikationen, verfügbaren Animationssystemen und die Anzahl frei erhältlicher Animationen im World Wide Web. In ihrem Artikel von 1998 geben Price, Baecker und Small die Zahl verfügbarer Animationssysteme als "über 150" an [156]. Seit dem Zeitpunkt der Publikation ist diese Zahl nochmal deutlich gestiegen.
Diese Arbeit stellt ein Framework für Algorithmenvisualisierung (AV) vor, das zahlreiche besondere Eigenschaften bietet, die es von aktuellen Animationssystemen abhebt. Hierzu zählt insbesondere die sehr weitreichende Dynamik des Frameworks hinsichtlich seiner Zusammenstellung. Für diesen Zweck wird die Verwendung dynamischer Lookup-Datenstrukturen konsequent auf die Administration sowohl von Komponenten als auch deren Attributen ausgedehnt. Als Ergebnis entsteht ein Framework, das die Einfügung einzelner Komponenten zur Laufzeit durch dynamisches Laden ebenso unterstützt wie die Entfernung von Komponenten und die Einführung zusätzlicher "Attribute".
Ein besonderes Konzept zur dynamischen Vermittlung nutzbarer Funktionalitäten durch eine Abarbeitung von Anfragen und Bereitstellung von Auswahlmöglichkeiten mittels einer sogenannten *Handler*-Klasse findet im Framework weitreichenden Einsatz. Dieses Konzept kann auch in vielen anderen Softwarebereichen erfolgreich eingesetzt werden. Zusätzlich unterstützt das Framework die Verwendung übersetzbarer GUI-Elemente auf Basis der Java Swing-Bibliothek, wobei gleichzeitig die Erstellung der Komponenten vereinfacht wird. Da diese dynamischen Eigenschaften des Frameworks für viele Anwendungsgebiete interessant sind, werden sie im Rahmen der Arbeit zunächst abstrakt vorgestellt, bevor sie in den konkreten Anwendungszweck der Algorithmenvisualisierung eingebettet werden.
Zu den besonderen Leistungen des Frameworks im Bereich der Algorithmenvisualisierung zählt insbesondere die Unterstützung für flexible Ablaufkontrolle inklusive vollwertigem Rückwärtsabspielen der Animationsinhalte. Zur Demonstration der Nutzbarkeit des Frameworks stellt die Arbeit eine Umsetzung in einen konkreten Prototypen vor, der weitere Funktionsmerkmale beinhaltet. Hierzu zählen etwa die Unterstützung mehrerer Generierungs- und Editierarten sowie eine eingebaute Skriptsprache mit Unterstützung für modulare Animationskomponenten. Die direkte Übersetzung von Animationsinhalten mit automatischer Anpassung der Positionen davon betroffener Objekte wird im Rahmen des Prototyps erstmals im Bereich der AV vorgestellt.
Die Arbeit ist wie folgt strukturiert. Kapitel 2 gibt einen kurzen Überblick über die Geschichte der Algorithmenanimation. Basierend auf Price *et al.* [156] werden vier Rollen im Umgang mit AV-Systemen vorgestellt: *Nutzer* verwenden das System zum Betrachten und Interagieren mit der Animation; *Visualisierer* erstellen die Animationen; *Entwickler* erstellen oder erweitern das Animationssystem; *Programmierer* implementieren den zu animierenden Algorithmus, eventuell ohne dabei von einer geplanten Animation zu wissen. Einzelne Personen können dabei mehrere Rollen innehaben. So wird beispielsweise der *Visualisierer* oftmals die Rolle des *Nutzers* annehmen, um sich von der Qualität der erstellten Animation zu überzeugen.
Zusätzlich wird ein einfaches Klassifikationsschema für Animationssysteme vorgestellt, das die Systeme einteilt in *themengebundene* Systeme sowie anhand der Animationserstellungsart nach *manueller Eingabe, API-Programmierung, Skriptsprachenbefehlen* oder *logischen Beschreibungen*. Die Verteilung der einzelnen Rollen sowie typische Stärken und Schwächen der Ansätze werden dabei ebenfalls beleuchtet.

Basierend auf einer umfangreichen Betrachtung verfügbarer Applets und AV-Systemen sowie Literaturrecherchen werden Anforderungen an AV-Systeme formuliert. Die Anforderungen sind unterteilt in allgemeine sowie von den einzelnen Nutzungsrollen ausgehende und werden unter Berücksichtigung der relevanten Publikationen in Kapitel 3 präsentiert. Aktuelle AV-Systeme realisieren nur eine stark eingeschränkte Untermenge dieser Anforderungen. Andererseits macht es die stetig steigende Erwartungshaltung hinsichtlich Funktionalität in allen Softwarebereichen unwahrscheinlich, dass es ein "ideales" AV-System geben kann, das allen Wünschen gerecht werden kann. Daher sollte ein AV-System so flexibel gestaltet werden, dass Ergänzungen und Erweiterungen der Funktionalität so leicht wie möglich realisiert werden können.
Aufgrund dieser Überlegung präsentiert Kapitel 4 den Entwurf des ANIMAL-FARM Frameworks für dynamisch erweiterbare und anpassbare AV-Systeme. Das Framework adressiert sowohl softwaretechnische als auch AV-bezogene Anforderungen. Aus Sicht der Softwaretechnik ist insbesondere die abstrakte Beschreibung des Frameworks interessant, die auf dynamischen Laden und der intensiven Verwendung dynamischer *Lookup*-Strukturen beruht und damit eine dynamische Zusammenstellung und Erweiterbarkeit erlaubt. Hierzu zählt auch die strikte Entkoppelung zweier zentraler Komponenten durch das *Handler*-Konzept, das eine sehr konsequente Ausdehnung des Prinzips der Trennung von Aufgabengebieten (engl. *separation of concerns*) erlaubt und als Entwurfsmuster (engl. *design pattern*) für den Entwurf dynamisch anpassbarer Kommunikation dienen kann. Ein allgemeines Softwarepaket zur Übersetzung von grafischen Elementen rundet den softwaretechnischen Leistungsumfang des Frameworks ab.
Aus AV-Sicht ist einer der entscheidenden Beiträge des Frameworks die effiziente Unterstützung des Zurückspulens und sogar des flüssigen Rückwärtsabspielens von Animationen. Auf dem *First International Program Visualization Workshop* im Jahr 2000 stellte ein Mitglied des Programmkomitees die Bedeutung des Zurückspulens als "one of the most important 'open questions' in AV" heraus [2]. Das hier vorgestellte Framework beinhaltet diese Funktionalität, wurde aber weniger als ein halbes Jahr nach Publikation des Artikels fertiggestellt, was die Ausdruckskraft der ihm zugrundeliegenden Konzepte unterstreicht.
Kapitel 5 stellt die erste Umsetzung des Frameworks vor: das ANIMAL System. ANIMAL ist dynamisch erweiter- und anpassbar und komplett unabhängig von einem Verwendungskontext. Die meisten denkbaren grafischen Inhalte im zweidimensionalen Raum können basierend auf den unterstützten Primitiven *Punkt, Linienzug / Polygon, Text* und *Kreisbogen* sowie den Animationseffekten für das *zeigen / verbergen, verschieben, rotieren* sowie *umfärben* von Primitiven dargestellt werden. In einigen Fällen müssen Primitive zur Repräsentation eines neuen Objekttyps kombiniert werden.
Die Erweiterbarkeit von ANIMAL unter Nutzung der Strukturen des ANIMAL-FARM Frameworks wird in Kapitel 6 diskutiert, indem unter anderem Listenelemente als neue Primitive vorgestellt werden. Ein Listenelement besteht aus einem Text, zwei Polygonen sowie einer Menge von Linienzügen mit einer Pfeilspitze am Ende als Modellierung der Zeiger. Andere Beispielerweiterungen im Kapitel umfassen einen *Zoom* Effekt, weitere Untertypen des allgemeinen Effekts zum Verschieben von Objekten, zusätzliche Import- und Export-Filter sowie die Hinzufügung weiterer Sprachenunterstützung zur grafischen Benutzerschnittstelle.
Die Evaluation von ANIMAL in Kapitel 7 zeigt, dass ANIMAL alle gestellten Anforderungen bis auf sieben unterstützt. Sieben weitere Anforderungen werden nur teilweise angeboten, während fünf zusätzliche Anforderungen zwar erfüllt werden, von ANIMAL als kontextunabhängiges System aber nicht garantiert werden können. Insgesamt erfüllt ANIMAL mehr Anforderungen als irgendein anderes aktuelles AV-System.

Die wissenschaftlichen Hauptbeiträge dieser Arbeit liegen in verschiedenen Bereichen. Die in Kapitel 3 zusammengetragenen Anforderungen resultieren aus einer sorgfältigen Literaturrecherche sowie eigenen Überlegungen. Sie sind umfangreicher als bisherige einschlägige Publikationen. Gleichzeitig werden die vier verschiedenen Nutzungsrollen in Betracht gezogen sowie allgemeine Anforderungen gestellt.

Das Framework dehnt die Verwendung von *Hashtabellen* zur Speicherung von Werten konsequent auf die Anforderungen der dynamischen Erweiterbarkeit und Adaptivität aus. Anstelle einfacher Werte wird Hashing zur Verwaltung der zu einem konkreten Zeitpunkt zur Verfügung stehenden Komponenten genutzt, was eine leichte Ergänzung oder Entfernung von Komponenten begünstigt. Der Zustand der einzelnen Komponenten wird ebenfalls in einer dynamischen Struktur verwaltet anstelle einer festen Codierung in Form von Attributen. Hierdurch können Entwickler leicht weitere Zustandsbeschreibungen einfügen, ohne den Quelltext des Frameworks oder des Gesamtsystems zu ändern. Im Rahmen des Frameworks finden zusätzlich mehrere Entwurfsmuster Verwendung, indem etwa die zentralen Hashtabellen in einem *Singleton* [61, p. 217ff] verwaltet werden.

Die bereits erwähnten *Handler*-Klassen dienen zu einer vollständigen Entkoppelung von primären Komponenten. Für diese Vorgehensweise zur Vermittlung verfügbarer Funktionalitäten zwischen den entkoppelten Komponenten finden sich zahlreiche Anwendungsgebiete. Ein Hauptvorteil des Ansatzes ist dabei die stark vereinfachte Entwicklung von Erweiterungen der entkoppelten Komponenten durch eine klare Trennung der Aufgabengebiete (engl. *separation of concerns*). *Handler* agieren in diesem Sinn als ein spezielles Entwurfsmuster für die bessere Unterstützung von erweiterbaren Systemen.

Die Strategien für die Erstellung dynamisch erweiterbarer Systeme werden im Rahmen der Arbeit erstmals auf den Entwurf eines Frameworks für AV-Systeme übertragen. Hier kann die verwendete Modellierung ihre volle Flexibilität durch die von ihr unterstützten Operationen herausstellen. Die strikte Trennung der Aufgabengebiete mittels des *Handler*-Konzepts ermöglicht dabei eine einfache und effiziente Unterstützung des Rückwärtslaufs in Animationen. Gleichzeitig ermöglicht die Grundstruktur des Frameworks die Ausdehnung des Rückwärtslaufs – sowohl in Form eines schrittbasierten Zurückspulens als auch in dynamischem Rückwärtsabspielen – auf dynamisch eingebettete Erweiterungen von Animationseffekten und grafischen Objekte, ohne dass der Autor dieser Komponenten hierzu eine zusätzliche Implementierung leisten muss. Verglichen mit den meisten bisherigen AV-Systemen bietet das vorgestellte Framework "echtes" Rückwärtsspielen anstelle der Nutzung einer eingeschränkten Historie. Zusätzlich verfolgt das Framework eine deutliche Trennung von Inhalten (engl. *separation of concerns*), die in ihrer Ausgestaltung eine sehr große Flexibilität erlaubt.

Die im Rahmen des Frameworks vorgestellten Komponenten für übersetzbare GUI-Komponenten werden zusätzlich auch für die Unterstützung von übersetzbaren Animationen genutzt. Hierzu ist eine relative Platzierung der Primitiven erforderlich, um den unterschiedlichen Abmessungen von Texten in verschiedenen Sprachen Rechnung zu tragen.

Die in ANIMAL eingebettete und im Rahmen dieser Arbeit vorgestellte Skriptsprache ANIMALSCRIPT bietet zusätzlich Unterstützung für modulare Animationskomponenten mit der Möglichkeit zum Datenaustausch. ANIMALSCRIPT-Animationen können manuell erstellt oder automatisch generiert werden, wobei die Eingabedaten des Nutzers oder Visualisierers entsprechend berücksichtigt werden. In Kombination mit dem JHAVÉ-System [131, 181] unterstützt ANIMALSCRIPT ebenfalls interaktive Vorhersagen als didaktisch motiviertes Modell zur Verbesserung des Verständnisses.

Teile dieser Arbeit wurden bereits als Konferenzbeiträge oder Journalartikel veröffentlicht. Die

verschiedenen Techniken zur Generierung von Animationsinhalten für Lehrzwecke wurden zuerst bei der SITE Konferenz 2000 präsentiert [173]. Ein Beitrag bei der SIGCSE Konferenz im Jahr 2000 [174] erläuterte die Motivation für die Verwendung von AV-Systemen in einer Vorlesung zur Einführung in die Informatik. Der Beitrag stellte den damals aktuellen Stand des ANIMAL-Systems vor. Zwei Beispielanimation zu *Quicksort* und der *Verifikation* von Algorithmen wurden detailiert beschrieben, zusammen mit den im Einsatz von AV gewonnenen Erfahrungen.

Auf der Basis zweier Konferenzposter [175, 183] und der sich daraus ergebenden Diskussion mit Konferenzteilnehmern wurde auf der ITiCSE 2000 [184] der Entwurf des ANIMAL-Systems vorgestellt. Zu diesem Zeitpunkt bot ANIMAL nur eine grafische Schnittstelle zur Erzeugung von Animationen und eingeschränkte Unterstützung von Zoomen und Export. Ein Beitrag beim *First International Program Visualization Workshop* im Anschluß an ITiCSE 2000 [177] und einem Beitrag zur SIGCSE 2001 [179] stellte die Skriptsprache ANIMALSCRIPT vor. Eine kurze Beschreibung der dynamischen Erweiterbarkeit von ANIMALSCRIPT befindet sich ebenfalls in dem letztgenannten Beitrag.

Die vier Nutzerrollen gemäß Price *et al.* [156] wurden zuerst in einer gleichzeitigen Veröffentlichung in den Zeitschriften *Informatik / Informatique* [178] und *Novática* [176] der Informatik-Dachorganisationen der Schweiz und Spaniens verwendet. Diese Artikel beschreiben auch weiterführende Merkmale von ANIMAL, wie etwa die Internationalisierung und die Strukturierung von Animationen.

Drei weitere Publikationen befinden sich zur Zeit im Druck. Ein Artikel in der Aprilausgabe 2002 des *Journal of Visual Languages and Computing* [180] umfasst die Unterstützung der vier Nutzerrollen sowie die grundlegenden Merkmale im Rahmen von ANIMAL. Ein Beitrag bei der ITiCSE 2002 [181] präsentiert mehrere pädagogische Anforderungen an AV-Systeme und erläutert, wie diese in der Kombination von JHAVÉ [131] mit ANIMAL umgesetzt werden. Wesentliche Entscheidungshilfen für oder gegen konkrete Animationssysteme liefert der Beitrag bei der *SEC III* Konferenz [170].

Weitere Publikationen befinden sich zur Zeit in Begutachtung für den *Second International Program Visualization Workshop*. In [171] wird das ANIMAL-FARM Framework von Kapitel 4 vorgestellt. Ausgewählte aktuelle Forschungsvorhaben im AV-Bereich, die in [182] vorgestellt werden, sollen AV-System interaktiver machen und in die Richtung intelligenter Tutorensysteme weiterführen. Die Entwicklung eines umfangreichen AV-Repositories wird in [40] vorgestellt.

Contents

1 Introduction **1**

2 Related Work **5**
2.1 Introduction 5
2.2 A Short History of Algorithm Animation 6
2.3 Animation User Roles 7
2.4 Animation Generation Approaches 8
2.4.1 Topic-Specific Animation 8
2.4.2 Manual Generation in a GUI 10
2.4.3 API-based Generation 12
2.4.4 Scripting-based Generation 13
2.4.5 Declarative Visualization Generation 14
2.4.6 Generation By Code Interpretation 15
2.5 Evaluation of Representative Tools 17
2.5.1 Algorithm Animation Applets on the WWW 18
2.5.2 Algorithm Animation Systems 29
2.6 Summary 39

3 Requirements Analysis **43**
3.1 Introduction 43
3.2 General Requirements 44
3.2.1 System Requirements 44
3.2.2 Development State and Performance 46
3.2.3 Applicability 47
3.3 User Requirements 48
3.3.1 Content Presentation 48
3.3.2 Animation Display Controls 50
3.3.3 User Interaction 50
3.3.4 Educational Support 52
3.3.5 User Interface 54
3.3.6 File Exchange 55
3.3.7 Algorithm Understanding 56
3.3.8 Miscellaneous 57
3.4 Visualizer Requirements 58
3.4.1 Applicability 58

3.4.2 Animation Display . . . 60
3.4.3 Animation Generation . . . 61
3.5 Developer Requirements . . . 63
3.6 Programmer Requirements . . . 64
3.7 Summary . . . 64

4 The ANIMAL-FARM Algorithm Visualization Framework 67
4.1 Introduction . . . 67
4.2 Concepts for Extensible Framework Design . . . 68
4.2.1 Using Properties to Model Object State . . . 68
4.2.2 Flexible Import and Export . . . 75
4.2.3 Internationalization and GUI Creation Support . . . 78
4.2.4 Component Assembly and Administration . . . 81
4.2.5 Decoupling Associated Objects Using Handlers . . . 83
4.2.6 Summary of Framework Extensibility and Adaptivity Approaches . . . 86
4.3 A Brief Framework Overview . . . 88
4.4 Graphical Objects . . . 90
4.5 Animation Effects . . . 95
4.6 Graphical Object Handlers . . . 98
4.7 Animation Representation . . . 102
4.8 Animation Player Front-end . . . 106
4.9 Animation Import and Export . . . 112
4.10 Summary . . . 112

5 The ANIMAL Animation System 117
5.1 Introduction . . . 117
5.2 ANIMAL Architecture . . . 117
5.2.1 Graphical Primitives . . . 118
5.2.2 Point Primitives . . . 121
5.2.3 Polyline and Polygon Primitives . . . 123
5.2.4 Text Primitives . . . 124
5.2.5 Arc Primitives . . . 125
5.3 Animation Effects . . . 126
5.3.1 Show Effect . . . 129
5.3.2 Timed Show Effect . . . 130
5.3.3 Color Change Effect . . . 130
5.3.4 Move Effect . . . 132
5.3.5 Rotate Effect . . . 133
5.4 Transformation Handlers . . . 133
5.5 ANIMAL's Animation Display GUI . . . 135
5.6 ANIMAL's Editing GUI . . . 138
5.7 Import and Export . . . 141
5.7.1 ANIMALSCRIPT . . . 142
5.8 Summary . . . 144

6 Extending ANIMAL Using ANIMAL-FARM **147**
6.1 Introduction . . . 147
6.2 Extending Graphical Primitives . . . 148
6.2.1 Example Extension: *Image* Support . . . 148
6.2.2 Example Extension: *List Element* Support . . . 149
6.3 Extending Animation Effects . . . 151
6.4 Extending Handler Capabilities . . . 153
6.4.1 Implementing a New Handler . . . 153
6.4.2 Example Extension: *Move* Subtype Support . . . 154
6.5 Adding Language Support . . . 155
6.6 Extending Import and Export Facilities . . . 156
6.6.1 Example Extension: Adding JSamba Import Facilities . . . 156
6.6.2 Example Extension: Image and Video Export . . . 157
6.7 Interactivity Support . . . 157
6.8 Summary . . . 158

7 Evaluation **159**
7.1 Introduction . . . 159
7.2 Evaluation of ANIMAL's Support for the Requirements . . . 160
7.2.1 System Requirements Evaluation . . . 160
7.2.2 Extensibility Requirements . . . 160
7.2.3 Development State and Performance . . . 162
7.2.4 Applicability . . . 163
7.2.5 Animation Generation . . . 163
7.2.6 Content Presentation . . . 165
7.2.7 User Interface . . . 165
7.2.8 Content Display and Controls . . . 166
7.2.9 User Interaction . . . 167
7.2.10 Educational Support . . . 167
7.2.11 Algorithm Understanding Support . . . 168
7.2.12 File Exchange . . . 169
7.2.13 Programmer Requirements . . . 169
7.3 Summary . . . 169

8 Conclusions **171**

A References **175**
A.1 Extension Listings . . . 181

Chapter 1

Introduction

One central challenge in computer science lies in understanding the dynamics of algorithms and data structures. This is of particular importance in two areas: computer science education and programming, especially concerning debugging. Understanding how and why an algorithm such as Quicksort works is challenging for students and novice programmers. Finding a presentation that allows learners to see how the intricate details of an implementation cooperate and interdepend is usually difficult. Pinpointing the precise location of an implementation bug is difficult even for advanced programmers. Both tasks are especially difficult if a static medium such as paper or an irreversible presentation technique such as writing on a blackboard is used.

Software for instructing novices how to program and debug their code is increasingly used in computer science education. Algorithm Visualization (AV) systems are a special application of software visualization, focusing on visualizing the dynamic behavior of software. The most common application areas of AV lie in education and debugging. The interest in AV from computer science education has steadily increased over the last years, as evidenced by the number of publications on the topic. AV software visualizes the behavior of algorithms and data structures and thus mostly frees the user from having to manually trace the underlying implementation code.

A large number of AV systems are already available, mainly from educational institutions. They differ in a large number of ways, including the degree of interactivity, display strategy and the type of supported input. The interactivity ranges from slide show-like displays without any interactivity to full-fledged debuggers that let the user specify attribute values and invoke methods. Some displays may run automatically, possibly even without enabling the user to pause or slow down the display. Other displays are triggered solely by user events such as pressing mouse buttons. A large variety of different display control elements exist between the two extremes. Finally, the input data used by AV systems ranges from actual source code in a specific programming language to interactive graphical editing using direct manipulation.

The large variety of available systems presents a formidable challenge for users interested in employing AV. The strengths, weaknesses and restrictions of a given system are usually not obvious at first glance. One drawback common to most current AV systems is the lack of support for easy customization of the system. Furthermore, it is highly unlikely that a given fixed system can meet all demands of future applications. However, most systems cannot easily be extended with additional features.

This thesis covers three main aspects for addressing these concerns. First, an extensive set of requirements for an "ideal" AV system is defined. These requirements also address different types of interaction with AV systems including displaying and animation generation demands.

Second, we introduce a framework design for extensible and configurable systems. The framework consequently uses dynamic loading for component acquisition and hashing for administrating the components. Each hashed component acts as a *Prototype* [61, p. 127ff] and can be cloned for acquiring a new instance. The administration of components using hash tables allows easy addition and removal of individual components at run-time. By modeling the object state by properties instead of fixed attributes, the framework also allows the introduction of new object properties at run-time. The *handler* concept introduced in this thesis allows a very strict separation of concerns between two parties, with the negotiating handler fully decoupling the components. The framework also provides a package for on-the-fly translation of arbitrary Swing-based GUI components. The AV components of the framework support efficient animation rewinding and even reverse playing, which were considered "one of the most important 'open questions' in AV" in one of last year's more prominent AV publications [2].

Finally, we present a reference implementation prototype of the framework and analyze how it measures against the set of requirements. The benefits of the framework design regarding extensibility and adaptivity are emphasized by example extensions. As the prototype is extensible, requirements which are currently unsupported may be addressed by later extensions.

This thesis is organized as follows. Chapter 2 introduces key terms used throughout this research, followed by an extensive examination of related AV systems. In chapter 3, we define and motivate a large set of AV system requirements. Where applicable, we also provide references to other publications that outline the rationale for a given requirement. Chapter 4 presents the design for dynamically extensible and configurable frameworks. The chapter introduces the consequent usage of dynamic data structures for supporting dynamic extensibility, as well as the *handler* concept useful for offering selective views of a given object's functionality. The general-purpose framework design is then applied to AV with an emphasis on supporting the requirements discussed in chapter 3. In chapter 5, we present the implementation decisions taken in developing the prototypical ANIMAL AV system. The system is based on the framework introduced in chapter 4 and the set of requirements presented in chapter 3. Chapter 6 explores example extensions of the ANIMAL system. The features offered by ANIMAL are compared in chapter 7 with the requirements presented in chapter 3. Chapter 8 concludes the thesis and outlines areas of further research.

Parts of this thesis have been previously published in conference papers or journal articles. The different techniques for generating animation content for educational purposes were first presented at the AACE SITE conference in 2000 [173]. A SIGCSE paper in 2000 [174] motivated the use of adopting AV systems within an introductory computer science course. The primitives and effects supported by ANIMAL were presented including example animation screen shots. Two example animations, *Quicksort* and *Introduction to Verification*, were discussed in detail. The paper also summarized several of the lessons we learned during the course. Based on two conference posters [175, 183] and the discussion with conference participants, a publication at the ITiCSE conference in 2000 [184] outlined the design of the ANIMAL AV system. It also presented the first rough set of requirements for AV systems. At this time, the system offered only manual generation within a GUI and very limited zooming and export facilities. A paper presented at the *First International Program Visualization Workshop* in 2000 (published in 2001) [177] and a SIGCSE paper [179] introduced the scripting language ANIMALSCRIPT, illustrating the main features and the general structure. A short description of how new features can be added dynamically was included in the paper. The four usage roles described by Price *et al.* [156] were adopted by a joint publication in the journals *Informatik / Informatique* [178] and *Novática* [176] of the Swiss and Spanish Computer Societies, respectively. They also focused on additional functionality of the ANIMAL system, such

as internationalization and the labeling of animation steps.
Three new publications are currently accepted for publication and in print. A journal article in the *Journal of Visual Languages and Computing* special issue on Software Visualization [180] presents the features that ANIMAL offers for the four roles defined by Price *et al.* in [156] and outlines the basic features of ANIMAL. A paper at the ITiCSE 2002 conference [181] illustrates some of the pedagogical requirements for AV systems and how they are met by the combination of the JHAVÉ environment [131] and ANIMAL. Key decisions in adopting AV systems for teaching are presented in a paper for the *SEC III* conference 2002 [170].
Additionally, three publications are currently under review for publication in the *Second International Program Visualization Workshop*. The ANIMAL-FARM framework is presented in [171]. A second paper presents current research issues in AV, which shall take AV systems further in the direction of interactive intelligent tutoring [182]. Finally, the development of a comprehensive AV repository is presented in [40].

Chapter 2

Related Work

2.1 Introduction

Algorithm visualization is a subtopic of software visualization. It focuses on the visualization of the higher level abstractions which describe software [156]. Algorithm animation is a variant of AV that uses dynamic effects and transitions, rather than discrete scene images. Thus, it covers the dynamic display of actual implementation code, pseudo code or other abstract views. In the following, we will often refer to the field of AV as algorithm animation, or animation for short.
The interest in algorithm animation research and application to education has grown over the last years, as indicated by the growing number of publications. A good recent overview of the field is given in the book by Stasko *et al.* [198]. The basic form of algorithm animation is shown in Figure 2.1. The animation results from an "appropriate" transformation of the underlying algorithm. Various different types of transformation are examined in detail in later parts of this chapter.

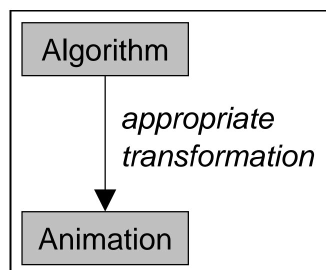

Figure 2.1: Base Form of Algorithm Animation Using an "Appropriate" Transformation

The term "animation" is understood differently by diverse researchers. Within the context of this work, animation always refers to a dynamic display of algorithmic behavior. The precise approach used for presenting the dynamics can choose from a set of options, including animated images or video streams. However, the focus is always taken to be on *algorithms*. Thus, special effects as used in movies like *Toy Story* are *not* counted as animations. This chapter presents some of the history of algorithm animation, the different roles taken by users of animation tools, and an overview of the different types of animation.
We give a brief outline of the history of algorithm animation and then discuss the different roles that a person using an animation system can assume. After exploring the different approaches

for generating algorithm animations, a large set of animation systems is briefly presented. We first examine several Java applets ordered by their degree of interactivity. Full-fledged animation systems are then examined ordered by the adopted generation type. The key findings of the chapter are summarized in the last section.

2.2 A Short History of Algorithm Animation

One main focus of algorithm animation is educational in nature: making algorithms easier to understand and grasp. For an overview of how the field has evolved, we take a step back to previous developments, many of which come from software engineering research. This research has led to a number of developments for addressing the problems of software complexity and comprehensibility. Examples include top-down design and stepwise refinement, as well as new concepts such as object-oriented approaches to software design and development. For example, *design patterns* [61] help to reduce the complexity of software by providing a reusable class structure for certain common structures. Other regions of research and progress include the organization and management of development teams, integrated software development environments and computer-aided software engineering (CASE).

Despite all these advances, the current appearance of programs usually does not contribute to making a program easily understandable. Baecker and Marcus [12] give an overview of various techniques for publishing C program code that help addressing this to a certain degree. Well-formatted code is easier to read and follow than unindented code, and the addition of various layout techniques can improve legibility even further. Comments following certain guidelines can be embedded into the code and extracted by a tool to generate documentation. This approach is employed for example in Java, but has been used at least since 1984, when the *WEB* system by Knuth embedded documentation into sources of the TEX typesetting system [106].

Probably the first visual aid for understanding computer programs are flowcharts which were first demonstrated in 1947 by von Neumann and Goldstein [13]. Several later developments allowed automatic generation from Fortran code, or embedding into source code. Nassi-Shneiderman diagrams, introduced in 1973, present an alternative approach that counters the unstructured nature of standard flowcharts.

All approaches described so far represent static visual displays of algorithms. The first dynamic displays go back to films by Knowlton [104, 105] that demonstrated a low-level list processing language developed at Bell labs. The movies are attributed as being the first to use animation techniques to portray program behavior and the first to address the visualization of dynamically changing data structures [13].

Several more short films by other educators followed, including material by John Hopgood for hashing and syntax analysis and an animated PQ-tree data structure algorithm. Probably the most influential film for algorithm animation is *Sorting Out Sorting* [14] by Baecker. Shown at many universities all over the world, the movie introduces nine different internal sorting methods, including an efficiency analysis. The movie is described in more detail in [11].

The 1980s saw the wider availability of affordable personal workstations with bit-mapped displays and graphical user interfaces. This advance allowed researchers to go beyond the prototypes and highly specific animations of the previous decade and develop full-fledged algorithm animation systems. The most important and well-known system of the time was the *Brown Algorithm Simulator and Animator (BALSA)* by Brown [30], followed by *BALSA-II* in 1988. Due to the tight

integration of *BALSA* with the teaching materials used, hundreds of undergraduates used the tool in their course of study. Several later tools were inspired by aspects of *BALSA*.
Since then, the number of animation systems has risen steadily. Price *et al.* [156] claim that by 1998, more than 150 software visualization prototype systems and animations had been built. The growing popularity of the Internet and especially the World-Wide Web has helped in spreading these tools and make users aware of the large number of offers available. *The Complete Collection of Algorithm Animations* [33] lists all resources found by its creator in 1998. The reference book by Stasko *et al.* [198] provides a good overview of the state of the art of the late 1990s.
Today, various tools stray into hitherto unexplored areas of algorithm animation, including program auralization [28] and three-dimensional displays [29, 163]. Given the development speed in the young area of algorithm animation, it will be interesting to see what the state of art will be in ten years. A good overview of the history of AV and general issues can be found in [198].

2.3 Animation User Roles

Price *et al.* [156] define four different roles in algorithm animation: *programmer, software visualization software developer* (or simply *developer*), *visualizer* and *user*. *Programmer* refers to the implementer of the underlying algorithm to be animated, for example Quicksort. *Developer* refers to the implementer of the animation system, while the *visualizer* specifies the animation. Finally, the *user* views the resulting animation. Depending on the system, this may also include various degrees of interaction.
Figure 2.2 shows the different roles and how they tie in with the algorithm, animation system and the animation display. Note that in this context, the programmer may be unaware of animation plans by the visualizer. Therefore, many animation systems may not be able to provide services for the programmer role.

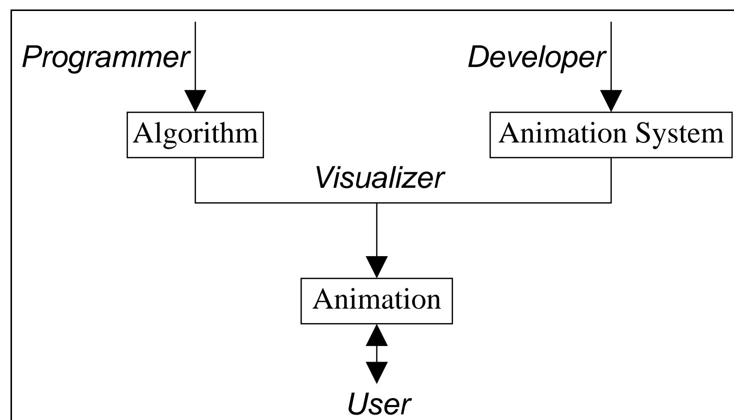

Figure 2.2: Schematic View of User Roles

Each role has specific expectations of algorithm animation systems. Developers may be interested in how easy it is to update or extend the system. They may also want to be able to adapt the system to their preferences. Visualizers require flexible animation generation; different ways of generating animations may be required to address personal preferences or experience levels. Finally, users

want to have a tool that runs smoothly, is easy to use and offers features such as video player-like controls.
Many animation systems are geared for one or at most two different roles. In general, the later in the development or display cycle of an animation a role appears, the more likely it is to be supported by a given animation system. Thus, users located at the bottom rung of the generation process will usually find several features that allow for attractive animation display. Visualizers are also likely to find features that help them in developing new animations, with the exception of some closed-shop systems that can only display selected animations with fixed data sets.
Developers are often not specially supported, due to the fact that most systems are simply shipped "as is" and are not meant to be extended or adapted at the client side. Finally, support for the programmer roles requires special care, as the programmer may not be aware of future animation plans. Thus, support for this role is usually limited to programs that work on interpreting the underlying algorithm's source code or extract debug data for interpretation.

2.4 Animation Generation Approaches

In this section, we discuss the approaches for generating animations. This classification is especially important for the visualizer role responsible for generating animations, but may also have consequences on the other roles. Visualizers should choose an animation system that fits their personal skills and preferences. Such systems may be easier or "more fun" to use, which may be reflected in both the quantity and quality of the generated animations. The type of animation generation does not directly impact the features offered to the developer, programmer or user role. Therefore, the following sections focus on the support for the visualizer role, enabling them to determine the base type of animation generation most appropriate to their situation.
There are several different approaches for how animations can be generated by a visualizer. One approach is using a graphical user interface, as included in most presentation tools such as Microsoft PowerPoint™ and StarOffice Impress™, or special generation tools such as Macromedia Flash. On the other end of the scale, some systems can automatically generate animations by interpreting source code or debug data. Between these extremes lie approaches that employ method invocations using a special visualization application programmer's interface (API), use a collection of ASCII-based commands (often also referred to as "scripting"), generate animations from declarations such as logical predicates, or explicitly encode animations for special topic areas. The following sections examine the different approaches for animation generation and characterize the distribution of the four roles in each approach.

2.4.1 Topic-Specific Animation

Topic-specific animation systems offer special support for a limited number of applications. This support is usually hard-coded, so that the system is restricted to the covered topics.
Figure 2.3 shows the typical role distribution for topic-specific animation systems. The *programmer* implements the original algorithm. The *developer* implements the complete tool including the graphical user front-end for displaying the animation. The developer may use the implementation provided by the programmer, or re-implement the algorithm. This alternative is indicated by a dotted line connecting the original implementation and the animation system. The developer has to adapt the algorithm so that it provides an animation while it is being executed. This is often

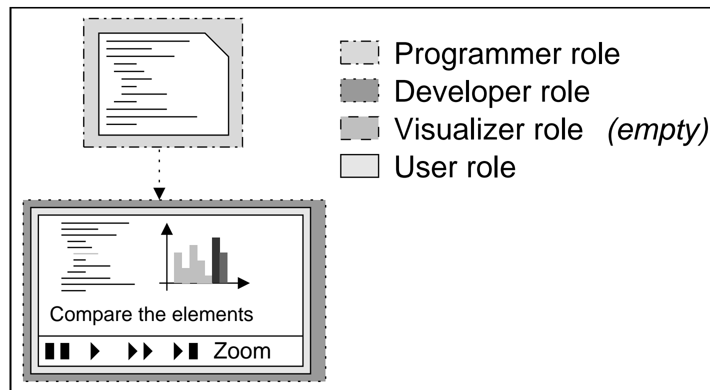

Figure 2.3: Role Distribution in Topic-Specific Systems

accomplished by adding special method invocations for generating the visualization. The original algorithm may also be split into separate units which are invoked in turn, each providing a visualization of special sub-aspects of the algorithm.
The animation of the underlying algorithm is generated automatically. Thus, the *visualizer* role has no direct part in the presentation. It may be possible to configure the appearance or behavior of the display, for example by setting the animation speed or zoom factor. However, these features can typically also be adjusted by the end user. Configurations by the visualizer may therefore be readjusted by the user, weakening the role of the visualizer.
The *user* interacts with the graphical front-end to see the animation. The amount of adjustable features varies with each tool, but will usually include a “play” or “step” operation for displaying the next step. Figure 2.3 also indicates some other possible features including pause, run, execute until end, or zooming.
We select the image compression packages RLE, Quadtree and JPEG presented in [101] for illustrating the typical strengths and weaknesses of topic-specific animations. Each of these packages animates a special approach for image compression using a fixed graphical front-end for visualizing the embedded implementation code. Therefore, they cannot be used for other topics such as the common application area of sorting algorithms. Each system supports exactly one type of image compression algorithm: run-length encoding (RLE), Quadtree image segmentation and JPEG.
The tools are very helpful in these application areas; however, they are also restricted to the embedded features. Several other approaches for image compression are missing. For example, the image compression algorithms employed in the popular GIF or PNG are not explained. The tools also do not support newer compression approaches such as *Wavelets* [35], fractal compression using iterated function systems or weighted finite automata [41, 96]. Comparing the performance of the covered algorithms is made more difficult due to their implementation in separate tools.
These attributes are similar for most topic-specific systems. Developers of such specific systems possess a precise knowledge of the animation content. Therefore, they can optimize the systems to provide the best possible illustration of the topics as they perceive it. Note that in general, the perception of the most helpful or appropriate way of presentation may differ between the developer, visualizer and user. Most topic-specific animation systems do not give the user a chance to adapt the display to his or her preferences.
Topic-specific systems may often prevent visualizers from influencing the animation display. In an

educational context, the visualizer role will typically be assumed by the educator, while the users will be the students. The main advantage of topic-specific systems for visualizers is that they are freed from having to generate the animation. Additionally, most topic-specific tools will allow the user to change the input parameters, for example by providing a different image to be run through the compression algorithms in the packages described above. This allows users to experiment with different data sets to get a better understanding of the algorithm's behavior.
The visualizer should carefully test the system before recommending it to his or her students. Apart from different perceptions of "good" presentation, there may also be fine but important differences in the interpretation of the underlying algorithm. For example, there is a wide variety of ways for choosing the pivot element in the *Quicksort* algorithm. If the approach used within the tool is not identical to the one used by the visualizer in the educational context, the users may become confused. Even worse, such small differences may go unnoticed by both visualizer and users. In this case, they may later find that they have developed different understandings of finer aspects of the covered topic. This may be especially harmful for the user in an examination context.

2.4.2 Manual Generation in a GUI

GUI-based generation is typically found in presentation tools such as Microsoft PowerPoint™ or StarOffice Impress™. Most presentation systems can also be used for algorithm animations, although they are not specialized for this usage. Figure 2.4 shows a typical distribution of the roles in GUI-based animation generation.

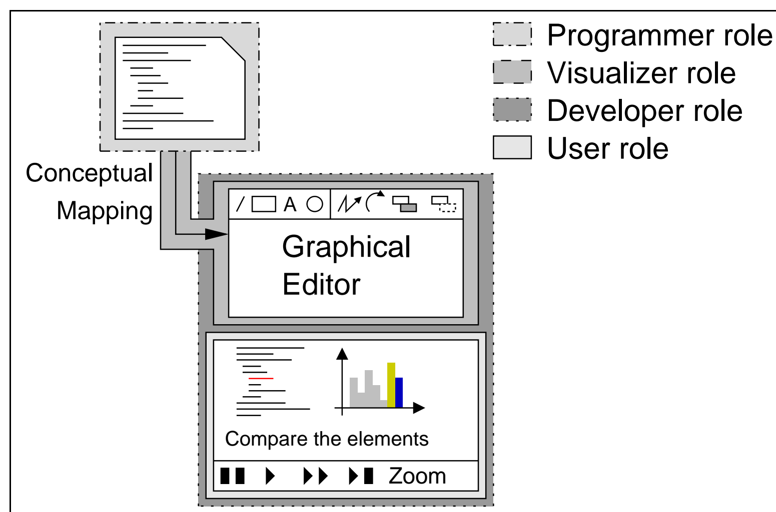

Figure 2.4: Role Distribution in GUI-Based Systems

The *programmer* implements the algorithm to be animated. The *developer* is responsible for implementing both the graphical editor and the user interface for displaying the animation. The task of the *visualizer* is to perform a conceptual mapping that transforms the algorithm into a visual representation. The GUI usually offers a set of graphical primitives and animation effects acting on them. The visualizer generates an animation by assembling a set of graphical primitives and transforming them using the animation effects. The schematic system shown in Figure 2.4 offers the basic graphic primitives line, rectangle, text and circle. It also offers moving, rotating, changing

the color and the visibility of graphical primitives, as shown from left to right. Finally, the *user* can use the graphical front-end to view the animation.

The graphical front-end used for displaying the animation may be the same as used for generating the animation. In the figure, the animation display contains the standard control elements including a zoom operation. The graphical editor used for generating the animation may also be accessible to the user.

As an example of GUI-based animation generation, we consider manual generation using a standard presentation tool such as Microsoft PowerPoint™ or StarOffice Impress™. Several examples of this can be found in the World-Wide Web, for example [158, 126, 22]. The decision to use a presentation tool is usually influenced by the fact that the tools are already being used for a slide-based presentation. Embedding animations in the same tool used throughout the presentation is the most "natural" approach. This also prevents possible problems regarding switching tasks during the presentation or lack of memory. The latter is especially important when performing presentations on notebooks which usually have less RAM than full-fledged desktop computers.

Most systems employing a graphical front-end for animation generation adhere to the *WYSIWYG* approach – "what you see is what you get". Thus, the visualizer will usually see a direct reflection of his or her actions in the graphical display. This ability to immediately see the effects of any given action makes the approach very helpful, especially for novices or laypersons. The level of abstraction expected of the visualizer is also lower than in most other approaches.

The visualizer has a high degree of freedom in designing the animation, as the algorithm and the animation are completely separated. Thus, he or she can choose any level of abstraction, and adapt the graphical properties of the display and operations to his or her preferences. For example, a manually generated animation may gloss over some aspects of an algorithm and focus on the more salient features, as perceived by the visualizer. Both the extent and the way in which this is used may differ between visualizer. For example, complex operations may be decomposed into their basic components and executed step by step.

Another strength of GUI-based generation is that the visualizer can easily enhance the animation with explanatory notes. As the animation is decoupled from actual implementation code, the explanation does not have to be embedded in the code, and can be arbitrarily terse or verbose, according to the visualizer's interests.

Finally, it is very easy for visualizers to illustrate the behavior of a given algorithm when certain common bugs are present. For example, invalid partitioning strategies for the Quicksort algorithm can be presented, illustrating how sorting fails. The visualizer does not have to be concerned with runtime considerations. Combining this approach with focusing strategies also enables the visualizer to highlight the effect of typical coding mistakes. Infinite loops due to incorrect recursive calls or loop conditions only impact the visualization. Incorrect pointer assignments also cannot corrupt the actual algorithm. The commonly held opinion that one learns the most from mistakes can be tested by presenting such typical coding errors.

Most graphical systems have a proprietary storage format, for example the format used by Microsoft PowerPoint™. Import of and export to this format is usually supported in related tools. However, the format is usually ill suited for automatic generation. Part of this is due to the lack of freely available format documentation or the definition of the subcomponents. For practical purposes, this means that the visualizer usually has to generate the animation manually. There is often no free API that the visualizer can use for generating the required format.

Changing a single value of the algorithm's input data may require editing the whole animation within the GUI. Thus, this type of animation generation is more suited to generating a prototype

animation that serves as a guiding example. This animation may be shown and discussed within a presentation. The presence of only one animation of a given algorithm and the lack of support for quickly adapting the animation to different parameters may reduce the learning chance of the user. The time required to generate a full animation manually should not be underestimated by the visualizer. The precise amount of time needed differs between reports. Our experience with the ANIMAL system presented in chapter 5 indicates that visualizers already familiar with the given system require roughly six hours for generating an "average" algorithm animation. This figure assumes that the visualizer has already decided on the presentation and thus can directly start generating the animation. Interestingly enough, this time span corresponds to the required time for generating a "low-fidelity" animation using papers, pen and scissors, as reported by Hundhausen and Douglas [84].

2.4.3 API-based Generation

API-based generation involves method invocations using a special visualization application programmer's interface (API). The extent of support for primitive types and effects depends on the underlying visualization API.

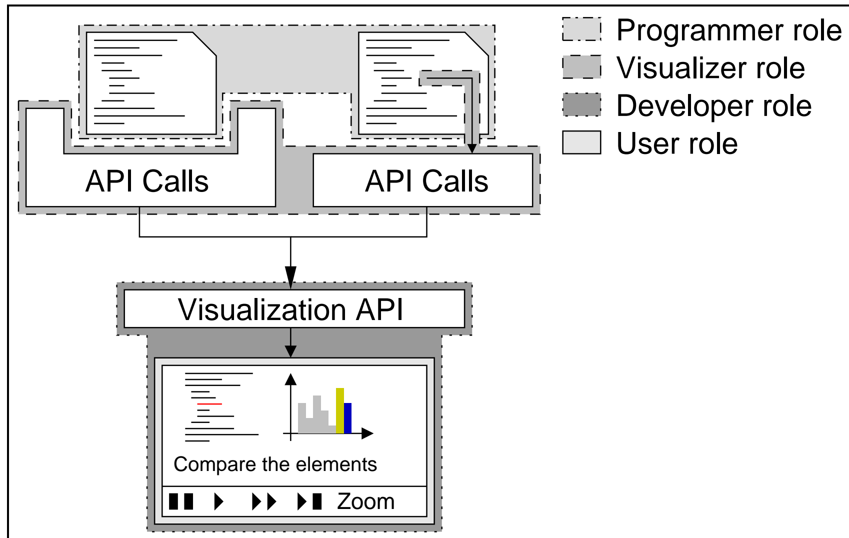

Figure 2.5: Role Distribution in API-based Systems

Figure 2.5 illustrates the different roles when using this generation approach. The *programmer* is responsible for implementing the actual algorithm. The *developer* has to provide an appropriate implementation of the visualization API, as well as a front-end for displaying the resulting animation. The *visualizer* has to ensure that the appropriate API methods are invoked. The *user* interacts with the display front-end.

API invocations can be implicit or explicit. The upper left section of the figure symbolizes implicit API method invocations. This may be supported by supplying a special class that incorporates method invocations instead of a standard class. Note that this requires that the original implementation expects appropriate objects which can be replaced by the visualizer with customized objects. An example of this approach is outlined in [138]. Implicit API invocation will in most cases still

require code modifications, but may be possible without touching the actual algorithm implementation. Explicit invocations, on the other hand, modify the underlying implementation code by adding explicit method invocations. This approach is shown in the upper middle of the figure.

API-based generation has several advantages. Once the visualizer has implemented the appropriate method invocations, a new animation can be generated by simply invoking the algorithm. Illustrating the behavior of the algorithm for different parameter values requires only passing in the appropriate parameters. Depending on the complexity of the algorithm, the visualizer may also be able to generate a new animation during a presentation. Users may also profit from this fact if the implementation is available to them.

A good visualization API may contain many methods that allow the visualizer to assemble the animation easily. Finally, method invocations - whether implicit or explicit - are a "clean" way of modifying the underlying algorithm. Provided that the API method names and parameters are not too cryptic, the original algorithm may remain readable even when using explicit invocations.

However, the API-based generation also has disadvantages. First of all, the algorithm to be animated usually has to be implemented in the same programming language as the visualization API. There are a small number of exceptions, such as Java's ability to invoke "native code" methods using the JINI interface. However, for most practical purposes, the visualizer has to settle for using the same programming language.

API-based generation requires the visualizer to have a certain amount of programming skill in the API's programming language. GUI-based generation, on the other hand, only requires that the visualizer understands the algorithm and is able to "draw" it. Finally, API-based generation limits the visualizer to the methods provided in the API. The extent to which this is problematic depends on the extent of the API.

2.4.4 Scripting-based Generation

Scripting-based generation uses an intermediate format for representing the animation. Typically, this format consists of a set of text-based commands for generating or modifying graphical primitives. The title derives from the similarity of the output to a "program" written in a scripting language such as *PHP* [16] or *Perl* [208]. However, this should not be taken to imply the format is a full-fledged programming language capable of handling standard algorithms. Typically, the expressiveness of the scripting language is strictly limited to animation purposes. Methods, variables, or loops are usually not supported.

Figure 2.6 illustrates the distribution of roles in scripting-based generation. The *programmer* implements an arbitrary algorithm or program. The *visualizer* is responsible for generating the animation in either of two ways. Firstly, he or she may substitute one or more parameter objects with specific objects that produce scripting output, as illustrated in the top left part of the figure. Alternatively, he or she may add explicit method invocations or other statements that result in the generation of scripting code within the algorithm. This output is analyzed by the scripting parser and displayed in the user front-end. Both parser and front-end have to be implemented by the *developer*. The *user* merely interacts with the front-end for displaying the animation.

Modifying an algorithm to generate scripting code is very similar to adding method invocations in a special visualization API, as described in section 2.4.3. Instead of reading the method documentation of a visualization API, the visualizer has to become familiar with the syntax of the scripting language. The scripting output can usually be stored in a file and loaded by the parser, allowing the visualizer to store a set of animations. However, the compiler of the programming language can

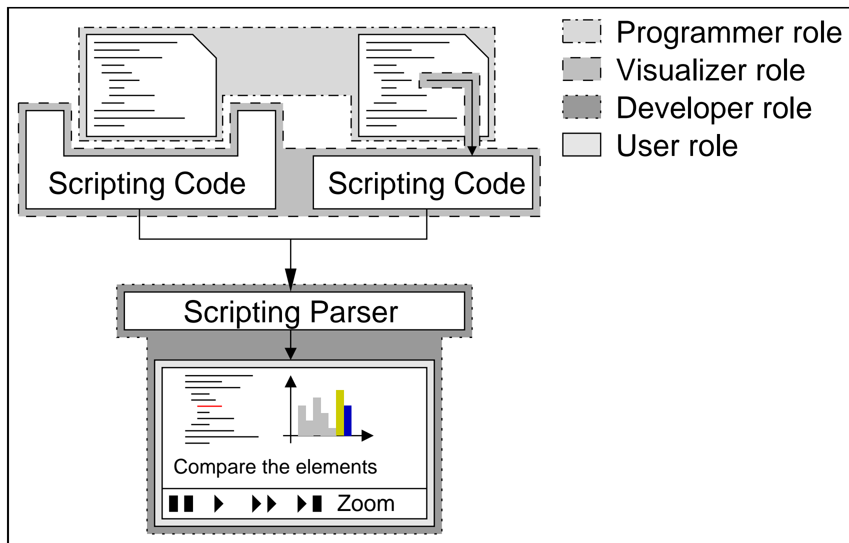

Figure 2.6: Role Distribution in Scripting-based Systems

only detect syntax errors concerning the invocation of the statements used for generating the scripting code. Thus, a syntactically correct underlying algorithm may generate syntactically incorrect scripting code.

Generating scripting code may be easier than figuring out appropriate API method invocations, depending on the visualizer's programming skills and the extent of the API. Scripting offers the advantage of storing animations to disk, so that visualizers can prepare a set of animations. If an animation using the requested parameter values is already available, it can be loaded directly; otherwise, the algorithm has to be invoked according to the parameters. One especially helpful feature of scripting based on stored scripting code is that the visualizer can fine-tune the animation by manually editing the stored code. Once the desired result is reached, this can be reinserted into the implementation code for generating the animation. Thus, the visualizer may avoid having to recompile and execute the algorithm to determine the output. Depending on the characteristics of the algorithm, this can save a huge amount of time.

2.4.5 Declarative Visualization Generation

Declarative generation regards visualization as a mapping from a given program state to graphical representations. It uses abstract mathematical expressions which can result in complex visual representations. The declarative approach was introduced in [164] and is summarized in [163].

Figure 2.7 illustrates the distribution of roles. The *programmer* develops the program code. The *visualizer* defines a mapping from program states to their graphical representation, and the *user* examines the results of the visualization. The *developer* has to implement a system that analyzes the mappings and generates the representation, as well as a front-end for displaying the animation. The conceptual execution model updates the display whenever the program changes state in a relevant manner. There may also be a special directive allowing the visualizer to deactivate the visualization update [44].

Declarative visualizations are usually also embedded into the code as abstract specifications, for

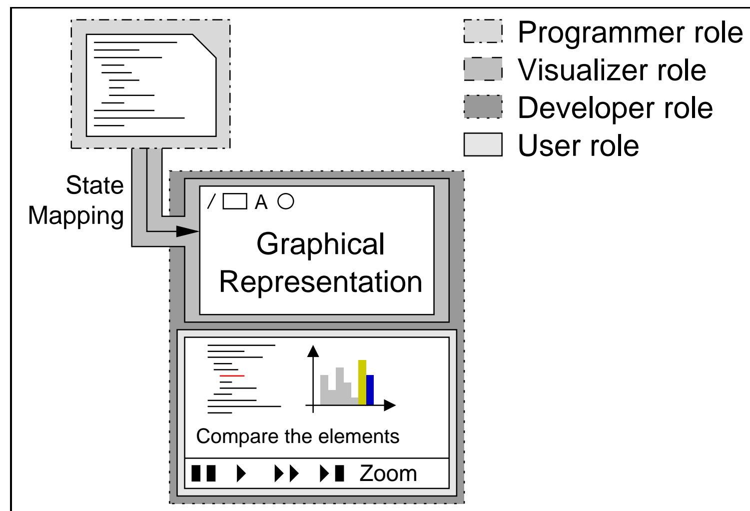

Figure 2.7: Role Distribution in Declarative Visualization

example in a special comment notation. A special compilation or extraction mechanism is required for mapping these abstract specifications into a computation. However, the developer also has to provide the mechanism for extracting the specifications, as well as a front-end for displaying the visualization itself. Thus, declarative visualization tends to shift part of the work required for generating visualizations from the visualizer to the developer.

Mapping the program state to a graphical representation may be difficult for visualizers not schooled in mathematical reasoning. Declarative visualization system such as *Pavane* [165] and *Leonardo* [39] use predicates for specifying the mapping. Figuring out how to read the predicates and use them appropriately may present an initial obstacle.

2.4.6 Generation By Code Interpretation

Some animation systems offer a direct visualization of algorithms from the underlying code. There are three main approaches for achieving this. The first approach relies on a debugger used for retrieving the current state of the program. The second approach preprocesses the underlying source code and modifies it before sending it along to the compiler or interpreter. Alternatively, the code may also be interpreted "as is". The latter two approaches may require slight modifications to the code, such as using a customized version of input and output methods.

Figure 2.8 illustrates a typical role distribution for code interpretation-based animation. The *programmer* implements the algorithms without regard to their visualization. The *developer* is responsible for implementing an appropriate technique for analyzing the code and interpreting it, as well as a user front-end for displaying the animation. The *user* interacts with the front-end as usual.

The *visualizer* role depends on the mode of generation. Systems that rely on debugger data, such as *DDD* [212] or *Kami* [205], need no modification of the actual source code. The only interaction for customizing the display is usually the addition or removal of breakpoints for retrieving the current program state. If the breakpoints or a full session can be saved, as in *DDD* [212], the visualizer can preselect them for the user. Otherwise, the user is responsible for the animation display.

Systems that preprocess unmodified algorithm code, such as *WinHIPE* [128] and *ZStep 95* [114],

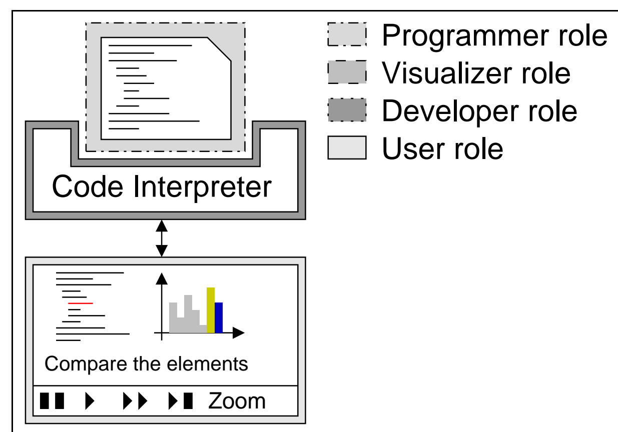

Figure 2.8: Role Distribution in Code Interpretation Systems

do not require a visualizer. The user will typically be able to choose the amount of detail of the display using a special control bar.

The main strength of code interpretation-based animation is certainly the tight connection between the code and its visualization. Errors in the graphical display are nearly impossible, apart from those caused by bugs in the code interpretation modules. Thus, the user always sees the actual implementation and can analyze the exhibited behavior. Changes to the implementation code will be reflected in the animation, once the code has been recompiled or reloaded for interpretation. Users can easily change aspects of the programmer's code to determine the actual effects this causes.

The main weaknesses of the approach lie in its restricted expressiveness and level of abstraction, as well as a possibly severely restricted granularity control. The expressiveness of the animation is restricted to the display of the actual code execution. Explanatory texts or cross-references cannot easily be added, so that the code has to "speak for itself". This is probably sufficient in many applications areas such as code debugging. However, certain aspects of the implementation may be difficult to understand based on the implementation alone. For example, moderately advanced sorting algorithms such as *Shellsort* and *Mergesort* may confuse the user when only a display of the changed elements is presented.

It is often very difficult, if not outright impossible, to change the level of abstraction. Users may find it very difficult to understand some algorithms containing very efficient implementations of comparatively "mundane" subtasks. For example, Dijkstra's Shortest Path algorithm requires the determination of an unvisited node with minimum connection costs from the start node in each loop iteration. It is comparatively easy to state this abstractly, as shown in the previous sentence. A standard approach for determining the next node involves storing the connection costs in a priority queue which has to adapt to changed connection costs in each loop iteration. Visualizing this segment of the algorithm may confuse users not yet familiar with Dijkstra's algorithm and mix *implementation details* and *algorithm issues*.

The user interface may provide only a limited selection of operations for adjusting the granularity of the display. For many educational applications, it might be preferable to take the result of a given set of code lines for granted. In Dijkstra's algorithm, it may be beneficial to assume that the node has been determined "somehow", so that users can focus on the algorithm's base idea. The user might also want to skip lengthy initialization code and focus on the actual problem. In some cases,

this may be possible by setting or erasing breakpoints, as in *DDD* [212], or using special "show value" controls as in *ZStep 95* [114]. If this operation is not supported, the user has to sit through the full initialization process.

Finally, code interpretation-based animation is not suitable for all possible application areas. Obviously, it can only be used if an appropriate implementation of a given topic is available. The programming language usually has to match the programming language used in the interpretation tool. Debugger-based tools may not have this restriction, but will usually require an executable file which may preclude the animation of programs implemented in interpreted programming languages such as Smalltalk or Lisp.

2.5 Evaluation of Representative Tools

Price *et al.* [156] state that more than 150 animations and authoring systems have been developed. However, this number stems from 1998. Since then, several new systems have been introduced, while others may have vanished. For example, several of the links valid in 1998 have become invalid. On the other hand, systems newer than the summer of 1998 are not included in the count.

The "Complete Collection of Algorithm Animations" (*CCAA*) [33] contains a set of linked Web pages enumerating the animations found by Peter Brummund in 1997/1998. Each animation is characterized by its title, author or institution of origin, the set of provided algorithms, special requirements and a short overview of what the animation offers. The site is organized by algorithm types on the one hand and site locations on the other hand. Adding a new animation would thus require changing at least two different explicitly encoded Web pages, one each for the algorithm and the site listing.

Peters [151] presents a taxonomy for algorithm animation systems adapted from the taxonomy introduced by Price, Baecker and Small in [155, 156]. The appendix of the Master's Thesis classifies selected software into this modified taxonomy. The appendix is available on a CD-ROM and as a collection of Web pages.

A topic with such wide-spread interest as software visualization and its subarea of algorithm animation cannot fully be classified by static media. Hard-coded web pages as provided by Brummund are too cumbersome to maintain manually, and CD-ROM based approaches prevent the direct addition of new links. Users interested in the field profit from a dynamic listing of available animation software. In order to be easy to maintain, the listing must allow for easy updating and addition of elements. At the same time, it should also offer a certain amount of classification of the animations, for example by topic. A full classification as employed by Price [156] or Peters [151] is probably more than the average user requires. However, the user should be able to easily locate certain animations or topics, such as "all sorting algorithms".

We have recently begun installing a new algorithm animation collection on the Internet [168] that addresses these issues. Users can choose the types of animations they want to see by the following criteria: topic classification, for example "sorting", language used in the animation (if any), animation system, or all animations. The pages are dynamically generated by a set of PHP [16] scripts. The underlying data is extracted from a MySQL database [127]. The web pages therefore always reflect the current state of the collected data, and do not require manual maintenance.

Evaluating all algorithm animation systems or applets available on the Internet is beyond the scope of this thesis. Instead, we constrain the discussion to some of the more relevant systems or applets. The main consideration of "relevance" here rests with the supported features of a given system or

applet. Thus, the system chosen may not always be the most popular or well-known tool. Indeed, we will not be able to mention several tools that some reader might consider relevant. However, our focus in this evaluation lies not on a concrete tool's qualities or shortcomings, but rather on typical features found in specific types of systems.
In the remainder of the chapter, we introduce and evaluate some of the applets and animation systems available on the Internet. The central difference between applets and systems is that the applets presented are usually hard-coded for the given topic and possibly also a fixed input set, while animation systems are capable of handling a larger set of algorithms and animation topics. The requirements for an "ideal" algorithm animation system stemming from evaluating other tools are given in the next chapter.

2.5.1 Algorithm Animation Applets on the WWW

In this section, we examine typical algorithm animations presented as applets. As stated in the previous section, the choice of concrete applets stems from the set of features they offer. Thus, the applets presented may not be the most popular or publicized applets. Instead, we select "typical" representatives for a class of applets to discuss the typical attributes, starting with the least complex applets. Whenever possible, a set of links to related applets is also given.
Furthermore, applets are excluded whenever they can be recognized as display front-ends of animation systems. Full-fledged animation systems are discussed in the following section. This section, on the other hand, focuses on hard-coded animations. Thus, practically all applets discussed in this section belong to the category of *topic-specific* algorithm animation, as described in section 2.4.1. Note that this does not imply that the content of the applet is fixed to one parameter set or a single algorithm. For example, the user may be able to select a concrete sorting algorithm or customize the input values.

Non-Interactive Applets

The simplest forms of algorithm animation applets are fully automatic with no special user interaction. As a typical representative, we examine the illustration of *Insertion Sort* by Sekisita [190]. Figure 2.9 shows a screen shot of this applet. The elements of the array are represented by colored bars separated by a small amount of space. The current insertion operation concerns the two highlighted elements.

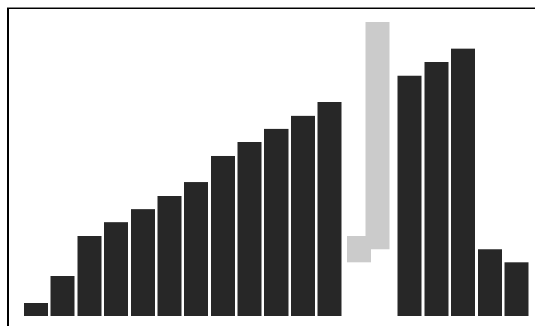

Figure 2.9: Insertion Sort Applet by Hiromasa Sekisita [190]

The animation uses blue for the array elements not affected by the current operation, and red for the currently affected elements. Using blue for inactive and red for the currently active elements is also common in several other applets. The grey values of the elements correspond to 39 for the blue elements, and 87 for the red elements. Color-blind users may find it very difficult to tell the two colors apart. This problem becomes even more relevant if a screen shot of the applet is printed. The colors in Figure 2.9 have been adapted to printing purposes by brightening the currently active array elements to 204.
The Insertion Sort applet does not support any user interaction: it starts immediately once the page is loaded and continues until the values are sorted. Thus, if the user is inattentive when the page starts, he or she may miss parts of the animation. This can easily happen if the download of the page takes a certain while. The algorithm used by the applet is fixed and cannot be influenced by the user, although the same author also provides applets illustrating Bubble Sort [189] and Selection Sort [191]. The author also provides a special page which allows the user to choose the target algorithm to visualize [192]. The supported algorithms include *Selection, Bubble, Insertion* and *Quicksort*, *Depth / Breadth First Search* and *Dijkstra's Algorithm*, the string searching algorithms *Brute Force, KMP* and *Mischar Search*, three tree traversal algorithms, *Binary Search* and *Dictionary Search*.
There are many similar applets, mainly for illustrating sorting algorithms. One of the first such Java-based applets was implemented by James Gosling and Kevin A. Smith of Sun Microsystems [70]. Included in the Java development kit distributions in the `demo/applets` folder, these applets have inspired many authors. Figure 2.10 shows an example screen shot illustrating three different sorting algorithms running in parallel. Many educators have been using similar algorithm animations for several years. However, most of these animations have never been published and thus remained bound to the context they were presented in.

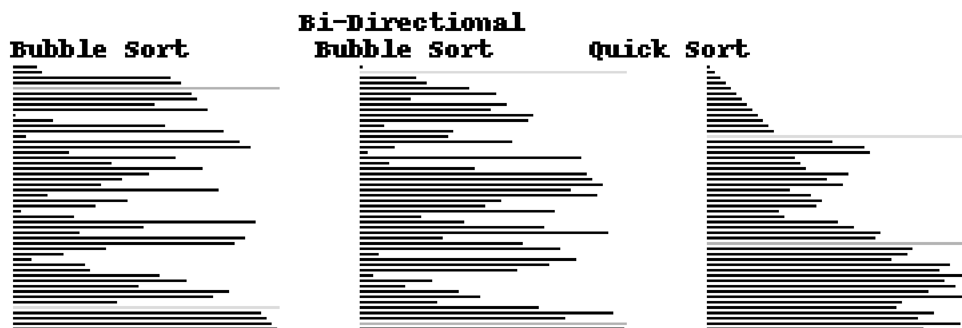

Figure 2.10: Screen Shot of Gosling and Smith's Applet for Sorting Algorithms [70]

Comparing Figure 2.9 and 2.10 shows both similarities and differences. Sekisita's animation portrays the array as a set of colored bars in horizontal orientation. Gosling and Smith's implementation, on the other hand, uses simple lines (often also called "sticks") with vertical orientation. Sekisita highlights elements currently being exchanged by changing their color and lifting them up from the array's base line. Gosling and Smith simply update the display with the changed state of the array. Sekisita's applet starts automatically once the page is loaded, while Gosling and Smith's applet is started when the user clicks on it. Furthermore, Sekisita only presents a single sorting algorithm on each page, while Gosling and Smith include three applets on their page: *Bubble Sort, Bi-directional Bubble Sort* and *Quicksort*.

Finally, Gosling and Smith use two color lines to illustrate the low and high index of the search - in their own terms, the “high” and “low water mark” of the current sorting interval. The low water mark is colored in red, and the high water mark is in blue. For *Bubble Sort*, the role of the lines is exchanged: the blue line describes the value of the innermost loop variable, while the red line represents the value of the outer loop variable. The same problems with color perception as described above apply to this selection. Again, the colors have been adapted for this printed thesis, with the blue line shown in a darker shade of grey than the red line. The standard array elements are painted in black. Thus, the Bubble Sort algorithm shown in Figure 2.10 currently examines the fifth array element and has already finished sorting the largest five array elements. Note that the two special lines *always* have a fixed width, obscuring the actual array element's value at the current index.

Many applets have been inspired by the freely available code of Gosling and Smith's animation. For example, [69] contains a collection of fourteen sorting algorithm animations including the most common sorting algorithms, including various variants of Merge Sort and Quicksort. Each algorithm is a separate applet that merely inserts a different sorting algorithm into Gosling and Smith's implementation. Another example is provided by Andrew Kitchen [103], covering Bubble Sort, Quick Sort, Odd-Even Transposition Sort and Shear Sort. The web page also states the complexity classes of the algorithms.

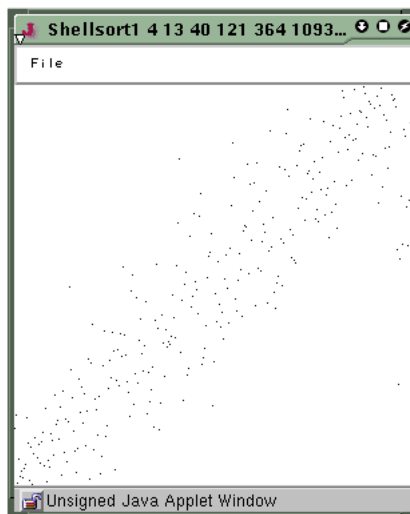

Figure 2.11: *Shellsort* Animation taken from [186]

Another popular presentation type for sorting algorithms uses 2D dot plots. Figure 2.11 shows an example of this, provided by Robert Sedgewick [186]. Here, the x axis represents the array index, and the y axis the value of the element. The situation in Figure 2.11 illustrates a roughly sorted array where most dots are set near to a diagonal line from the bottom left to the upper right corner. The associated applet offers a set of different increment sequences for *Shell Sort* and the related *Brick Sort* and *Shakersort* algorithms.

The animations presented so far use different activation techniques for starting the animation. Some animations start once the applet is loaded, others wait for the user to click on them, and Sedgewick's animation requires user inputs before it starts. However, none of them allow the user to control the

animation's progress once the animation has been started. There is no *pause, stop* or *single step* facility. Thus, the user has to pay constant attention and be able to follow the algorithm at the pace set by the animation. Several other animations share this limitation, for example the animations for breadth- [187] and depth-first search [188] by Hirosama Sekisita. Taisuke Fukuno's animation of the *Towers of Hanoi* [60] shown in Figure 2.12 also runs until all eight disks are sorted and then restarts after a short delay. The disks possess different sizes and colors, with the smallest disk being the darkest. A counter at the bottom left keeps track of the current step number.

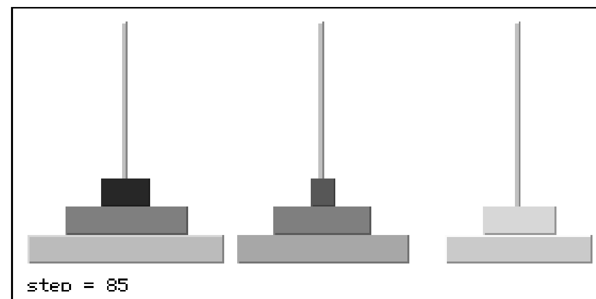

Figure 2.12: *Towers of Hanoi* Animation taken from [60]

Applets with some user control

There are many examples of applets which support elementary user controls. We focus on two to outline the range of what we consider elementary controls. Firstly, Tanya Filipova's animation of a solution approach for the *Traveling Salesman Problem* [58] allows the user to generate a new graph, run the animation or perform a single adaptation step. A sample image of this animation is shown in Figure 2.13.

Gabrielle Ortmann's Sorting Algorithm Demo [141] is similar to the sorting demonstration by Gosling and Smith, but combines the different algorithms in a single applet. It also allows the user to toggle the animation speed between "fast" and "slow" and stop the animation. The algorithms offered include Insertion, Bubble, Bidirectional Bubble, Quick, and Shellsort.

More refined applets give the user better control over the animation display. As an example, we regard the Bubble Sort animation provided by Alejo Hausner [73]. Figure 2.14 shows a screen shot of this applet. The animation uses a standard sticks view to portray the array elements and highlights the currently regarded elements. In Figure 2.14, these elements are shown in a slightly brighter shade of grey than the other array elements. The colors of the screen shot have been improved for printing purposes; color-blind users might find it difficult to tell the array elements apart in the original animation.

The user can control the animation by reinitialization, a run, pause and single step button. Additionally, the animation speed is controllable by a slider. The two scrollbars on the left and right side allow the user to adjust the animation's zoom factor and scroll within the display. The applet offers three different input sets: random, descending or almost sorted array elements. The name of the algorithm and the number of events performed are shown at the top of the animation applet.

Hausner also offers other comparable animation applets for Merge Sort [74] and Quicksort [75]. The Merge Sort applet controls are identical to the Bubble sort applet. The Quicksort applet offers three different types of display: random, sorted data in a sticks view as shown in Figure 2.14, and

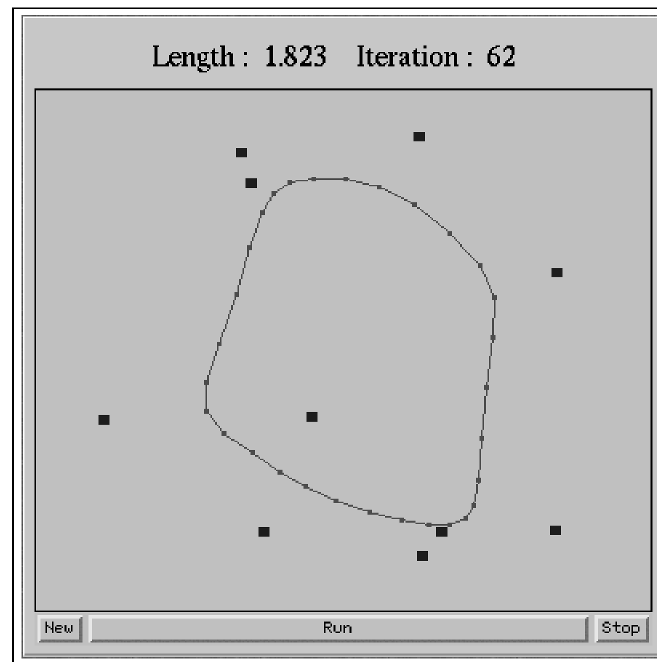

Figure 2.13: *Traveling Salesman Problem* Applet by Filipova [58]

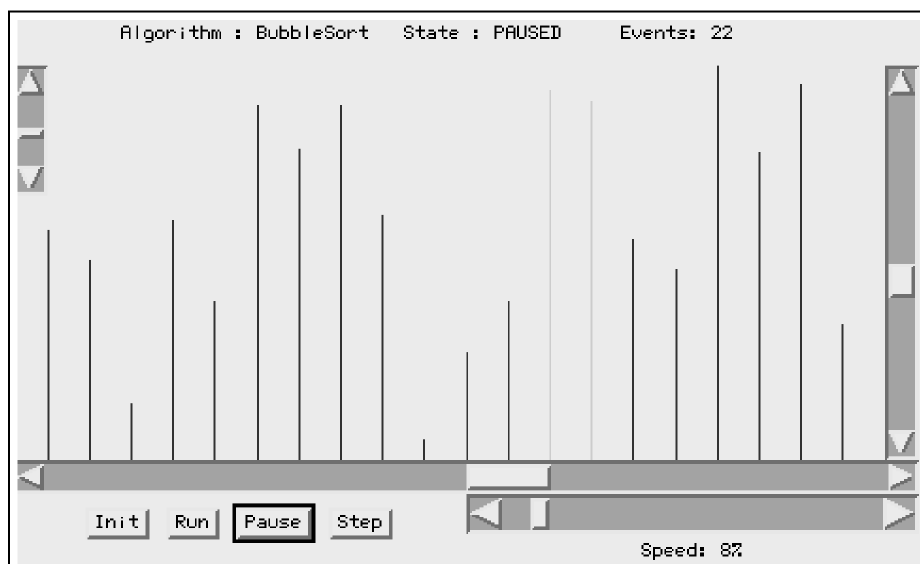

Figure 2.14: Bubble Sort Screen Shot taken from [73]

random data with a dot plot similar to the one shown in Figure 2.11. Contrary to the other applets by Hausner, this applet does not show the number of events.
Sandeep Poonen [154] provides an applet that uses odometers to display the number of move and comparison operations. The algorithms offered cover Insertion Sort, Selection Sort, Bubble Sort, Shaker Sort and Shellsort. The data is either random or in descending order. The user controls are limited to single step, run (called "finish" in the applet) and pause.

Applets with accompanying extra information

Extra information can be placed in the embedding web page or encoded in the applet. For example, David Eck's *xSortLab* [52] applet incorporates two lines of explanatory text at the bottom of the applet. The applet allows the user to control the display speed. Apart from choosing between "go" and "step" mode, the animation can also be restarted. The user can select two different animation speeds: normal and "fast". Figure 2.15 shows a screen shot of the applet in action.

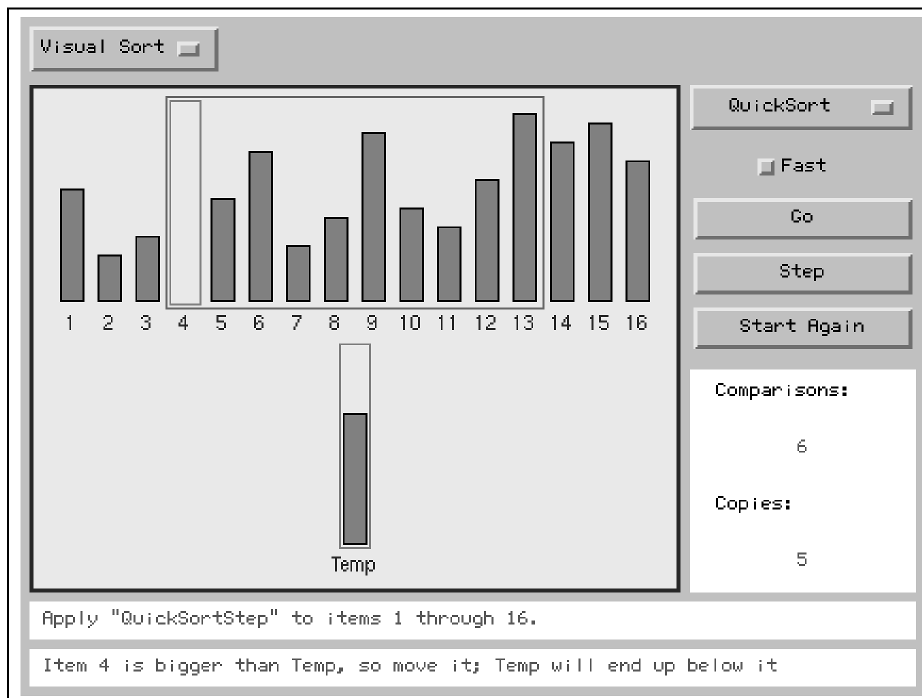

Figure 2.15: Quicksort Animation Screen Shot from Eck's *xSortLab* applet [52]

The *xSortLab* applet contains three different visualization modes: the "visual mode" shown in Figure 2.15, a "timed" mode which displays the elapsed time and a "log" which stores the statistics of the sorting algorithm after completion. The current number of comparisons and copying operations are also given. Finally, the user can select the algorithm to watch, choosing between Bubble Sort, Selection Sort, Insertion Sort and Quicksort.
John Morris provides various animations for his Programming Languages and System Design course [123]. The user can run the animation fully, stop it or perform a single step. The delay between animation steps can be chosen from a set of predefined delays. The web pages from which

the applets are started contain a description of the algorithm including a complexity analysis. Figure 2.16 shows a screen shot of a Radix Sort animation with the stop, run, step and skip buttons.

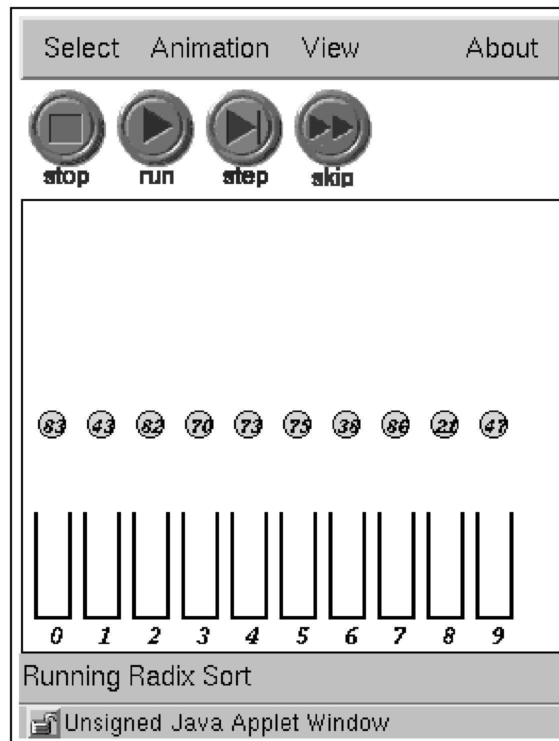

Figure 2.16: Radix Sort Animation Screen Shot from [9]

Some of the manually generated animation applets listed by Morris embed explanations directly into the animation, for example the applets for matrix chain multiplication [7], optimal binary search trees [124] and hashing [4]. The latter applet additionally allows the user to choose the probing strategy between linear or quadratic probing with c={1, 2, 4}.
Several of the applets provide a code view facility, either in the same window as the rest of the animation or in a separate window. Examples of this include Huffman encoding and decoding [6], heaps [5], various sorting algorithms including Insertion Sort [8], Bin Sort [3], Quick Sort [203], Dijkstra's shortest path algorithm [137] and some tree animations [136, 115]. The sorting applets allow the user to choose the input data from a predefined list of ascending, descending or random data.
Peter Brummund's sorting algorithm animator [32] goes a step further by letting the user adjust the graphical properties of the display. Color adjustments include the color of the foreground, background, sorted or inspected array elements and the highlight color. The number of blocks can be typed in by the user. The properties of the data can be toggled between random, ascending and descending order. Figure 2.17 shows a screen shot of the applet.
The animation is controlled by a pause and stop button, together with a speed selection slider. The number of comparisons and swap operations is given below the animation, as well as an optional explanation. The user can select the sorting algorithm from Bubble Sort, Insertion Sort, Merge Sort, Selection Sort, Shellsort and Quicksort. The Java source code of the algorithm including

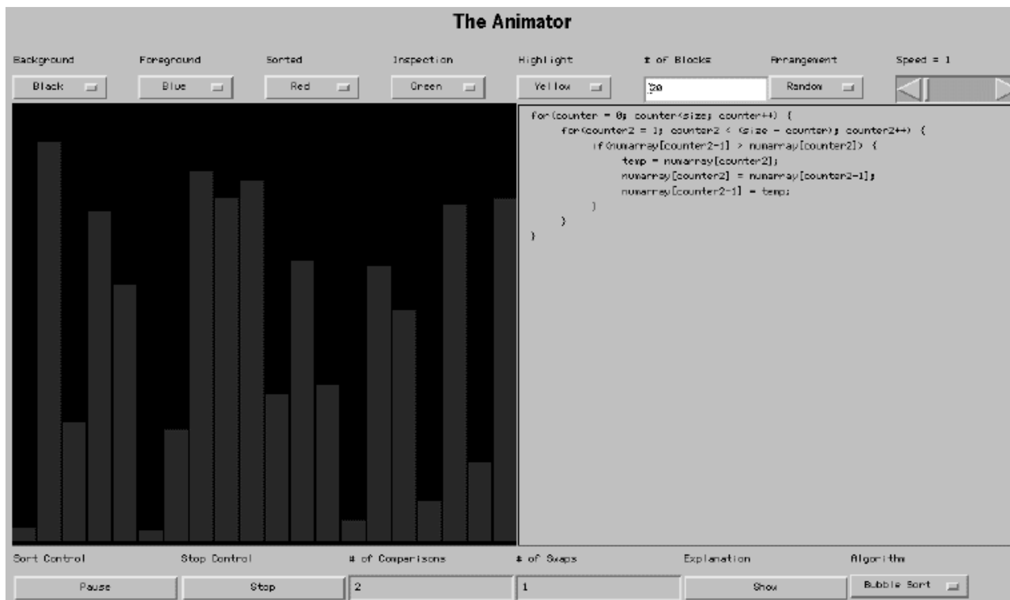

Figure 2.17: Bubble Sort Screen Shot taken from [32]

indentation and highlighting is also shown in the applet.
Athanasios Papagelis provides animations of Prim's [147] and Kruskal's [146] algorithm for generating the minimum spanning tree as well as for backtracking [145]. Figure 2.18 shows an example screen shot. The user can toggle between single step mode and a complete solution and may request a new problem. The pseudo code of the algorithms is shown on a separate page. The algorithms are described on the web page containing the applet. The backtracking animation also allows the user to adjust the problem size between 4 and 16.

Applets with user input

Hans-Werner Lang provides several web pages for selected algorithms for his German course on algorithms [109]. The animations can be divided into two groups. Shellsort, Quicksort and the Traveling Salesman Problem are animated automatically with very little user control except for typing in the number of elements and selecting the ordering of the data. Figure 2.19 shows a screen shot of Quicksort, with the number of exchanges shown at the top.
Some pages also contain a small selection of interactive "animations" prompting the user to select the appropriate elements. These interactive applets cover the Warshall and Prim algorithms, Heapsort and the partitioning of Quicksort shown in Figure 2.20.
There are several comparable applets that prompt the user for input data. For example, the Delaunay triangulation applet by Daniel Mark Abrahams-Gessel [1] shows the triangulation process based on the user's mouse clicks within the canvas. However, it only provides a reset button and no other controls. Similarly, Connie Peng's convex hull demonstration [150] uses user clicks as input to Graham's Scan with only a pair of Clear and OK buttons. The VoroGlide applet by Christian Icking *et al.* [87] also uses user clicks for Voronoi, Delaunay and convex hull calculation. However, it also allows the user to drag or remove individual points and directly updates the display.

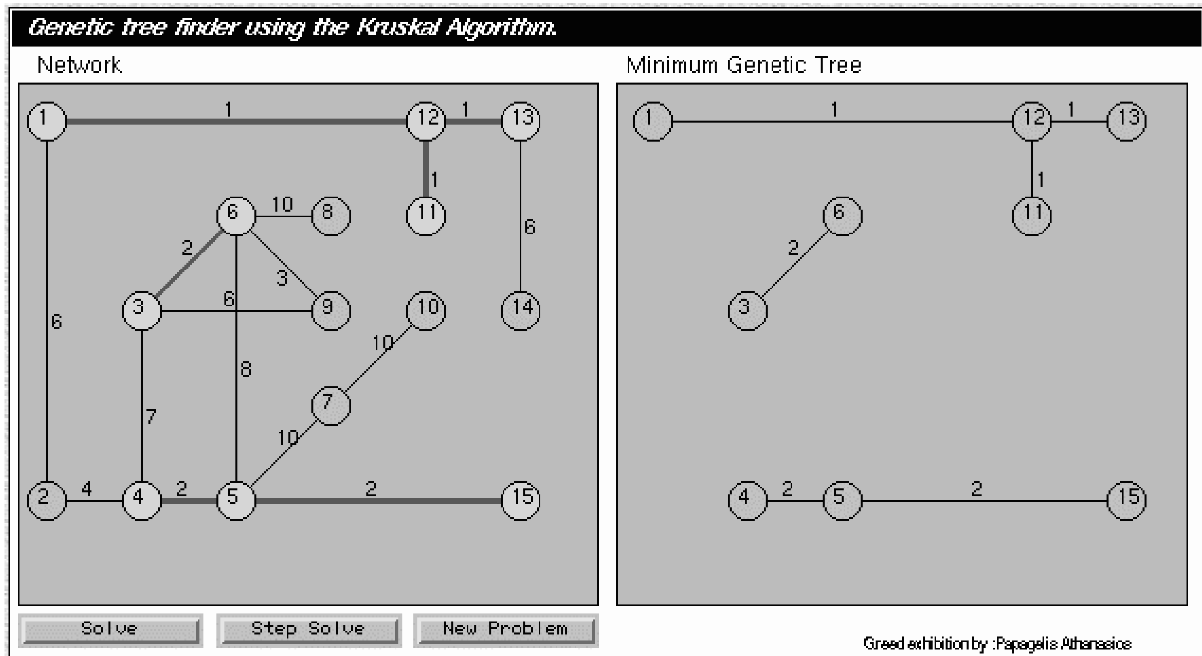

Figure 2.18: Screen Shot taken from Papagelis' *Kruskal* animation [146]

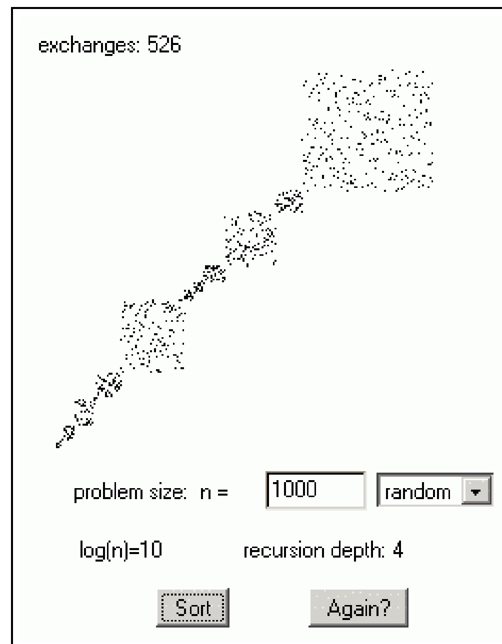

Figure 2.19: Quicksort Screen Shot from Lang's applet [109]

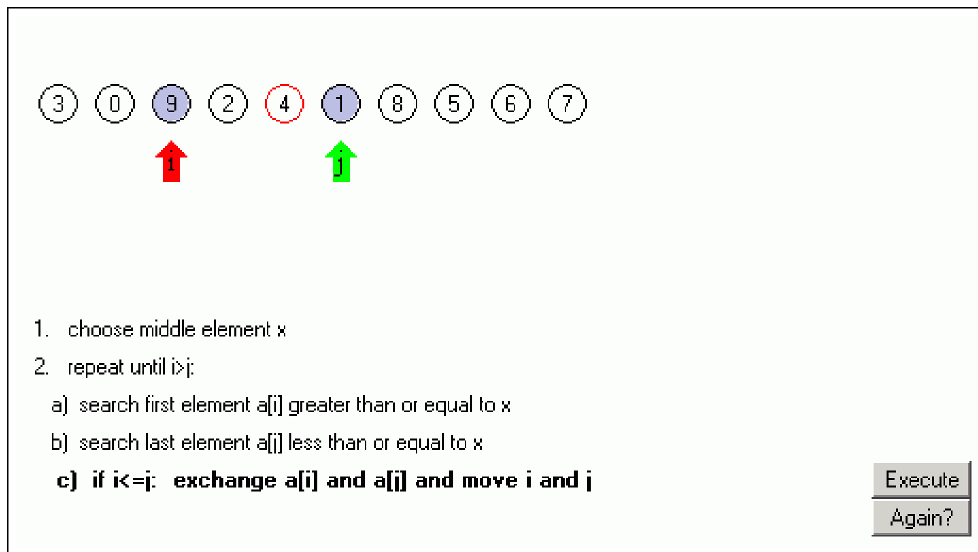

Figure 2.20: Partitioning in Quicksort as Illustrated in Lang's applet [109]

Paul Chew's Voronoi and Delaunay applet [37] determines both Voronoi diagrams and Delaunay triangulations based on user clicks. It also shows the empty circles and the Delaunay or Voronoi edges. Geoff Leach's Delaunay triangulation applet [112] illustrates the performance of three different algorithms with different complexities. Some of the display properties are adjustable by the user. User controls include start, stop, new problem and a toggle between automatic and "manual" stepping equivalent to single step modes. Figure 2.21 shows a screen shot of this applet.

Sandeep Mitra's animations [121] cover various sorting algorithms including Bubble Sort and Quicksort. The animations show the array elements both as sticks and by their numerical values. The user can control the animation display by adjusting the delay between steps and a set of controls also including going backwards in the animation. Some variant algorithms also let the user provide customized input data. One of the applets [122] allows the user to select up to four different sorting algorithms which are then executed in a "race" against each other.

The merge sort demonstration by David Neto [135] illustrates the algorithm by showing the subtrees used in the merging process. The data may be in ascending, descending, "strange" (customized) order or be provided by the user. A "shuffle" button can also be used to modify the element ordering. The user controls include go, step, rewind, stop and a speed control. The animation also shows the decisions made at each tree branching. Figure 2.22 shows a sample screen shot.

The sorting algorithms provided by Biliana Kaneva and Dominique Thiébaut [94] cover Insertion Sort, Quicksort with four variants, Shellsort, Selection Sort, Heap Sort and Bubble Sort. The input data can be in random, ascending or descending order. The delay between animation steps is adjustable, and the user can control the animation with a start, stop, resume and suspend button. The source code can optionally be shown with highlighting of the currently executed command.

The string-searching animation by Adriana Cássia Rosse de Almeida [43] allows the user to set both search pattern and text to search. Furthermore, more than a single animation can be executed, including the Brute Force, KMP and Boyer-Moore algorithm. The user can adjust the delay between steps, but has no other control over the animation.

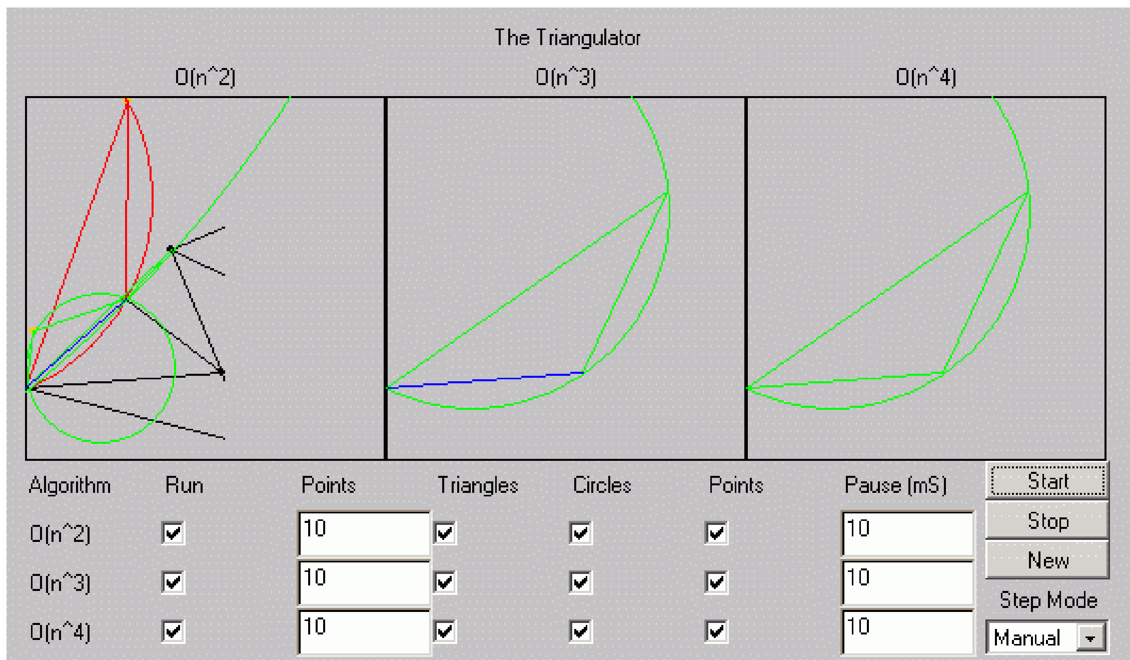

Figure 2.21: Delaunay Triangulation Applet taken from [112]

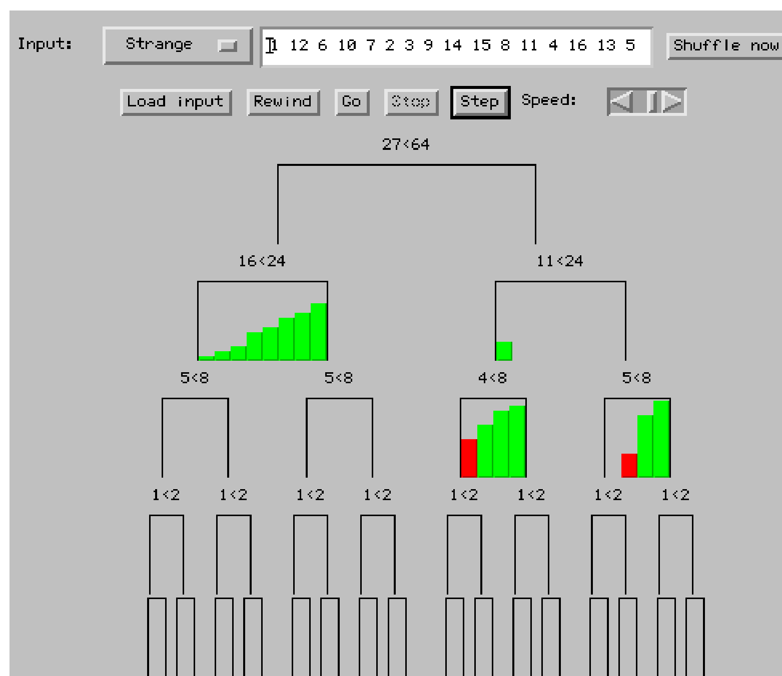

Figure 2.22: Screen Shot of Neto's Merge Sort Applet [135]

A limited number of applets incorporate the prediction of algorithm steps, for example [201], or even simulating an algorithm, for example [54, 56]. Typically, these applets are generated by full-fledged AV systems, or following generation guidelines as in [55].

"Animations" incorporating user activity

Several of the animation applets provided by Tom Naps [132] may use preexisting data used in another algorithm. For example, the data vector generated for one sorting algorithm may be reused for running a different sorting algorithm. The Boyer-Moore string searching algorithm also queries the user for both the searching pattern and the text to search. Most other animations do not allow the user to specify input parameters. However, they offer a "quiz mode" which queries the user about the current state of the display and the anticipated effect of the next transformation. The user can typically control the zoom factor of the display and use a single step mode. Some animations also contain a link to external HTML documentation of the algorithms.
The Heapsort animation by Nils Faltin and Tobias Gross [56] uses a didactical model for animation generation resulting from research by Faltin [55]. It allows the user to perform the execution of Heapsort interactively by picking a pair of nodes and selecting the operation to perform from a list of primitives provided.
The Towers of Hanoi animations by Friedrich Roschmann [166] and Romuald J. Żyłła [213] are both implemented in JavaScript. Instead of performing an animation, they merely visualize the current state and wait for the user to select the appropriate disk to move. The user can also adjust the number of discs.
Mathew Palakal offers visualizations of the data structures stack [143] and queue [142] as a part of the *Personally Active Computing Exploration Resource (PACER)* developed at Purdue University. The user can insert random elements to the data structure and remove them. The standard pointers into the data structure - *top of stack* and *front / rear* for the queue - are also shown. The site contains many other similar applets. More details on the *PACER* system are given in [144].

2.5.2 Algorithm Animation Systems

A large number of full-fledged animation systems has been developed over the last years. As we cannot cover all of them in this section, we will again focus on representative systems and briefly outline the properties of other related systems. We follow the separation of systems by animation generation approach as described in section 2.4: topic-specific systems, GUI-based generation, generation by API method invocations or scripting, declarative visualization and code interpretation.

Topic-Specific Approaches

All animation systems presented in this category are limited to a specific topic area. Both users and visualizers should be aware of this restriction, as it may imply that they have to use more than one system within their course. The presentation proceeds from the more limited to the more general systems.
Probably one of the first widely known algorithm animations comes from Ronald Baecker and Dave Sherman [14]: the movie *Sorting Out Sorting*. This movie was shown in many universities worldwide as an introduction to the covered sorting algorithms. It also contains a complexity analysis and a "race" in which multiple algorithms are executed at the same time. Another pair of older movies

by Ken Knowlton [104, 105] illustrated operations on data structures in Bell Lab's *L* programming language. These movies are regarded as being probably the first to address the effects of operations on data structures by dynamically updating the stored elements [13].
The choice of movies as the animation media implies that the user has a large degree of control concerning the display speed. Most standard video playing device are able to play the movie at normal speed, as well as faster or slower, and may offer "freeze frames" and single frame stepping. Many video playing devices support playing the movie in reverse. Rewinding the movie is also easy, although navigation to a certain scene within the movie is difficult to achieve.
However, presenting the animation as a movie also means that the user cannot adjust any other settings. Thus, the color of the movie cannot be adjusted, nor may the user select a target algorithm or provide a customized input set. In this aspect, the movie acts similarly to many of the animation applets available on the World World Web. Both common sense and research studies agree that the degree of user interaction should be higher than simply using a remote control to achieve learning benefits. More on this will be said in chapter 3.
Sami Khuri and Hsiu-Chin Hsu have developed a set of Java applications for animating CPU scheduling and page replacement algorithms [100] as well as image compression [101]. Each of these packages contains a graphical front-end that displays one algorithm at a time. Both packages allow the user to select the algorithm from a list of supported algorithms.
The scheduling and page replacement package supports the FIFO, Second Chance, Clock and LRU algorithms, as defined in standard operating systems references such as [204]. The user can provide input data by either typing in a sequence of page requests or clicking on a set of buttons representing one page request each. The animation places the previous state next to the current state. A *trace* button shows the effect of the next operation, while the *run* button executes all operations. The total number of page faults is displayed during and at the end of the animation.
The image compression package offers similar controls and illustrates the RLE, Quadtree and JPEG compression algorithms. Both systems embed a set of explanations into the animations that help the user to understand what is currently happening. The image compression package also includes a step-by-step illustration of how JPEG compression works, which should prove helpful to the user.
The *EROSI* system by Carlisle E. George [63] targets the visualization of recursion. Recursion is probably one of the most important basic concepts underlying many applications in Computer Science. However, it is also notoriously difficult to understand for novices [64, 92]. This is at least partially due to the danger of loosing track of context within multiple recursive calls. The *EROSI* system visualizes only selected recursive methods. During method execution, it opens small subwindows showing the new context for each new method invocation.
The *Bewegte Mathematik* ("Animated Mathematics") project by Stauff [200] is based on Microsoft Visual Basic 5 and therefore restricted to Microsoft Windows™ and MacOS™. It contains a large selection of animations for illustrating diverse aspects of mathematics, especially the algorithmic components such as determining the area of an object. Alas, the system's usability is limited due to the platform restrictions, the lack of an applet version and the fact that the whole tool is held in German.
Ted Hung and Susan H. Rodger introduce two tools usable within an automata theory course in [86]. *JFLAP* is a package containing several tools for helping students understand topics of formal languages and automata theory. More specifically, it covers finite state automata, pushdown automata, single or double band Turing machine as well as grammar and regular expression conversion. On starting the tool, the user can select one of the subtools, and may also use more than one tool at the same time. Each tool has an extensive documentation and facilities for both generating

and simulating the generated automata. [71] offers additional comments on using the program. The other system presented in [86] is called Pâté and can be used for parsing and transforming grammars. Pâté shows the textual or graphical visualization of a derivation for a given grammar (restricted or unrestricted). With the textual visualization, a step-by-step derivation is displayed including the rules used at each step. In the graphical visualization, a parse tree for the derivation is shown with each node representing a symbol or a variable. Starting with a user-defined context-free grammar, Pâté performs the standard operations to transform the grammar into Chomsky normal form (CNF) [79]. A graphical representation shall aid the user in determining new productions in the removal of unit and useless productions. Figure 2.23 shows a screen shot of Pâté containing both textual and graphical visualizations.

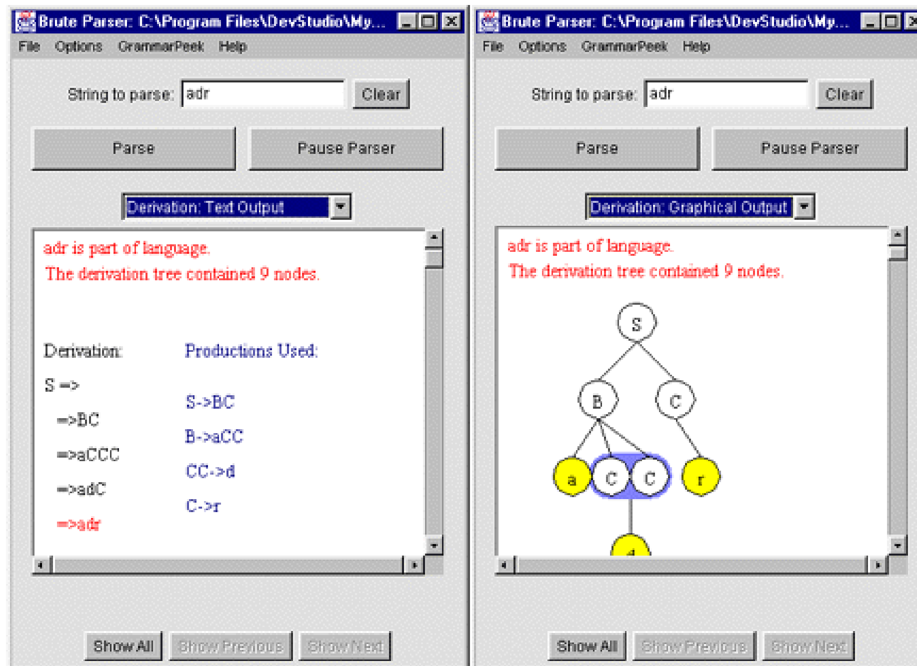

Figure 2.23: Pâté Screen Shot taken from [162]

JELLRAP [95] is another system for animation in the field of formal languages and compiler construction. Its main focus lies in the parsing of strings with a variety of algorithms. It covers the LL(1), LL(2) and LR(1) algorithms. *JELLRAP*, *Pâté* and *JFLAP* are all available on the WWW on Susan Rodger's tool page [161].
In a similar vein, *Ganimal* by Stephan Diehl, Andreas Kerren and Reinhard Wilhelm [47] focuses on interactive, web-based learning software for compiler design. The project currently offers an applet illustrating the compilation of programs, as well as illustrations of Heapsort.
William Yurcik and Larry Brumbaugh present a web-based little man computer simulator [210] used for animating the inner workings of a simple computer architecture. The user can control the simulator with "step into" and "step over" buttons. After setting breakpoints, the simulator can also be put into a "burst" mode which continues until the next `HALT` statement is reached.
PILOT [24] is an interactive tool for learning and grading developed by Stina Bridgeman *et al.* The tool first generates a random problem for a limited set of application algorithms. The example

discussed in [24] uses minimum spanning tree problems. The user then has to select the operations that he or she supposes represent the algorithm's execution. In the example, the user has to decide on the next edge or node selected by the algorithm. The grading component of the systems displays an evaluation of the user's performance once the "Check" button has been pressed. *PILOT* also automatically determines a grade for the user according to a built-in point table.
Duane Jarc *et al.* have implemented a system that incorporates interactive prediction [90]. Similar in concept to *PILOT*, the user has to select the next element touched by the algorithm. The system also includes an animation unit that does not require any user input. However, a preliminary evaluation of the system indicates that the benefits of interactive prediction are not as good as hoped for. Several students seem to rely more on simply clicking elements and waiting for the system's reply. As no grading component is installed and the process may be repeated as often as wanted, the easiest way to "solve" any assignment is using trial-and-error. Whenever the system claims that the currently selected node is incorrect, the student can simply select another node. In this type of usage, any learning effects would probably be accidentally. However, it is often very difficult if not impossible to prevent users from abusing the system in this way.
There are many more topic-specific algorithm animation systems than can be listed here. We therefore constrain ourselves to a single sentence for pointing the reader to a few other systems of interest. Dershem and Vanderhyde [46] present a visualization of all methods of a given Java class that can be invoked by entering input values into text fields and clicking on a button; it is thus not a true algorithm animation system. De Pauw *et al.* [149] present a visualization system for object-oriented programs showing critical operations and statistics from a program execution. Steven Robbins describes several applets for interactive network protocol simulations in [159]. Heath *et al.* [76] describe visualization approaches for performance evaluation and optimization of concurrent programs. Achim Janser has implemented an interactive environment for teaching and learning computer graphics and image processing [89]; however, the system is only available in German and restricted to MacOS™ and Microsoft Windows™ .

GUI-based Systems

Compared to all other generation approaches, the number of GUI-based systems usable for algorithm animation is small. The key word here is *usable* - most of the systems actually used for animations were not specifically designed for this purpose.
Among the most often encountered tools for algorithm animation purposes are the presentation tools belonging to office packages. Typical examples include Microsoft PowerPoint™ and StarOffice Impress™. The main reasons for using them for creating and using algorithm animation are threefold: they may already be used for generating and presenting course slides, they usually have a broad installation base, and they are usable by both instructors and students.
The fact that the visualizer may already be familiar with the tool is highly significant in this choice. On the one hand, the time required for being able to use the tool is reduced or even eliminated. On the other hand, the structure of the tools and their *WYSIWYG* approach of generation reduces the level of abstraction, seemingly making it easier to "get results".
However, the last argument may be highly misleading. While the visualizer is easily able to see the progress the animation makes, manual generation is also very time-consuming! Hundhausen *et al.* [84] and our own experiences agree that manually generating an animation may well take several hours using either standard artistic supplies such as pen, paper and scissors or a generic GUI-based tool. Part of the problem stems from the fact that the whole animation has to be assembled

manually. Even careful planning of the animation beforehand and heavy use of the clipboard facility embedded in office presentation tools may not significantly speed up the process.
Using standard presentation tools for algorithm animations has several advantages for the visualizer. The main advantage, as outlined above, is the visualizer's possible experience with the tool. Most importantly, though, visual generation with the mouse is something that most of today's computer users have become used to. Therefore, the learning curve that visualizers have to master is particularly gentle. This becomes most evident when switching from a GUI-based tool to a tool using a different generation approach. Finally, presentation tools are also relatively easy to use by the user. The presentation is usually regarded as a sequence of discrete slides. Therefore, the user can usually navigate the animation in both directions, and may also jump directly to an interesting state.
Presentation tools also come with a set of typical disadvantages concerning algorithm animation. First of all, they typically belong to an office system such as Microsoft Office™ or Sun StarOffice™. This means that on the one hand, the cost for getting the program can be quite high. For the visualizer, this may not be an issue due to a possibly already existing installation. However, each user may also require an installation of the software. This is especially true if users are expected to take on the visualizer role in the course of their work. Office packages have become very extensive over the last years, regarding both the number of features offered and the amount of disk space required for an installation. Even worse, the office system may not be available for the target user or visualizer platform. For example, there is no port of Microsoft Office™ to other operating systems than MacOS™ and Microsoft Windows™ .
However, the main disadvantage of presentation tools lies in their fixed state. Each tool has its own set of offered features that the visualizer will usually be unable to extend or adapt to specific needs. Despite the ability to use diverse effects for inserting objects into an animation, the effects usable for changing elements already presented may be severely limited. The tool will usually cater mostly to presentation authors by offering fancy background motives or text fonts. The standard graphical primitives including lines are also supported. However, the tools are not designed for generating elaborate drawings.
To further compound the problem, presentation tools are unlikely to offer support for special primitive types such as arrays or linked lists. A typical animation topic such as sorting an array of integer values thus requires the visualizer to draw the array including all array cells, element values and possible pointers to cells. From the tool's point of view, this merely represents a set of objects that happen to be close to each other, without semantic interpretation. Standard features such as grouping all array elements may be helpful. However, this also forces the visualizer to undo the grouping whenever an element is moved or otherwise changed. Typically, presentation tools will start each new "slide" with a clean state, so that all relevant elements have to be copied from the previous slide.
An animation generated in a GUI-based tool does not have any connection to the algorithm it is supposed to present. On the one hand, this gives the visualizer a large amount of freedom in portraying the contents. On the other hand, this also means that the animation represents the visualizer's understanding of how the underlying algorithm behaves - which may differ from the actual behavior. Such errors can only be detected by the human viewer.
Finally, perhaps the worst disadvantage of GUI-based systems is that the animations generated cannot easily be reused. As the full animation is coded manually without a direct connection to the algorithm, any change to the algorithm or the input values requires manual changes to the animation. In extreme cases, changing a single input value may require changing all states of the

animation, or even starting from scratch. This makes it very difficult for visualizers to provide a set of examples for the same topic.

Presentation tools typically only support manual generation or importing of presentations in other formats. The presence of a scripting language, such as Lingo for Macromedia Director, allows at least a certain degree of automatic animation generation. However, the language used must also be simple enough to learn for visualizers. Embedding languages such as Microsoft Visual Basic for Applications™ (VBA) into office components goes in this direction, but still requires the visualizer to learn a programming language.

The addition of VBA has prompted some researchers to examine other office components for their usefulness in algorithm animation. For example, Jukka Eskola and Jorma Tarhio [53] have built the experimental *Flopex* system for dynamic visualizations in MicrosoftExcel. Flopex is a flowchart programming environment embedded in Microsoft Excel using a special interpreter written in VBA. The system can interact with the spreadsheet due to the access to the full Excel functionality and all cells and values in VBA.

Flow charts are generated by copy and paste from predefined elements. These elements include flow lines, read, assign, and conditionals. The "program" can be run either continuously or step by step, and highlights the active cell in each step. The current value of all variables used in the program is shown in spreadsheet cells. Contrary to animations generated within presentation tools, the Flopex interpreter actually executes the algorithm represented by the flowchart. While this prevents the visualizer from employing abstractions, it also makes reuse easy, as the user can provide input data in one column of cells. However, the limitations regarding platform dependency and license costs for presentation tools also apply to Flopex due to its dependence on Microsoft Excel. Furthermore, the usage of Flopex is limited by the features offered by the underlying flowchart interpreter.

Jari Lavonen *et al.* have developed the icon oriented programming system *Empirica Control* for Microsoft Windows™ [111]. The system was designed to support teachers in technology education at Finnish comprehensive schools. The system contains both the programming tool and an I/O interface. Example applications addressable by the system include the indication and control of changes in a physical environment. To this end, the interface provides access to eight independent opto-isolated digital output channels and two independent analog inputs. The paper illustrates several example programs using loops, conditionals, logical connectors and trigger events [111].

The *DANCE* system by Stasko *et al.* [195] uses a graphical editor to generate method invocations in the API-based animation tool *TANGO* [197]. The animation depends on run-time information. *DANCE* is supposed to ease the design of the animation and support program development with rapid prototyping. The generated animation commands can be manually fine-tuned and enriched with additional C source code. However, due to the underlying complex API, laypersons may find *DANCE* comparatively hard to pick up. This is especially the case if they are not familiar with the *path-transition paradigm* [197] used in *TANGO*.

Christopher Hundhausen and Sarah Douglas advocate a simplified graphical user interface for generating "low fidelity" animations in [83]. "Low fidelity" here refers to rough and unpolished animations similar to those built using simple art supplies. The system consists of two main components: the prototypical language *SALSA* for creating and presenting visualizations, and the interactive programming environment for the language, called *ALVIS*. Basically, the user is given a simple graphics editor window for sketching and cutting out visualization objects and a storyboard window in which these elements can be placed. The animation itself is specified either by direct manipulation of the elements or by typing in *SALSA* commands. Additionally, the control interface supports reversing the display of the animation as well as a complete rewinding.

Systems with API-based Generation

In the Java visualizer class by Tom Naps [133], the visualizer declares a "visualizer" instance in the code and invokes the object's *add* method to generate a visualization. A special *toString()* method converts a given execution state to the appropriate String representation. The output can be visualized in Naps' *GAIGS* system. Naps claims that the benefits of this approach include heightened student interest and better appreciation of object orientation and polymorphism, as the same visualizer class is used in different classes. Students acting as visualizers have to determine the key points in the algorithm where the state of the data structure is likely to change in a "relevant" way. The visualizers need not concern themselves with knowing the concrete API methods to invoke. Therefore, this approach may prove to be easier to use for novices, and may lead to fast results. A similar recent approach can be found in [97], which currently supports only arrays.

Rasala [157] presents an animation technique for automatic array animation in C++. The visualizer has to replace standard C++ arrays with a special template class and use the methods provided for modifying the array. The added features of the array template include highlighting compared elements and different views including bar, dot or line charts. Similarly, LaFollette *et al.* [108] describe an animation approach that hides the actual animation method invocations from the visualizer by replacing the standard primitive types with special self-animating C types. The resulting output can be examined in several different views including source code, parameters, global variables, heap state, current operation, call stack and local variables. The user can run the animation, pause it or perform a single step. Currently, the system is limited to acyclic graph structures which the authors claim to be "sufficient for Computer Science I/II courses".

Dershem and Brummund [45] describe the architecture underlying the animation applet shown in Figure 2.17 on page 25. To generate the animation display, the user has to embed specific commands to the algorithm implementation. Similarly, the *Viz* system by Domingue [48] records "history" events for playback. These events are embedded as method calls into the source code. Once the modified algorithm is executed, the history is recorded and used to generate one or more views. This approach is also used in the *Internet Software Visualization Lab (ISVL)* system [49] used for distance education, which also allows the user to annotate the recorded session.

One of the first animation systems with API-based generation is Marc Brown's *Zeus* [25], the successor of the famous *BALSA* systems. Zeus allows the user to select different views, each of which is presented in its own window. For example, a transcript view shows events as a symbolic expression as the event is generated. Zeus incorporates the standard controls *play, pause, step* and a speed control, and also allows the user to store state snapshots.

Zeus follows the *interesting events* approach introduced by Brown and Sedgewick which is summarized in [31]. Basically, this refers to annotating each part of the underlying algorithm with a method invocation that generates an "appropriate" visualization of the current state during the execution. Several other tools use this approach, for example the system by Merlini *et al.* [118]. Here, the API also supports array and binary tree structures and smooth transitions. It is also one of the relatively few systems that offer a "back" facility for stepping to the previous step.

John Stasko developed the *path-transition paradigm* in his dissertation. Summarized in [197], this paradigm performs each object transition along a path, which naturally leads to smooth and flexible transitions by portraying the state of a transition at various intervals of the path. The first systems incorporating this paradigm were Stasko's *TANGO* and *POLKA* systems [197]. The *Vivaldi* project [140] follows a similar approach, but restricts each transition to changing exactly one attribute of one object. Thus, a diagonal move effect requires two transitions to modify the x and the y

coordinate separately. The expressiveness is not limited by this approach, as multiple transitions can be combined in a step.
The extensive *Catai* system [36] requires the algorithm to be implemented according to certain specifics, for example in C++ and using the LEDA library. It offers an efficient distributed interactive animation that can be integrated into the Web. To use the system, the visualizer has to extend the non-animated data structures of the underlying algorithm to animated data structures. The modified algorithm using the new data structures must then be placed in a special class. Some special methods for handling the graphical objects have to be defined by the visualizer.
The first of the three algorithm animation tiers in McWhirter's *AlgorithmExplorer* system [116] relies on a specification using an abstract data type. This abstraction maps to a set of API-based methods that realize the conceptual API. The *Java Data Structures Library (JDSL)* by Goodrich *et al.* also incorporates special facilities for visualizing data structures in Java, as introduced in [15]. The only requirement is that visualizers replace their data structures by those provided in the JDSL.

Systems with Scripting-based Generation

The term "scripting" is sometimes confusing to novices in algorithm animation. Scripting usually does not refer to full-fledged programming languages such as the scripting languages *PHP* [16] or *Perl* [208], but rather to a command-based notation for specifying visualization objects. This type of "scripting" languages often does not support variables, methods or higher constructs such as loops or conditional statements.
One of the most popular and widely used animation systems incorporating a scripting language is *JSamba* [196], a Java implementation of the visualization engine *Samba* used in Stasko's *POLKA* system [197]. JSamba animations contain exactly one command or comment per line. The commands mostly use the standard names for defining or transforming objects, such as "triangle" or "move". Each object must be given an unique identifier so that it can later be referenced for transitions. JSamba allows the user to zoom in and out of the display. To support this, object coordinates are specified by floating-point values in the interval [0, 1] relative to the screen width and height. JSamba supports multiple views and smooth transitions. However, the visualizer cannot specify the duration or offset.
Rodger and Pierson's *JAWAA* [161, 152] is very similar to JSamba. There are some minor differences regarding the names and parameters of some operations. JAWAA is specifically geared to support the common data structures array, stack, queue, graph and tree, as well as the standard graphic primitives. In contrast to JSamba, the number of nodes of polygon lines is not limited - JSamba restricts the number of nodes to 8. JAWAA does not support multiple views or zooming.
The JHAVÉ system by Tom Naps *et al.* [131] is a client-server environment for algorithm animation. It uses either GAIGS or JSamba as its visualization agent. However, the JSamba scripting language has been extended to incorporate multiple-choice quizzes and links to external documentation. The most recent version of JHAVÉ [181] also incorporates the ANIMAL system developed in the course of this thesis and presented in chapter 5.

Declarative Systems

As described in section 2.4.5, declarative systems employ predicates that describe the state of the system. These predicates are usually placed in special comments inside the original source code, so that the code effectively remains the same if a standard compiler is used. A special compiler or

interpreter built into the declarative system is used to extract the actual predicates from the code and uses them to map the current state into appropriate actions.
Declarative systems are still rather uncommon, although there are also some newer developments. Probably the most visible declarative systems is Gruia-Catalin Roman's *PAVANE* system [163]. Animations are generated by predicates embedded into program code. Changes in the program state that touch a defined predicate cause an update of the graphical display. Pavane has been applied to a wide range of visualization problems including concurrent algorithms, rapid prototyping and computer architecture simulation. The authors claim that "competent" programmers are capable of annotating finished programs within three to six hours of effort.
Another, more recent system is *Leonardo* by Pierluigi Crescenzi *et al.* [44, 39]. Leonardo incorporates the special logic programming language *ALPHA* which maps the concrete and abstract data structures to the visualization. The integrated environment includes a special text editor for editing C and ALPHA code. The source code is translated by both a C and ALPHA compiler and linked to generate the actual code. The virtual execution environment contains windows for text input and output, source code, input data and a control bar for controlling the behavior of the special virtual machine. The system may use more than one view for the data. The virtual machine supports fully reversible execution. ALPHA also contains special predicates for toggling the visualization on or off. Specific operation subtypes can be passed as an integer, for example for synthesizing the display of a rectangle [44]. Leonardo is currently only available for the MacOS™ platform.

Systems with Code Interpretation

Code interpretation-based systems can in principle be divided in two different subgroups: systems that rely on external data, such as generated by a debugger, and systems that work directly on the code, possibly including some modifications. The most prominent representative of the first class is the *GNU Data Display Debugger (DDD)* by Andreas Zeller *et al.* [212]. DDD is a graphical front-end for debuggers such as *DBX, XDB* or *Ladebug*. Within the scope of this thesis, its most interesting feature is the graphical display of data structures within programs. Each element in a data structure is represented as a graph node; pointers between elements are represented by edges. The optional alias detection recognizes links to the same object and thus also allows the detection of circular pointer structures.
For visualization purposes, the user has to insert breakpoints into the code. Whenever a breakpoint is encountered, the display is updated according to the current values of the watched variables. A dynamic animation can be generated by associating breakpoints with a *continue* statement. Apart from a graph display, the full functionality of GNUPLOT can be used to plot scalar, 1D or 2D arrays. The user can export the plot data and save the state of a session. To harness the full power of DDD, the user has to be familiar with the basic operations of a debugger. This may cause problems especially for programming novices.
The *Kami* system by Minoru Terada [205] also relies on debugger data automatically extracted using the GDB debugger and the Perl *Expect* package. It offers a *play, stop, rewind* and *until* mode and also allows the user to step over user-defined "trivial" methods. The animation is automatic and passive without user interaction beyond the controls listed. Kami always highlights the currently executed line of code. Its most prominent feature is the use of the *paper-slide metaphor* for illustrating method invocations. Based on a traditional form of children's entertainment in Japan, a new slide is placed over the current activation frame when a method is invoked. Parts of the invoking slide still remain visible and highlight the context of the method invocation.

A similar approach to illustrating method invocations is taken in the *VisMod* project by Ricardo Jiménez-Peris *et al.* [93]. In contrast to the previous two systems, *VisMod* is an integrated programming environment with code interpretation. *VisMod* pops up a window showing the return result of an invocation of a recursive method. However, as Jiménez-Peris and Patiño-Martínez point out in [92], this is insufficient when a recursive program generates multiple calls. Typical examples of this category are Fibonacci and Ackermann numbers and most divide-and-conquer algorithms including Quicksort. For this purpose, they propose to visualize the call tree annotated with the result(s) of the calculation.

VisMod is one of several systems based on a functional programming environment. In [148], the same authors propose to use visualization in text mode, including a visualization for list and graph structures. The presentation includes the expression before and after evaluation, as well as the expression or equation to be evaluated. Color unification of the variables and constructors in the expressions is used for user guidance. Newly introduced right-hand side elements are shown in a common color. The user can set breakpoints either on a function or equation level.

The *ZStep 95* project by Henry Lieberman *et al.* [114] offers comfortable controls for adjusting the display of the animated code interpretation, including step over and back up operations. The execution speed can also be adjusted. Stuart Watt has developed a similar system for code interpretation [209]. Both systems are targeted for the *LISP* programming language. Neither system focuses on algorithm animation; however, the update of the graphical state display can be regarded as a (static) algorithm visualization.

WinHIPE [207, 128, 129] provides a software visualization system for the functional programming language Hope. The current evaluation state is presented as a series of static images shown after a delay, and can also be stored as bitmap files. Apart from the standard play and pause operations, the player for the static snapshots also allows the user to rewind the animation, go backwards or adjust the display speed. The images are based on rewriting rules. A new development supports the dynamic generation of HTML pages from a style sheet defining the page name, animation number and image size.

There are several similar systems for imperative programming languages, especially for *Pascal* with its long tradition in university teaching. For example, Christopher Boroni *et al.* [20] describe a program animator that uses a special execution virtual machine. After choosing a source file, the program can be displayed in a single step or continuous mode. The display can also be reversed. The environment also generates a “run time cost” on predefined operation weights. In a recent publication, they describe how the extended system was integrated into a hypertextbook [21]. For more information, the reader can also look up the URL listed in [68].

The support for animating the behavior of object-oriented programs is also growing. For example, Dean Jerding *et al.* [91] present the *GROOVE* systems that shall help programmers in designing object-oriented protocols in a graphical, animated manner. The well-known *Jeliot* [72] system by Haajanen *et al.* directly animates Java source code. Some minor modifications to the program code may be necessary. The same is true for Steven Zeil’s *AlgAE* system [211].

Jeliot 2000 [113] is not a newer version of *Jeliot*, but rather a modified re-implementation for a different target group, namely 10th grade high-school students. Accordingly, it places its main emphasis on the basic programming elements: input/output, assignments, conditional and loop structures. It is also limited to primitive types and arrays thereof. Animating the input and output process requires the user to replace the standard Java methods with special classes for input and output handling included in the package.

There are also several systems that perform code interpretation for specific tasks. For example,

Javiva [206] focuses on the visualization and validation of student-written programs. To this end, it extracts pre- and postconditions as well as abstraction functions from special comments embedded in Java source code. These conditions are used to generate an instrumented class file with "appropriate" validation method invocations. Whenever a condition is violated, the class informs the user with a pop-up window. Optionally, the user can embed method invocations to the JDSL [15] described in the section on API-based generation.
The *GRASP* programming environment *(Graphical Representation of Algorithms, Structures and Processes)* [88] is a special software engineering tool. Besides syntax highlighting and code folding, it also incorporates the display of control structure diagrams (CSDs) showing the context of each block inside the code. It can be used for a variety of languages including Ada95, C++, Java and the hardware definition language VHDL.
Finally, the *PVaniM 2.0* system [199] provides both online and postmortem visualization support as well as rudimentary I/O for long running, communication-intensive PVM applications. It provides these features while using several techniques to keep system perturbation to a minimum. The online graphical views provided by PVaniM 2.0 provide insight into message communication patterns, the amount of messages and bytes sent, host utilization, memory utilization, and host load information.

2.6 Summary

Algorithm animation as a special type of software visualization illustrates the dynamic behavior of algorithms. Researchers and educators have been interested in visualizing algorithmic operations for a long time. Starting with paper-based notations such as flowcharts, the field has evolved into instructional movies and computer-based animations. Today, a large selection of algorithm animation systems is available. People interested in creating or using algorithm animation may have to spend much time in finding a system that fits both their skills and expectations. To address this problem, we have explored the different usage roles and animation generation approaches in this chapter. We have also discussed a large set of animation applets and full-fledged systems.
Four roles can be associated with algorithm animation systems [156]. The *programmer* is responsible for implementing the algorithm. He or she need not be aware of any interest in visualizing the algorithm. The *developer* designs, implements or refines an algorithm animation system. The *visualizer* combines the output from the two previous roles by mapping the algorithm to a visualization in the chosen system. Finally, the *user* interacts with the animation generated by the visualizer. The degree of interaction depends on both the animation and the underlying system.
Focusing on the *visualizer* role usually adopted by an educator, six different animation generation approaches can be isolated. Apart from (semi-)automatic *topic-specific* systems, animations may be generated using a *graphical user interface*, by *API method invocations*, using *scripting commands*, by *declarative* programming using predicates, or by *code interpretation*. The main characteristics of these approaches are summarized in Table 2.1: the *input type* needed for generating an animation, the typical tasks of the *developer* and *visualizer* role, and the main benefits and drawbacks of the approach. The *programmer* is by definition independent of the algorithm animation. The interactions possible for the *user* depend on the animation display front-end of the system, not on the approach used for animation generation. Therefore, the *programmer* and *user* role are not included in the table.
None of the generation approaches is inherently superior to the others. Rather, the circumstances of the planned usage help in determining the most appropriate approach. For example, if algorithm

Generation Type	Input type	Developer's Task	Visualizer's Task	Benefits	Drawbacks
Topic-specific	algorithm parameters, other input values	Implement or embed code, and map code to visualization.	—	Highly specific; allows special enhancements. Predictions fairly easy to embed.	Restricted to fixed context, usually not reusable - different tools needed for different topic areas.
GUI-based	mouse and keyboard	Provide a graphical front-end for editing; support "appropriate" primitives and effects.	Manually generate animations by drag & drop with mouse and keyboard.	Easy to learn; may focus on or skip aspects; may embed explanations and choose abstraction level; may present effects of selected "bugs".	(Very) time-intensive generation; usually no automation support; animation only valid for given values; may contain (unintended) errors.
API invocations	method calls	Implement and document API; ensure appropriate mapping to visualization operations.	Embed API calls into the original code or use special objects with embedded API invocations.	Each invocation generates a new animation; may use arbitrary parameter values; code can be given to students; supports a "clean" embedding.	Language(s) fixed by API; programming skill required; limited by provided API methods; modifying underlying code may alter behavior and reduce legibility.
Scripting commands	scripting code	Implement parser; document language; map commands to appropriate visualization operations.	Write code manually or embed generation in algorithm, explicitly or implicitly using special objects.	Embedded commands generate animation in each invocation; reusable if stored to file; allows manual fine-tuning; algorithm implementable in any language.	Some programming skills required; limited by commands provided in scripting language; code embedding may affect behavior and reduce legibility.
Declarative	predicates	Implement analyzer for predicate extraction; map predicates to graphical representation.	Insert the predicates into the original code or specify separately.	Each invocation generates an animation; use arbitrary parameter values; clean embedding; may allow user-defined predicates.	Requires familiarity with mathematical reasoning (predicates) and a good documentation of predicates.
Code Interpretation	source code or executable file	Provide code analyzer, debugger or interpreter.	May have to define breakpoints or replace some method invocations, esp. for standard I/O.	Tight connection between code and animation; immediate effect of modified values.	No explanations or pseudo code; low abstraction level; usually little granularity control; not useful for all algorithms.

Table 2.1: Typical Properties of Generation Approaches

animations are limited to a single application area, a topic-specific systems may be the most appropriate choice. However, if several topics are covered, choosing a set of topic-specific systems forces users to familiarize themselves with different systems over the length of a course. This may lead to confusion and diminished motivation. The key findings regarding the generation approach can be summarized as follows:

- GUI-based animation systems are probably the easiest to learn for algorithm animation laypersons who want to generate their own animation. This is mostly due to the fact that these systems provide immediate feedback to any action taken. Also, most computer users have gained sufficient familiarity with GUI-based systems from the window managers running on their system, such as *GNOME* or *KDE* for Linux, or the graphical interface of the various Microsoft Windows™ releases.

  However, the generation is very time-consuming and usually can portray only a specific set of input values. Changing parameter values can only be addressed by generating a new animation.

- API-based generation may be easy to pick up for more experienced programmers used to invoking methods in external libraries. The calls are directly embedded in the code, either by explicit invocation or by using self-animating data structures. Each new invocation of the code generates a new animation. Thus, it is easy to portray the effect of changed parameter values on the chosen algorithm.

  While the embedding of additional method invocations is relatively "clean" from a programming point of view, it may affect the algorithm behavior due to side effects. Run-time measurements also become ineffective in instrumented code. Furthermore, the method invocations may reduce the code legibility if the division between algorithm and animation code becomes diffuse.

- Scripting-based systems are conceptually close to API-based generation. Instead of method invocations in a special library, the visualizer generates scripting commands, typically by sending them to the standard output or to a file. The approach mainly shares the same benefits and drawbacks as API-based generation. However, scripting generation is not as "clean" as method invocations due to infusing the code with output statements. On the other hand, the generation of the scripting code is independent of the programming language in which the algorithm is implemented. Storing the generated code in a file allows the animation to be reused.

- Declarative systems usually embed the predicates in special comments [163]. Thus, they usually affect the code legibility less than scripting-based systems. In principle, each predicate could be mapped to a method invocation in an equivalent API. Thus, the approach also shares the same benefits as API-based generation. Some systems also allow the visualizer to define new predicates. However, the visualizer must have a certain skill in mathematical reasoning to use these systems.

- Code interpretation frees the visualizer from most tasks, except perhaps the setting of "appropriate" breakpoints. These breakpoints can be similar to Marc Brown's concept of *interesting events* [26]. Some code modifications may be necessary if the source code is interpreted in a fixed environment. The main benefit is the automatic tight connection to the actual algorithm

execution; on the other hand, this closeness may also make abstract reasoning or focusing on the "main" parts of the algorithm more difficult. For example, skipping ahead in the code still requires the execution of the actual code. This may take a long time, as in the case of NP-hard problems [62].

Section 2.5 presented several systems for each generation type. The available systems can be split into two categories: *applets* that illustrate a selected topic and full-fledged *animation systems*. The applets presented in section 2.5.1 were grouped according to the amount of user interaction and additional information presented. Simple non-interactive applets may not show the underlying source code or any textual description. More advanced applets let the user specify the input data and control the animation, for example by adjusting the display speed or using a single step mode. Some applets also contain additional information that explains the current state of the display. Section 2.5.2 discussed a selection of full-fledged systems for each animation generation approach. Several of the systems let the user specify at least some input data. They may also allow the generation of new animations by the visualizer.

The large number of available tools makes the choice of an appropriate algorithm animation system difficult for both users and visualizers. Visualizers should try to determine the type of generation that best fits their personal preferences and skills. However, depending on the chosen generation approach, the number of remaining choices may still be large. Therefore, both visualizers and users need a set of requirements that each system can be checked against. The most appropriate system for their use can then be determined as the system that matches most of the requirements. The following chapter focuses on the generation of such a set of requirements.

Chapter 3

Requirements Analysis

3.1 Introduction

The large number of available algorithm animation systems makes finding an appropriate system difficult. Simply picking an arbitrary system may result in problems. For example, it may take a while to figure out that the chosen system does not support all required operations. Evaluations of available animation systems have to be constantly maintained to reflect the current state. Such lists are unlikely to be both up to date and complete. For example, the 21 pages devoted to an evaluation in section 2.5 contain only a small selection of the currently available applets and animation systems.
Therefore, we take a different approach in this chapter. Rather than trying to pinpoint the strengths and weaknesses of selected systems, we explore the requirements for a hypothetical "ideal" algorithm animation system. Some of these requirements conflict with others, and may even contradict each other. One standard requirements reference are the "ten commandments of algorithm animation" by Gloor [67]:

1. Be consistent
2. Be interactive
3. Be clear and concise
4. Be forgiving to the user
5. Adapt to the knowledge level of the user
6. Emphasize the visual component
7. Keep the user interested
8. Incorporate both symbolic and iconic representations
9. Include analysis and comparisons
10. Include execution history

These "commandments" represent a mixture of demands on the different roles in algorithm animation, as defined in section 2.3. We argue that all requirements represent the interest of the *user* role. However, some of the requirements place demands on how the *visualizer* generates the animation, notably requirements 1, 3, 6, 7 and 9. Additionally, the *developer* has to embed appropriate functionality in the system so that some requirements can be met by the visualizer. This mostly concerns requirements 2, 4, 5, 8, 10 and to a lesser extent requirements 6 and 9.
The goal of this chapter is to provide the reader with a comprehensive set of requirements for animation systems. The requirements are split into five categories. The first category contains general requirements applicable to all roles. The following four categories are dedicated to the requirements for one specific role each and are discussed in separate sections. Each requirement is described and motivated. Not all requirements are equally important on an objective scale. Additionally, the reader may regard several requirements as less important according to his or her preferences. Most requirements are placed on functionality offered by the animation system, and thus address the work done by the developer. The chapter ends with a brief summary of the main requirements.

3.2 General Requirements

This section focuses on general requirements which are independent of the current role taken by the person using a given system. Therefore, they apply to most or all of the different roles. Typically, the requirements are implicit for the *developer* role, as the developer can define the system to fit the requirements. They are usually of little or no concern to the *programmer* role. This is especially true if the programmer is unaware of any visualization plans. The requirements can be split into three categories: requirements regarding the machine the system is used on, development state and performance of the animation system, and applicability of the animation system.

3.2.1 System Requirements

GR1. Operating system independence

Following the presentation of the Java language, this argument has gained a large following. The recognizable trend of many authors, including such illustrious firms as IBM, goes toward the development of platform-independent software. The main rationale behind this trend is the ability to reach the widest possible audience without having to port code to different systems.

Several programs are only available for selected operating systems. For example, *WinHIPE* [207, 128, 129] is only available for the Microsoft Windows™ family. *ZStep 95* [114] and *Leonardo* [44, 39] are only available for MacOS™. Thus, users running the increasingly popular Linux or other operating systems are excluded from using either of the tools.

A common statement regarding operating systems is that "most people run Microsoft Windows™". This is used as a justification that it is unnecessary to use portable languages such as Java or provide a port to other platforms. However, the argument does not consider that many universities use UNIX or MacOS™ on their computers. Arguably, university students or school pupils are among the primary clientele for algorithm animations. Thus, preventing a portion of them from using the tool is unwise. The press also periodically indicates that several firms consider shifting to Linux for speed and cost considerations.

GR2. Modest hard disk and RAM requirements

Hard disk capacity is comparatively "cheap" and new "off the shelf" computers are being equipped with an increasing amount of memory. Thus, it is tempting to ignore the space and memory requirements of tools. However, the amount of memory used is of importance to many users in a variety of ways. For example, even large hard disks become filled due to the tendency of software to grow larger with each new release. Additionally, university campuses or firms often restrict the amount of space a given user can allocate. Buying additional disks or RAM modules once the capacity is exhausted may not be possible due to financial or organizational reasons. Private users may also refrain from downloading sizable tools due to the amount of time required.

GR3. No special hardware or additional component requirements

Depending on the features of a given tool, special hardware may be necessary. This includes supposedly "standard" hardware, such as CD-ROM or DVD drives. However, it may also concern video decoder hardware or special graphics adapters, for example for 3D support. The developer should consider that *each* additional requirements may limit the number of users. For example, not all computers today are equipped with a CD-ROM drive. This concerns older computers as well as some terminals found in university computing centers. It is questionable whether the interest of a given user is sufficiently high to invest the money in additional components, let alone buying a new computer.

GR4. Independence of an Internet connection

The obvious drawback of requiring an active Internet connection is that it directly limits the user range to those who can access the Internet. Many universities allow their students to use the Internet. However, this does not mean that the students also possess a private Internet connection at home. Even if this is the case, using the system may result in costs for the user in the form of Internet connection or phone bills.

Finally, and perhaps most importantly, the system may be rendered unusable if the Internet connection fails. In the simplest case, the user has no way to connect to the Internet. For example, he or she may be traveling, or have no modem or Ethernet plug. Problems at the client or the system site, restrictive firewall policies and a variety of connection problems may also prevent system usage.

Some systems require a connection to the Internet due to their nature. For example, client-server systems naturally require a connection to the server component. If the server is not shipped to the user for technical, legal or other reasons, the system is only usable with an active connection. In these cases, the developer has to take into account that users may be prevented from accessing the system due to the reasons given above. If the Internet connection is not an essential part of the system, it should be added as an *option* which leaves (part of) the system functionality available if the connection fails. Otherwise, possible users of the system should be informed that an Internet connection is required. They can then decide whether they want to download the system based on their connection situation.

3.2.2 Development State and Performance

GR5. System maintenance and stability

Many software developers offer clients access to pre-release versions of the software, commonly called "beta software". Users may associate "beta software" with many unknown or known, but not yet fixed, bugs and lacking some of the expected features. Honesty and fairness to the client thus require that *any* piece of software that is not considered "finished" should mention that it is still "beta" or "under development". Additionally, the user should be able to determine whether the authors plan to maintain the system, so that bugs may be fixed in a future release.

Ideally, the system should be available as Open Source. Other developers are then able to develop additions or adaptations, as well as fixing known bugs.

GR6. Update policy

Similar to the previous requirement, a regular update policy can improve user confidence and satisfaction with the software. "Regular" does not necessarily mean "scheduled", so that no fixed dates for updates have to be given. A good update policy will mention that an update is planned, possibly including a list of the bug fixes and additions. This allows users to determine if problems or shortcomings in the tool will be fixed. Without such information, users may consider changing to a different tool if they cannot resolve their problems.

GR7. Total Cost of Ownership

One of the "buzzwords" in software development over the last years, *total cost of ownership* (TCO for short) addresses the sum of all costs associated with using a given system. This includes the cost of acquiring the system, the price of necessary additional components (**GR3**), the cost for the working time spent on developing content, and Internet connection fees (**GR4**). Other cost factors may for example be the cost for acquiring an update, license fees for software generated with the system or using an optional stand-alone "player" software. Finally, the user may have to invest money in additional documentation, demonstrations, tutorial material or simply in printing the digital manual. Comparatively "cheap" systems regarding the original price may have a high cost of ownership once the other costs are taken into account.

If possible, the user should be advised about additional costs. Thus, the system description should include a list of expected components with their associated price.

GR8. Sufficient system performance

The performance of the system depends on the underlying hardware. The term "sufficient" is at least partially subjective. However, the operations of the system, such as displaying the sequence of animation frames, should not fall below a certain threshold. For example, the illusion of smooth motion as in video clips requires a refresh rate of typically between 20 and 32 frames per second. Users should be advised of minimum or typical requirements for standard performance. For example, statement such as "requires an Intel Pentium III with at least 700 MHz" help users in deciding whether an installation is useful. Such information is usually included in commercial software products, but often absent in shareware or freeware products.

GR9. Easy downloading, installation and maintenance

A large amount of administrative tasks are placed on both university faculty and system administrators. Therefore, tool acquisition, installation and maintenance is one of the general hindrances for adopting a tool. Even moderate effort may prevent effective use of the tool [21, 167].

Another consideration is the effect of updating the software. It is often unclear whether the update will modify, replace or simply ignore an existing installation. For example, the updated version may be unable to access "old" files due to changes in formats. If the update replaces the previous version, those files may be lost. Whenever possible, the user should therefore make a backup of the software *before* installing a new version or update.

3.2.3 Applicability

GR10. Flexible application area

As discussed in the previous chapter, software that focuses on a single topic may provide better support for its specific focus. However, this advantage also requires changing to another system for exploring a different topic. Each system is likely to have its own specific user interface. Therefore, both visualizers and users may have to learn using the system without benefitting from their previous experience. The need to locate a set of systems for covering the range of topics covered adds to the complexity of system installation and maintenance (**GR9**) and may decrease the interest in using the systems.

GR11. Well-documented system goals

Khuri [98] points out that the intended goals of a system should be well-documented. In his article, he targets the developer of an algorithm animation system. However, users and visualizers can also benefit from this. A documentation of development goals may outline possible application areas. It can also inform interested parties that the software may not be appropriate for the planned area of use.

GR12. Upward file format compatibility

New releases of software tend to add functionality in addition to fixing bugs. This added functionality usually also impacts the file format. While the new release thus may have to modify its save and load routines, it should still be able to process files written with older versions. If this basic requirement is violated, all animations generated with a previous release may be rendered unusable after installing the update.

Note that the requirement does not prevent the author from adding to the file format or even changing the basic way animations are stored. Rather, it demands that the older parsing routines are still supported. This may also contain the existence of the previous storage routines, allowing the user to store an animation in a format that users of a previous release can process.

GR13. Downward file format compatibility

GR12 requires that users can still process files generated in an *older* release. However, it would also be helpful if users of an *older* release could read animations generated in a *newer* release. Due to format changes and the possible use of features not present in their release,

some compromise in the quality of the animation may be unavoidable. However, the ability to get at least a rough impression of the new animation is helpful and may also be a good acquisition incentive.

3.3 User Requirements

This section examines the requirements that the *user* role places on an "ideal" algorithm animation tool. As described in section 2.3, *user* refers to the end user of an animation tool. Several user requirements thus deal with interface decisions and the didactical structure of the animation. The structure of the animation is decided by the visualizer; however, some of the user expectations may require tool support for specific operations.
The requirements can be split in the following eight different categories, discussed in separate subsections:

- content presentation,
- animation display controls,
- user interaction,
- educational support,
- user interface considerations,
- file exchange with other systems,
- algorithm understanding support
- and miscellaneous requirements.

3.3.1 Content Presentation

UR1. Embedded textual explanation of view

A common proverb in several languages holds that "a picture is worth more than 1,000 words", although the precise number of words differs between languages. Hundhausen [81] presents an evaluation of several cognitive science theories and empirical evidence that indicates that people remember pictures more easily and readily than text. However, stand-alone pictures may not be quite as helpful for *understanding* the content portrayed.

Several algorithm animation studies point out that one of the most important features of an animation tool is the ability to embed a textual explanation [55, 2, 18, 21, 99, 134, 128]. This is also supported by psychological research, as cited in [18]: text and graphics must be presented simultaneously and coordinated well. Anderson and Naps [2, 134] state that the presence of explanatory textual materials seems to "be a minimal requirement to achieve any level of understanding". The underlying rationale is that the text is used to explain the graphical display, especially the mapping between the underlying algorithm and its graphical presentation. Without this additional help, the users may be unable to fully understand the significance of what they see.

UR2. Embedded pseudocode with highlighting

The addition of pseudocode is one possible application of **UR1**. Several studies [82, 51, 32, 201, 207, 38, 55, 174, 21, 110] consider pseudocode embedding important for understanding. Pseudocode offers a comparatively compact way of conveying the structure of an algorithm without focusing on the precise syntax of a programming language. Highlighting the currently executed line of pseudocode guides the attention of the user. It may also improve the mapping between the algorithm and the animation [55, 174].

Pseudocode simplifies the presentation of algorithms with long or intricate implementation. For example, it is comparatively easy to state *Dijkstra's* algorithm in pseudocode by skipping over the details on how the *next cheapest reachable node* is selected. The actual implementation, which may incorporate optimizations or special data structures such as a *priority queue*, may confuse students. However, the focus of the algorithm does not lie in the specific implementation details!

UR3. Metaphors and Applications

In a similar vein as supporting pseudo code with highlighting, abstract views of an algorithm may help the user in getting a better understanding. These abstract views may be a pseudocode version of the algorithm, "stories" [50] or metaphors built around the algorithm. One of the common metaphors is searching for a telephone number in a phone book, usually used to explain *interpolation search*. The actual choice of the abstraction depends on the preferences and social environment of both visualizers and users.

The ability to present a "real world" example of the algorithm in action may increase user interest. It also offers a different view of the algorithm which some users may find easier to grasp than the algorithmic presentation.

UR4. Linking to documentation

Layout considerations restrict the amount of documentation that can be embedded into an animation. Thus, users may profit from a link to documentation that is presented in a separate window. There is usually no "best" way to provide a given material that works well for all users. Empowering the user to choose the reading speed as well as the order of reading the material helps in addressing different needs. This may also include sidetracks unanticipated by the visualizer. Hypertext documentation gives the control of reading and navigating the documentation to the user and is preferable to straight text [65, 2, 21, 99, 134, 167].

Note that the *location* of the documentation is also important. Documentation which only resides on a Web server requires an Internet connection, which may reduce usability for some users - see **GR4** and [167].

Anderson and Naps [2, 134] propose the addition of material in three different stages of interactivity. *Static* materials contain a non-changing hypertext material that is always present, regardless of the stage of execution of the algorithm. *Algorithm-sensitive* material changes with respect to the execution of the algorithm, but is unaware of the actual data values affected by the operations. *Dynamic* material finally is fully aware of the current data, and thus may incorporate this knowledge into the documentation.

3.3.2 Animation Display Controls

UR5. Full-fledged video player controls

A large number of studies stresses the importance of the animation control for the user. The standard controls should at least support *pause* and *resume* [25]. Tools that view an animation as a slide show of connected steps should also support *step* functionalities. Other requirements often encountered include the ability to *fast forward* the animation either freely [67, 114] or to predefined "points of interest" [83, 99].

Some researchers also stress the importance of arbitrarily stepping *backwards* in the animation to fixed "points of interest" [67, 83] or arbitrary points in the animation [118, 39, 55, 2, 21, 113, 134, 128]. By the time users notice that they are confused or have lost track of the animation, the appropriate point for regaining the train of thought is most probably passed. Therefore, the tool should support backtracking. Anderson and Naps [2] consider this essential for effectively using animations "to achieve any non-trivial level of understanding". The article also points out that achieving this kind of backtracking is difficult in many cases. Hardly surprisingly, it is absent in most animation systems with dynamic displays. Naps [134] further points out that achieving the first two levels of Bloom's taxonomy [19] require a rewind facility.

UR6. Embedded breaks between steps

Some animation tools as well as general presentation tools such as Microsoft PowerPoint™ offer a "continuous" display more reminiscent of a slide show. While it may be tempting for visualizers to incorporate this facility into their animations, breaks between steps can be very relevant. The time needed for assimilating the animation content differs between users, as does the speed of comments given by an instructor [174]. One of the common points of critique in animations is that "the animation doesn't wait for you to think" [2]. This may even be true if a *pause* button is provided, as the user may be too slow in pressing the button, leading to the problem of rewinding described in **UR5**.

Forcing the user to press a key or click on a button whenever the next step should be displayed is also inadvisable, as it may easily lead to fatigue. Depending on the algorithm being animated, the animation may easily contain more than one hundred steps, each requiring a separate key press or mouse click.

3.3.3 User Interaction

UR7. User interactivity support

Gloor [67] points out that the user should be active at least every 45 seconds. The simplest way to achieve this is by combining slide show modes with explicit *play* commands issued by the user at his or her own speed. Anderson and Naps [2] propose that a greater degree of interaction allowed by the instructional design of the tools ties in with a greater degree of understanding. This also underscores the importance of embedded breaks between steps (**UR6**) combined with flexible controls (**UR5**) that prompt interactivity, if only on a mostly mechanic level.

UR8. Support role shift from user to visualizer

As a logical consequence of requirement **UR7**, the ultimate level of interactivity is letting the user adopt the visualizer role. Several studies present convincing arguments that letting users design their own animation is most effective for understanding [160, 194, 85, 2]. The central issue then is the degree to which the animation system makes the design of a visualization abstract. Both the level and quantity of programming needed for generating an animation should be minimal to allow the user to focus on understanding the algorithm [2].

Note that this does not mean that the animation of a given algorithm has to be easy to modify. Rather, the animation tool should present some avenues for the user to become a visualizer, independent of any given animation. Thus, the user will generate a new animation, which may be used interactively by other users. Of course, educators acting as visualizers may also provide a basic animation structure that has to be completed by the student users.

UR9. Input data generation or specification

Some animation systems allow the user to specify the input data by entering concrete values [71] or selecting the type of input from a predefined set [32]. Gloor stresses the importance of enabling the user to specify input data by making it part of the second of his “ten commandments of algorithm animation” [67]. As he also points out, this functionality is less important for users new to either the animation tool or the portrayed algorithm. For these users, a predefined set of inputs should be provided that illustrates the salient features of the algorithm. The same argument is advocated by Khuri [99].

The user may find it difficult to fully design a “helpful” input. For example, novices to the *Quicksort* algorithm may find it difficult to figure out the correct type of data set to choose for worst-case or best-case behavior. The input facilities for entering the data may also overwhelm the student due to the nature of the algorithm. For example, entering a complete graph as done in [71] for formal languages already requires a certain level of understanding. Anderson and Naps state that a “strategic” decision may be sufficient [2, 134]. In this case, users are restricted to predefined sets of input types that help them in deciding the critical issues. In sorting algorithms, this selection could be restricted to *random, in order, in reverse order* or *almost in order*.

UR10. Incorporate predictions and “quizzes”

A comparatively new development in algorithm animation is the inclusion of *interactive prediction*. The animation is stopped at specific breakpoints, and the user is prompted to predict what will happen in the next step. This prediction may incorporate clicking on specific areas of the screen. For example, a *spanning tree* algorithm animation may ask the user to select the next node or edge chosen by the algorithm [24, 55, 90].

Alternatively, a text prompt may appear that presents a “quiz” question and prompts the user to provide an answer. These quizzes can cover a variety of types, such as *true / false*, *multiple choice* with one or more correct answers, or *fill in the blanks* [131]. The developer and visualizer may decide whether users get direct feedback regarding the correctness of their selected input.

Interactive prediction shall serve to sustain the concentration of the user. Studies by Byrne *et al.* [34] indicate a benefit of using prediction, although this benefit was not statistically

significant. Part of the unreliability may stem from the difficulty of the topic chosen for the experiment. The structure and quality of both the prediction integration and the animation itself may also have affected the experiment. Finally, previous experience of the users may have impacted the results. In the study, strong users remained strong, while users who made lots of errors in the prediction seemed not to learn enough from it in time for the post-test. A recent study by Hundhausen [85] indicates that prediction can be a very effective way of achieving significant improvement in understanding an algorithm as a problem-solving recipe. An upcoming dissertation by Hill [77] examines the instructional value of student predictions in tree animations.

However, the effectiveness of interactive prediction is still not fully evaluated. There may be significant differences between predictions in pencil and paper form in a highly controlled lab setting, as opposed to the integration of interactions in an animation tool. There are only few systems with fully integrated prediction, including JHAVÉ [131], the work by Jarc *et al.* [90], Bridgeman *et al.* [24], Miraftabi [120] and Faltin [54, 56, 55].

UR11. "Interesting" and "pathological" data incorporation

Animations with a predefined set of input values or supporting strategic input choices as described in **UR9** should also offer two special types of input. "Interesting" data refers to data that lets the user explore specific properties of the algorithm. For example, users learning the *Quicksort* algorithm may find it interesting to see the algorithm's behavior when comparing two swap candidates which are equal to the pivot element. This ties in with the distinction of whether *Quicksort* is stable. As this type of data will often not occur in randomized generation and may also not be representative, some authors also refer to it as "cooked" data [27].

"Pathological" data, on the other hand, takes the algorithm to extreme behavior. Typically, it also illustrates the worst-case behavior of the algorithm, for example when sorting an array of identical values. Brown and Hershberger [27] mention that the use of pathological data helped discover a subtle bug in the implementation of a polynomial decomposition algorithm. Other examples include selecting a single color bitmap for run length encoding [99].

UR12. Labeling of main steps

The third of the "ten commandments of algorithm animation" [67] includes a demand that important logical algorithm steps should be accentuated in the animation. This can be achieved by annotation or by providing a special controller element that lists the steps and also acts as a hyperlink for jumping to a given step. A similar demand is given in [83, 99]. Incorporating this display guides the user by highlighting the current step and may allow for a better in-depth understanding of the algorithm [66].

3.3.4 Educational Support

UR13. Embedded analysis, complexity and comparison

The ninth of the "ten commandments of algorithm animation" demands an inclusion of an analysis of the algorithm's behavior [67]. This allows comparing the performance of different algorithms, possibly also running multiple algorithms in parallel [27]. A similar demand is formulated by Anderson and Naps [2] who place the ability to perform a complexity analysis

at the second-highest rung on their algorithm understanding scale. Note that the ability to follow or even perform a complexity analysis also ties in with the highest level of Bloom's understanding taxonomy [134].

The analysis should ideally incorporate the actual data used in the algorithm, for example by explicitly stating the number of operations performed during the animation [67]. As a smaller requirement, it should include the complexity class of the algorithm in big-O notation.

UR14. Faculty propagation

Perhaps the most important requirement in gaining user interest in a given tool is the propagation by faculty or other teaching staff. Bazik *et al.* [17] state that much of the success of the BALSA system was due to the tight integration with the other resources for the algorithms and data structure course. Similarly, students are more likely to use algorithm animations if the teachers are enthusiastic about the tools themselves and propagate this interest. In some cases, even a recommendation or pointer to web resources by the teacher may be sufficient to prompt users to use the system.

UR15. Use of a visual layout notation

Certain visual layout notations such as grids or flowcharts may be helpful for novice users. For example, Mulholland and Watts [125] found using “fact cards” similar to spreadsheet tables helpful. Part of the reason is ascribed to the lower level of syntactic detail compared to using a “real” programming language such as *Prolog*. This may assist novice users in crossing the boundary from uncertainty into a confident use of the tool [125]. Similarly, Ben-Ari [18] argues that a secondary graphical notation including layout and typography may be advantageous. However, this notation must also be learned by the user and may not be obvious especially for novice users.

UR16. Adjustment of detail level

Both novice and advanced users may want to be able to adjust the amount of detail presented in an animation. Some systems support this feature, for example by allowing the user to select different views of the data. Anderson and Naps [2] postulate the adjustment of detail as the fifth of their eight levels on the instructional design scale. Similarly, adapting to the user's level of knowledge is the fifth of Gloor's “ten commandments of algorithm animation” [67].

The level of detail may be arranged by hiding corresponding elements or offering multiple selectable views [27]. For example, [107] incorporates several different selectable views which range from a basic display to detailed causality or communication views for distributed algorithms. Typical other examples include portraying sorting algorithms as an array, a set of sticks, a dot plot or a hierarchical recursive call tree [2]. The Heapsort learning applet by Faltin and Gross [56] uses a bottom-up approach that illustrates the individual methods separately, and thus avoids swamping the user with details.

3.3.5 User Interface

UR17. General user interface considerations

A vast amount of different issues have to be taken into account when designing a user interface. Discussing the issues is far beyond the scope of this thesis. For example, the user interface should be easy to use, self explanatory, consistent, prompting only for relevant interaction, flexible and supportive regarding the user's tasks. An in-depth discussion of user interface considerations can be found in a variety of *Human-Computer Interaction (HCI)* publications. A good starting point for novices to the topic is Faulkner's book on the "essentials" of the field [57].

UR18. Internationalization in animations and user front-end

An aspect often overlooked by the predominantly English-speaking software developers is the ability to translate content into the user's native language. In the context of algorithm animations, this concerns both the animation content and the graphical user interface. For example, many German students may be less willing to use a system which is held completely in English. While we may regard this as regrettable, most users are likely to prefer a system which is offered in their native language.

Translating the interface itself is possible by resource bundles containing the translated content [80]. In effect, this is a requirement that the user places on the *developer* role. Offering a localized animation, on the other hand, is a requirement placed on the *visualizer* who has to prepare appropriate translations [178, 179]. However, this also requires special support within the tool to either let the user choose from the list of encoded languages for each loaded animation, or to adapt the language to the language used for the user interface. Thus, the developer role also has to provide support for animation translation.

UR19. Smooth transitions

Smooth transitions make it easier to perceive the currently performed operation [197]. Gradual change allows the eye more time to notice the changing elements. It also frees the user from trying to find the difference between one picture and its ancestor, possibly from the internal representation in the user's memory. Many authors agree that smooth transitions are helpful, for example [27, 55, 2, 134]. On the other hand, applying smooth transitions to large data sets may be impractical as the operations may take too long and may make the user impatient.

Anderson and Naps [2] report that an informal study found that some users preferred smooth animations, while others preferred discrete snapshots with breaks in between. It should be noted that the system underlying the study used smooth transitions without providing default breaks between steps (**UR6**). It also offered no rewind facilities.

UR20. Color incorporation with "useful" settings

Colors can be very powerful for communicating information efficiently. Example applications may use colors to encode the state of data structures, highlighting of activity, tying multiple views together, emphasizing patterns and providing a visible history [27].

However, human computer interaction research shows that not all color combinations are equally suited. Unreflected color choice or an abundance of colors may hinder rather than

help user understanding [57]. Additionally, about 8 per cent of men and 1 per cent of women are color blind in varying degrees. Using color cues as the *only* means for conveying a certain information is therefore dangerous, especially if the brightness of the colors is similar. The developer of a system should pay attention to this fact and try to use colors that exhibit a significant brightness difference. Note that apart from color blind users, the brightness difference is also decisive regarding the quality of grey scale printing of screen shots. Further hints about using colors are given in [57, 99].

UR21. Easy customization

Each user is likely to have his or her own set of preferences regarding the user interface. This mostly concerns the colors employed, as stated in **UR20**. For example, Brummund [32] presents an animation of different sorting algorithms that can be adjusted by setting diverse color properties. Other areas of customization include the amount of information presented (see **UR16**) or its format. The work by Velázquez-Iturbide and Presa-Vázquez [207] supports templates that allow the user to specify the textual format of expression used for the animation.

3.3.6 File Exchange

UR22. Animation saving

This requirement is fulfilled by most systems incorporating graphical input. However, other systems are also advised to support storing the full animation or parts thereof in an internal format. The process of evaluating the input and building the animation may take a certain amount of time. In some cases, it may not be possible to recreate the precise animation content, mostly if the algorithm incorporates random events. Systems incorporating generation by declaration, API method invocations or code interpretation can benefit the most from an internal storage format. Scripting-based systems benefit less from this ability, as their content is usually already fixed to disk.

An internal, efficient storage system allows quick retrieval of a given animation. This is especially helpful if the animation is propagated over the WWW. A variety of compression algorithms can be used to further decrease the size of the file to be transmitted to the client. Java contains support for the *gzip* and *zip* compression algorithms popular with both UNIX and Microsoft Windows™ users.

UR23. Export facilities

Users are likely to appreciate the ability to export parts of the animation. Typical examples of export include storing the display as an image in one or more popular formats [128]. Video formats are obvious candidates for export, as they are able to appropriately convey the dynamics of the animation. Other possible export formats include customized animation languages specified in the increasingly popular XML [78], as well as formats used by other animation systems.

Despite the presence of special libraries such as Apple's Quicktime API [10], exports to video formats are still very uncommon. Similarly, some freely available APIs support image generation. Alas, most animation systems do not even offer facilities for exporting animations to other animation tools.

UR24. Import facilities

In a similar vein to **UR23**, users are likely to appreciate import filters. These filters can be used to load and possibly convert "legacy" animations. They also allow a user to access a variety of resources built by visualizers using other systems and incorporate them into the chosen system. Alas, import filters share the same fate as export filters, with most tools supporting only their own format.

3.3.7 Algorithm Understanding

UR25. Focusing of attention

There are various techniques for focusing the user's attention on the salient aspects of the algorithm. Color usage has already been mentioned in **UR20**. Other possibilities include shape analysis techniques [23], using only small data sets [51], incorporation of interactive predications (**UR10**), highlighting the main steps of the algorithm (**UR12**) and shading out of currently irrelevant elements [174]. Other techniques such as blinking may irritate the user and distract from the content [55]. Sound effects can also be used to get or direct the user's attention [28].

UR26. Didactical structure of animation

The structure of the animation has a significant effect on the user's chance of understanding the topic [34]. Faltin [55] proposes a bottom-up approach that starts with an outline of the problem and known solutions. After comparing relevant algorithms, the data structures used in the concrete algorithm selection are presented. This is followed by an introduction of the methods used by the algorithm and concluded with the actual implementation. The presentation could end with a discussion of the algorithm's efficiency, as described in **UR13**.

UR27. Small data sets for introduction

To avoid confusing especially novice users, any animation should start with a small set of data. Some researchers claim that at most seven elements should be used or changed in each step [55]. This ties in with psychological research which shows that short-term memory can store five words or shapes, six letters, seven colors and eight digits [99].

Starting with a small set of carefully chosen data makes focusing on the salient aspects of an algorithm easier [51, 2]. Larger data sets may be necessary for illustrating key properties of algorithms [27]. Hashing animations operating on a small data set may have to resort to "cooked" data as described in **UR11** to illustrate collision handling. Using a large, representative data set offers a realistic view of hashing, but may initially confuse the user. To avoid this confusion, the general principle should be introduced on a small data set, followed by the larger data set that shows the typical behavior.

UR28. Loop shortening

The user should be able to skip the rest of a loop after understanding its operation. It is thus often sufficient to perform only a few iterations of the loop fully and abbreviate or drop the rest at the user's discretion. Note that the user should be able to control *when* the rest is skipped, as the visualizer cannot predict the user's level of understanding. Gloor hints at this capacity in the third, fifth and seventh of his "ten commandments of algorithm animation"

[67]. Several other researchers also include the requirement, for example [68, 55, 98, 205, 210].

UR29. One "interesting event" per step

A single step in the animation should not be overburdened with operations. From the user's perspective, each step should contain at most one "interesting" event per step [27, 55]. Note that there may be intermediate steps which do not contain an "interesting event". These steps may focus on providing an explanation of what happened or is to happen, or show the preparation of the next "event".

The precise definition of what constitutes an "interesting event" varies between authors. The structuring approach advocated by Faltin [55] shows only interesting events. BALSA shows the algorithm state at defined breakpoints [31]. In both cases, the visualizer defines which elements are considered "interesting".

UR30. Adjustable speed control

Gloor regards speed adjustability as an aspect of the fifth of his "ten commandments of algorithm animation" [67]. Several other reports also stress the importance of a speed control [65, 99, 205]. The most well-known veteran system of algorithm animation, *BALSA* by Brown and Sedgewick [31], already incorporated a speed control. Since then, speed control has been added to many systems that use a slide show approach of displaying the animation, such as *JSamba* [196]. Step-based systems that wait for a user action to show the next step usually have less demand for a speed control.

3.3.8 Miscellaneous

UR31. "Graceful degradation"

GR8 discussed that the tool should offer "sufficient" performance. Gloor [65] also demands that the tool should degrade gracefully if running on a slower machine than anticipated. There are basically two alternatives for achieving this goal: leaving out intermediate image frames to "catch up", or showing the full content, only slower.

The first approach may result in jerky, unpredictable behavior, but maintains a certain degree of faith to the original timing constraints. The second approach ignores possible timing constraints without leaving out anything. Watching the animation may become unbearable if the user's computer is far too slow. While there is no easy solution to the dilemma, a good tool should offer some way of addressing this issue.

UR32. Incorporation of a "history"

Some reports indicate that a history of the previous step or steps is required for algorithm understanding [51, 84]. Gloor has made the inclusion of a history the last of his "ten commandments of algorithm animation" [67]. One way of offering a history is by showing the current and former value side by side [100, 101, 148].

UR33. Consistent presentation over diverse animations

A set of related animations should preferably be presented in a consistent way. Gloor makes this demand the first of his "ten commandments of algorithm animation" [67]. Consistency

concerns both the visual presentation [98] and the control elements [65, 99]. Gloor [65] states that the advantages of using a consistent view of a family of related algorithms include easier generation and adaptation as well as better comparability. Using a customized presentation that highlights the specific salient aspects of each algorithm in a family communicates its intrinsic workings better [65]. However, it also forces the user to adjust to the different notation employed.

UR34. Clear mapping of code to animation

The easiest way to fulfill this requirement is by letting the user specify the mapping as described in **UR8**. Several other techniques have already been presented, such as including textual explanations (**UR1**), using pseudocode (**UR2**) or interactive predictions (**UR10**). As Khuri [99] points out, finding an appropriate mapping is non-trivial and requires time, effort, careful thought and knowledge of the particular animation tool. Without providing a clear mapping, especially novice users may be unable to profit from the animation [98].

UR35. Restriction to meaningful user events

Interactive systems that allow the user to select elements or provide input as described in **UR9** should detect faulty or meaningless inputs. Gloor [67] states that these types of input should be disregarded. Optionally, a dialog may alert the user that the input is incorrect or meaningless.

One easy way of preventing meaningless interaction is by disabling control elements which are inapplicable in the current situation. For example, a button for rewinding the animation is meaningless if the animation is already at the first step.

3.4 Visualizer Requirements

As described in section 2.3, animations are generated by visualizers. The design of the animation and especially the mapping from code to visualization is therefore the main task of visualizers. In order to provide the "best" possible animation, the animation tool has to support a set of specific operations. Note that the importance visualizers place on the individual requirements may differ according to their personal preferences and understanding of what constitutes a "good" animation. Some user requirements as well as most general requirements also apply to the visualizer. See Table 3.1 on page 66 for a detailed mapping. The visualizer requirements are split into the categories applicability, animation display and animation generation.

3.4.1 Applicability

VR1. Focus on understanding the data structures and algorithmic steps

This should be the main focus of any educational algorithm animation. Part of the problem lies in enabling the user to understand the mapping between the algorithm and its visualization [81, 82, 51, 55, 2, 99]. Various ways for helping the user overcome this problem are listed in the previous section and mapped in Table 3.1 on page 66. Slightly simplified, the more freedom the visualizer is granted in designing an animation, the better. However, visualizers should also check the user requirements in section 3.3. An animation tool that offers

good support for generating animation, but is awkward to use for end users, may prevent even very good animations from being used.

VR2. Extensibility

As Khuri points out, no single animation tool will ever be universally superior across all kinds of users and tasks [99]. Despite this fact, there are a number of criteria that allow the visualizer to determine which system is "better" for their application. One central aspect for determining this is the extensibility of the system. The developer cannot anticipate future uses or application areas. Regular updates may address this issue somewhat; however, it is usually preferable if the visualizer can add extensions to the tool without having to download a completely new release - see also **GR9**.

The extensions may be implemented by the visualizer who then assumes the *developer* role. Depending on the structure of extensions, they may also be made available for download by other programmers over the Internet. Some systems explicitly support extensions [36, 38, 113, 178], though implementing them may require a deep knowledge of the target system [113].

VR3. Configurability

Similar to **UR21**, this requirement concerns the behavior and appearance of the tool's user interface. However, the focus lies more on the *generation* aspect. For example, the visualizer may be able to define the (textual) output format using a template [207], specify which input data is to be used [116], adjust color settings or delays between steps. Some of these settings may also be applicable to the user. If the system is extensible (**VR2**), the visualizer should also be able to add or remove extensions from the configuration.

VR4. Internationalization in generation front-end

UR18 requires that animations and the user front-end for displaying animations shall be translatable into the user's native language. The same arguments apply to the interface used by the visualizer for generating animations. See the discussion of **UR18** on page 54 for more details.

VR5. Incorporation of several generation approaches

Each generation approach presented in section 2.4 has its particular strengths and weaknesses. Visualizers are advised to determine the approach best suited to their targeted use of animations. For example, on the fly generation of animations effectively precludes manual generation in a graphical user interface. Another important consideration, however, lies in the visualizer's preferences, skills and prior experience. For example, API-based generation may not be feasible for a visualizer without programming experience.

One important aspect is that expectations in a tool grow over time as the visualizer's skill increases. Manual generation and direct code interpretation probably present the lowest initial hurdle for novice tool users. However, these approaches usually lack support for automatic generation and fine-tuning the display, respectively. The visualizer is therefore likely to appreciate a tool that "adapts" to his or her skill by providing increasingly more expressive and powerful ways of generation.

For example, a visualizer could proceed in the following way in using a given tool. First, a few animations are generated manually to get acquainted with the features of the tool. After

a certain level of mastery in the tool is reached, the visualizer may be interested in using a quicker way of animation generation. One possible approach is using a scripting language [65]. The scripting language code can normally also be generated within program code.

The next step could therefore be the addition of appropriate output statements into the algorithm to be animated. Output statements of this kind can significantly lower the legibility of the underlying implementation. Therefore, the visualizer could shift to API-based generation by exchanging the output statements with appropriate API method invocations. Note that this progression requires a tool that can handle animation generation in a GUI, by scripting *and* by API invocations.

VR6. Educational use

The central consequence of this requirement is that the tool should be freely available (**GR7**) for both teacher and students. Therefore, a free "player" version of the tool should be available for all major platforms (**GR1**), and the tool should be easy to download and install for the students (**GR9**). One possible way to achieve this is by assembling all components on a CD that can be used as a shared resource by teachers and students [167]. In order to be helpful in teaching, the tool should also fulfill most of the user requirements. Otherwise, the students may be reluctant to use the tool.

3.4.2 Animation Display

VR7. Establishing own conventions

To allow users to compare different animations of related algorithms, the visualizer should be able to develop consistent views (**UR33**). This can entail an "appropriate" set of conventions to denote different information [99]. For example, the shape, size, color or arrangement of objects may be used to convey information. It may also concern the division of screen space [99]. The tool should be prepared to support the visualizer in establishing such conventions.

VR8. Step skipping

The visualizer should be able to skip less interesting parts of the algorithm. This may concern repetitive evaluations, for example of loops (**UR28**). Other application areas are lengthy initializations or complex sub-operations that may distract from the main algorithm. Skipping these segments may allow the visualizer to focus on the more salient aspects of the algorithm without boring or confusing the users.

Skipping is very easy to manage in some generation approaches, but may be problematic in other approaches. For example, manually generated animations by definition only present what the visualizer has explicitly embedded. Code or step skipping is simply accomplished by not entering the information. API- and scripting-based as well as declarative systems may offer special operations that turn off animation effects. Alternatively, the visualizer may simply refrain from adding animation statements to the less interesting algorithm segments. Tools based on code interpretation may present problems if they offer no possibility for the visualizer to exclude code segments or at least set breakpoints.

VR9. Optional multiple views

As stated in **UR16**, multiple views may present different aspects of an algorithm [27, 107] or use different visualizations of the same underlying data [2]. However, some authors also criticize the use of multiple views as possibly causing confusion due to an information overload [99].

VR10. Close link between system and lecture materials

Brown *et al.* [17] claim that the success of the BALSA system at Brown University would not have been possible without a tight integration with the other resources for the algorithms and data structure course. It is reasonable to assume that this also holds for other systems. The closer the link to the lecture materials, the more relevant the tool becomes for both visualizer and user [21, 167]. Anderson and Naps place this demand at the top level of their instructional design scale [2].

3.4.3 Animation Generation

VR11. Automatic ad-hoc generation

Several studies stress the importance of automatic generation support [48, 130, 110, 128, 129]. Automatic generation frees the visualizer from the time-intensive task of explicit animation generation and allows for a consistent presentation of animations of the same topic. Additionally, animations can be generated on demand and on the fly. For example, if animations of Quicksort are generated automatically, the visualizer can respond to most question types by showing an appropriate animation.

Note that only code interpretation offers "free" automatic generation. Most other generation approaches require the visualizer to annotate the underlying algorithm with "appropriate" statements. The annotation may also be partially automatic, but this may result in unsatisfactory results. Manual generation approaches within a GUI typically do not support automatic generation.

VR12. Quick generation

Hundhausen and Douglas [84] state that the generation of a "high fidelity" animation of textbook quality took "just over 33 hours", most of which was allocated for graphics programming. As explained in Hundhausen's dissertation [85], the system used for content generation was *JSamba*. None of the students had previously worked with this system and therefore had to spend a certain amount of time in getting familiar with the system. The lack of advanced features such as relative placement may also have played an important part in the long generation process. See also **VR14** for a discussion of relative placement.

From our experience, a duration of about six hours for manually generating an animation seems more typical. The amount depends on the visualizer's experience, the ease of use of the tool and the degree to which automation is supported. By replacing manual generation with scripting, we could reduce the time required by a factor of roughly 3. Part of this is probably due to our familiarity with the underlying scripting language.

VR13. Reusable animation components or modules

Sometimes visualizers may have to provide a set of animations for the same general topic. As a typical application, the visualizer may generate multiple animations of the same sorting algorithm that expose different aspects. For example, one animation each could illustrate the best, worst and average case. Additional animations could be used to illustrate specific algorithmic behavior such as the sorting stability. The visualizer may also want to start the animation with general comments about the sorting algorithm as described in **UR1** and end it with a complexity analysis (**UR13**).

Generating basically the same start and end step sequence for each animation of a related topic is irritating and time-consuming. It is possible to generate a "template" animation that contains the standard start and end parts and use this to generate the other animations. However, it would be both "cleaner" from a programming point of view and easier if the visualizer could simply generate animations representing the start and end part, respectively, and embed those into the finished animations. Note that this may also require the passing of variable values, if the actual number of comparisons and assignments in a given invocation is to be used for the complexity analysis.

VR14. Relative object placement

Relative placement allows the visualizer to place a new object at a certain distance from another object. Alternative placement strategies are automatic (determined by the tool) or by absolute coordinates. Automatic placement may result in unfortunate layouts and has certain limitations. For example, optimal graph layout is NP-complete for most definitions of "optimal" [62].

Absolute coordinates force the visualizer to specify fixed coordinates for each object. Instead of fixed coordinates, normalized device coordinates can also be used [196]. This makes aligning objects very difficult for the visualizer. In some cases, placing two objects directly next to each other may require keeping track of their location and calculating the target positions. This is awkward to do and also reduces code legibility.

Relative placement allows the visualizer to specify coordinates by giving a reference object and an offset. The *bounding box* of the reference object can be used for precise placement [197]. More elaborate placement options may allow the reference to be an arbitrary node of a polygon or polyline or a user-defined location. The bounding box is also ill-suited for aligning text components, as the lower edge of the bounding box depends on the presence of characters that extend beneath the text base line such as `g, j, q, y`. For this application, it would be preferable if the text's base line could be used as an anchor point.

Both the *ALVIS* system [83] and *DANCE* [195] offer good support for relative placement. For example, "right-of" and offsets based on edges of the bounding box of a given object can be used.

VR15. Ease of learning

Most visualizers will not be interested in spending much time in becoming familiar with a given tool. Note that the requirement of *quick generation* (**VR12**) does not necessary imply that the generation of the *first* animation is also quick and easy to achieve! As stated in **VR5** and **VR12**, a tool that supports multiple approaches for generating animations may

significantly lower the initial learning hurdle. The ease with which a tool can be learned also depends on the visualizer's skill and previous experiences.

3.5 Developer Requirements

Developers of a system in our definition have only a small selection of requirements, as they can only place requirements on an *existing* system. Therefore, the development process of the initial release can be disregarded. The only remaining requirements are those that provide extensions for the system. Therefore, this section is restricted to systems that support extensibility (**VR2**).

DR1. Easy addition of components

In general, adding components should be as easy as possible. As perhaps the most important aspect, the amount of system code to touch should be minimal. In an extreme case, the developer may not have to touch *any* existing system code. This can be achieved by using advanced software engineering technology concepts such as *component-based software*, *aspects* [102] or *introspection*.

The system should preferably offer support for registering new components in a single place. This cuts down on the likelihood of forgetting to register a new component in one place. It also makes testing experimental additions much easier and allows for quick removal of dysfunctional units - see also **VR2**.

DR2. Extensibility documentation

DR1 states that it should be *easy* to add extensions by limiting the segments of code to touch. In addition, the *location* of the relevant code pieces must be well documented. Otherwise, external developers may have to go through the code line by line to determine the correct place for extension registration.

A sad experience with some software systems is that the presence of documentation is not sufficient, as some references are barely understandable. Additionally, the documentation should outline the general principle underlying the tool. In this way, developers stand a better chance of understanding both *where* and *how* they can extend the system. For example, the tool *Catai* [36] provides a short overview of what has to be done to add new operations. See also Meyer [119] for a discussion of extensible system design.

DR3. Extension without detailed system knowledge

Even if the locations of code to touch are limited in number, the level of tool familiarity required for appropriate modification should be as low as possible. For example, [113] provides a short overview of how *Jeliot 2000* is implemented and how new animators can be added. However, the authors also state that doing so requires detailed knowledge of the implementation.

3.6 Programmer Requirements

As defined in section 2.3, programmers are responsible for implementing the original algorithm. They are not concerned with a possible animation. Thus, there is little that an animation tool can offer to a programmer except for "standard" support from which any programmer can profit.

PR1. Code editing in an IDE

Integrated Development Environments (IDEs) are often used for implementation purposes. Examples include the well-known *Turbo* products by Borland, for example *Turbo C++*, and various environments for Java like *Borland JBuilder* or *Symantec Visual Café*. Typically, an IDE allows the programmer to write and edit the source code, compile and execute it. It may also include additional features such as debugging and generating user interfaces by drag and drop.

Integrated environments that incorporate animation, for example *Alice* [42], *Empirica Control* [111] and *Leonardo* [39], can significantly help both programmers and visualizers. Naps [134] states that animation as part of a programming environment is helpful for algorithm understanding in both the *analysis* and *synthesis* aspects of Bloom's taxonomy [19].

PR2. Standard libraries support

Another feature that can assist programmers in developing algorithms is support for standard libraries such as the *C++ Standard Template Library (STL)* [153] or *LEDA* [117]. Such libraries are usable independent of the actual visualization, and free the programmer from having to reinvent the wheel. Note that this is only helpful if the system used by the visualizer is able to animate the data structures specified using the library. For example, the *Catai* [36] system supports the *LEDA* C++ library [117].

PR3. Programming environment testbed

This requirement needs the animation system to be extensible (**VR2**). In addition, there must be a good documentation of how the system can be extended (**DR2**). Additionally, the system's extensibility should not require a detailed knowledge of the "inner workings" or advanced programming and software engineering skills (**DR3**). If these requirements are met, the system may be used as a test environment for programming. For example, an educator could ask the students to implement extensions. Provided that the system is easy to customize (**UR21**), the educator may also remove some components and ask the students to re-implement them. Such assignments could be used in any programming or object-orientation course once the students have reached a certain skill level in programming.

3.7 Summary

There are several ways for choosing an algorithm animation system. Basing the decision on recommendations by other users or the number of publications may cause frustration after extended use. For example, the chosen system may prove incapable of supporting all operations to be illustrated, without supporting the later addition of components. "Complete" lists of algorithm animations or animation systems may be incomplete and out of date.

In this chapter, we have explored an alternative approach. Based on the evaluation of a large but incomplete set of animation systems in section 2.5, we have stated 69 requirements regarding

algorithm animation systems. The requirements are split in five categories addressing 13 general requirements, and the specific requirements associated with each animation role: 35 *user*, 15 *visualizer*, three *developer* and three *programmer* requirements. Many requirements also have implications for other roles. For example, several requirements placed by the user or visualizer set specific implementation goals for the developer role. The complete list of requirements is included in the appendix on page 175.

Table 3.1 on the following page illustrates the roles touched by the individual requirements. A check mark "✔" is used to indicate that the requirement applies to the role. Parentheses indicate that the relation is indirect, typically because the requirement depends on certain actions from another role. For example, **UR9** requires support for input data specification. This is also an interesting feature for the visualizer role, as users will profit more from generating their own input than from merely using predefined values, as discussed for example in [67, 2, 134]. However, the visualizer cannot fulfill the requirement without specific system support provided by the developer. For example, the system - and thus also the developer - has to provide appropriate GUI elements that allow easy specification of the input values.

The reader will probably regard some requirements as more important than others. We do not claim that all requirements are equally relevant on an objective scale. Each reader may assign a slightly different importance to the different requirements. For example, the demand for platform-independent systems (**GR1**) may be regarded as irrelevant in a tightly controlled environment sharing the same operating system. However, several educational environments use some variant of Linux or UNIX operating systems, while most students probably use Microsoft Windows™ on their personal computers. In these cases, the requirement becomes highly relevant if the students should also be able to use the system at home. Therefore, the reader should carefully study the requirements for all roles and decide on the relevance of each requirement in his or her environment. The implications of the requirements on the system design are discussed in the following chapter.

Requirement	User	Visualizer	Developer	Programmer
GR1	✔	✔	(✔)	
GR2	✔	✔	(✔)	
GR3	✔	✔	(✔)	
GR4	✔	✔	(✔)	
GR5	✔	✔		
GR6	✔	✔		
GR7	✔	✔	(✔)	
GR8	✔	✔		
GR9	✔	✔		
GR10	✔	✔		
GR11	✔	✔		
GR12	✔	✔		
GR13	✔	✔		
UR1	✔	✔		
UR2	✔	✔		
UR3	✔	✔		
UR4	✔	✔		
UR5	✔	✔		
UR6	✔	✔		
UR7	✔	✔	✔	
UR8	✔	(✔)		
UR9	✔	(✔)	✔	
UR10	✔	✔		
UR11	✔	(✔)		
UR12	✔	✔		
UR13	✔	(✔)		
UR14	✔	✔		
UR15	✔	✔		
UR16	✔	✔		
UR17	✔	✔		
UR18	✔	✔	✔	
UR19	✔	✔		
UR20	✔	✔		
UR21	✔	✔	✔	
UR22	✔	✔		

Requirement	User	Visualizer	Developer	Programmer
UR23	✔	✔		
UR24	✔	✔		
UR25	✔	✔		
UR26	✔	✔		
UR27	✔	(✔)		
UR28	✔	✔		
UR29	✔	✔		
UR30	✔	✔		
UR31	✔	✔	✔	
UR32	✔	✔	✔	
UR33	✔	✔		
UR34	✔	✔		
UR35	✔	(✔)		
VR1		✔		
VR2		✔	✔	
VR3		✔	✔	
VR4		✔	✔	
VR5		✔		
VR6		✔		
VR7		✔		
VR8		✔		
VR9		✔		
VR10		✔	✔	
VR11		✔		
VR12		✔		
VR13		✔	✔	
VR14		✔	✔	
VR15		✔		
DR1			✔	
DR2			✔	
DR3			✔	
PR1				✔
PR2				✔
PR3				✔

Table 3.1: Requirements Overview

Chapter 4

The ANIMAL-FARM Algorithm Visualization Framework

4.1 Introduction

A *framework* consists of a set of cooperating classes that together represent a reusable design for a specific class of software [61]. In this chapter, we focus on motivating and describing a framework for algorithm visualization. The framework defines the basic architecture of all AV systems built using it. In general, it will emphasize the ability to reuse the basic design rather than individual classes or pieces of code.

Developers of AV systems can use the framework presented in the course of this chapter. They then have to focus only on the implementation of the basic classes which are called from the framework. The framework includes fixed and ready-to-use components as well as abstract classes and interfaces that have to be filled with content by the developers. All AV systems based on the framework will therefore have the same core components and a similar structure of classes, making them easier to maintain and more consistent.

Specifying a good framework structure is very difficult. Significant thought has to go into making certain that the framework can be reused easily on the basis of its merits regarding understandability, conciseness and easy applicability. The benefit of the framework is that it can make developing new concrete implementations much easier.

One central aspect in framework design is to make it as flexible, expressive and at the same time extensible and adaptive as possible. Within the course of this research, we have specified a framework for AV systems that satisfies these requirements. This framework is called ANIMAL-FARM, an acronym for "*A*dvanced *N*avigation and *I*nteractive *M*odeling of *A*nimations for *L*ectures - *F*ramework for *A*nimation *R*esource *M*anagement". The framework title already hints towards some of the development goals. Additionally, one property of an animal farm is that it can house multiple animals - and accordingly, the prototypical AV system built based on this framework, as described in the next chapter, is called ANIMAL. This system also illustrates the expressiveness of the framework. The remainder of the chapter describes the components of the ANIMAL-FARM framework. Section 4.2 illustrates how frameworks can incorporate adaptivity and extensibility. Section 4.3 presents an overview of the participating conceptual entities. Sections 4.4 to 4.9 introduce the components of the framework, including graphical objects, animation effects and displaying and editing front-ends. Section 4.10 summarizes the main aspects of the framework.

4.2 Concepts for Extensible Framework Design

We have to design a framework structure for AV systems that allows us to fulfill as many of the requirements in chapter 3 as possible. The requirements that have the greatest impact on the design of the framework are *extensibility* (**VR2**) and *configurability* (**VR3**). Related to the extensibility requirement are the developer requirements for easy addition of components (**DR1**) based on a good documentation (**DR2**). To support the generation of additions, the implementation process must not require deep system knowledge (**DR3**).
In the following, we will refer to additions as *extensions*. Our goal is the definition of a framework that can be modified by programming extensions and by configuring the participating objects. To make the generation of extensions as easy for the developer as possible, we shall strive to make the modification of existing code unnecessary wherever possible. As we will see, this goal can be achieved.
In this section, we examine approaches that support extensibility and adaptivity on a framework level. Providing these features within a framework shifts the burden of designing them from the developer of a concrete system or extension thereof to the framework developer. Essentially, this means that a large investment of time and effort once saves much time for all later developments based on the framework. The invested effort therefore pays off quickly.
In the following subsections, we will regard one application area for extensibility and flexibility each. The areas we want to incorporate into an extensible framework concern the object state representation (subsection 4.2.1), support for flexible import and export layers (subsection 4.2.2), internationalization and GUI element generation (subsection 4.2.3), dynamic component assembly and administration (subsection 4.2.4) and decoupling objects for better reuse (subsection 4.2.5). Subsection 4.2.6 summarizes the key concepts and briefly evaluates their strengths and weaknesses.

4.2.1 Using Properties to Model Object State

One of the most basic design decisions to make for any framework or system is the representation of object state. Most object-oriented systems model object state by *attributes*. These are declared and instantiated for each new object of a given class, so that each object has its own copy of the attribute. In some cases, a given attribute has to be shared by all objects of the same type. Java supports this with the `static` keyword, which essentially states that a shared copy of the attribute is kept.
The advantage of declaring attributes for modeling the object state are well-established: fast access to the data and type checking [139]. However, modeling object state by a set of declared attributes also has drawbacks. First of all, the user has to choose an appropriate access mode for each attribute. Java offers *private, protected, public* and *default* access for attributes or methods. The mode is chosen by placing the associated access modification keyword before the attribute or method name, with the following consequences:

- *private* attributes or methods can only be accessed within the class, and are thus invisible even to inheriting classes;
- *protected* attributes or methods are visible within all classes of the package and subclasses regardless of the package they belong to;
- *public* attributes or methods can be accessed by any class;

- default access is assumed when no keyword is given. It limits access to the package and therefore also prevents access within subclasses placed in a different package.

System-wide *public* access has to be granted if an attribute has to be accessed in a different package without an inheritance relationship between the providing and requesting class. The alternative solution is to declare most, if not all, attributes as private to the class, and provide methods for accessing or modifying the element's value. Considered good practice in object-orientation [119], external attribute value changes are prevented, as any assignment must be performed by invoking the special "set" method. Furthermore, the approach can be used to provide conditional access. For example, a method for setting the value may check the value subject to specific constraints and set it only if the constraints are met.
Another concern in representing object state by attributes is that a new attribute may have to be introduced whenever new capabilities are added to a class. For example, if users of a graphics tool decide that circles also need a fill color which is not supported in the base system, a new color attribute has to be added to the class representing circles. Following the modeling outlined above, two methods for read and write access may also be required. Thus, each change in the attributes of a given class requires changing the implementation including a recompilation of the sources. These changes may propagate to other methods, other classes or even different packages. For example, the declaration of a fill color and the appropriate access methods does not automatically affect how the object is stored or displayed.
The last, and in some cases most awkward, drawback of using attributes with a method pair for retrieving and setting the attribute is the large method interface of the class. As an example, we regard a system where most attributes are declared as private and are thus only accessible using the standard pair of access methods. This leads to very cumbersome invocations for initializing a given object to non-standard values. There are three ways to deal with this situation: the invocation of relatively few methods with a large number of parameters, passing a special data container parameter, or many subsequent invocations of access methods for setting one attribute each. All three approaches may be irritating and exhausting to a developer, especially if the method invocation order is relevant. If some methods expect multiple parameters of the same type, the developer also has to figure out the meaning of each parameter, which may be challenging if the documentation is sparse. In addition, some method invocations might be forgotten.
Java offers a partial solution for this unsatisfactory situation by using *properties*. Java properties are instances of the *java.util.Properties* class that represents a specialized hash table for *(key, value)* pairs of type *String*. Basically, the state of a given object is no longer represented by a collection of locally declared attributes, but rather by a property object. Property objects can also be loaded from and stored to a file.
Instead of working directly on the value of locally declared attributes, operations change the values associated with a given key in the property dictionary. The user may also provide convenience methods for setting or retrieving a given property. These methods have a very small interface consisting of a single parameter or return value. The value is stored at the associated key by copying or cloning the parameter, depending on the usage context.
The *java.lang.Properties* class inherits the insertion and retrieval methods of its superclass *java.util.-Hashtable* that allow the usage of arbitrary objects as keys or values. However, the usage of the inherited methods is strongly discouraged. The presence of even a single non-String key or value causes invocations of the built-in storage support to fail. Attempts to retrieve a property value which is not of String type returns the object's String representation. This is usually generated by

the *toString()* method inherited from the base class *java.lang.Object*. The default representation of any object is its class name followed by the "at" sign @ and the object's hash code. Unless the *toString()* method is overridden by the retrieved object, no information about the object values can be extracted from the String representation.

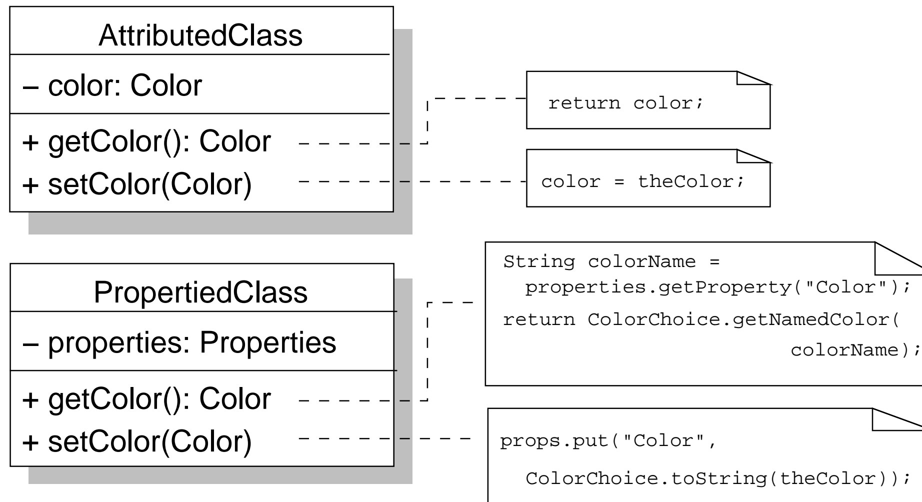

Figure 4.1: Modeling Attributes Using *Properties* with Customized Conversion Wrappers

This problem can be addressed by generating an appropriate *String* representation for each stored object. Figure 4.1 illustrates how *Color* objects can be stored and retrieved. The upper part of the figure shows a "standard" implementation using a color attribute. The lower half illustrates the equivalent implementation using properties.

The example assumes the presence of a *ColorChoice* class that converts instances of *java.awt.Color* to their String representation and vice versa. The resulting String object may have any format that allows the extraction of the values. Some typical encoding examples for the color *red* that allow for easy extraction of the values are textual ("`red`") or using the color coordinate in the RGB color space, typically coded either in hexadecimal ("`0xFF0000`") or decimal ("`(255, 0, 0)`"). The method for decoding the color value has to know the format used to convert the String representation back into a *Color* object.

The framework provides an extension of the *java.util.Properties* class called *XProperties* which provides customized conversion methods for a variety of input types. The class is placed in the package *animal.misc* that contains support classes of the framework. Figure 4.2 shows the class diagram of this class. The *XProperties* class contains a constant *Point* object representing the point of origin, used as the default return value for Point requests. Two class methods support the conversion of an integer array or segment thereof into a String. Another class method extracts a String array from a single String object using the element separator passed in the second parameter. These convenience methods are independent of the actual properties attribute and help in generating appropriate entries. Additionally, all storage and retrieval methods perform appropriate runtime exception checking and catch possible exceptions, as they may occur if a property value is cast to a wrong type.

XProperties allow the storage and extraction of *boolean, double* and *int* primitives, as well as *Color,*

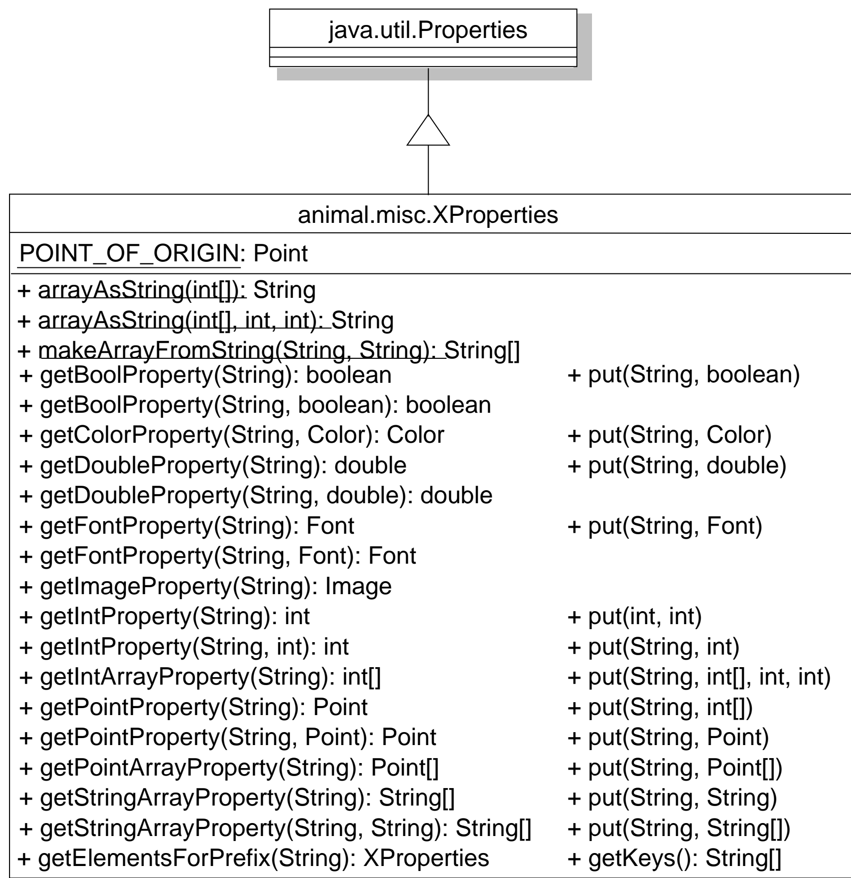

Figure 4.2: Class Diagram of the Extended Property Support Class *XProperties*

Font, Image and *Point* objects. Arrays with underlying type *int, Point* or *String* can also be inserted and retrieved. The set of keys can also be requested. Additionally, the *getElementsForPrefix* method returns a new *XProperties* instance which contains the set of all (key, value) pairs where the key starts with the prefix passed as a parameter. This operation is helpful if properties belonging to the same object or context are prefixed with a keyword followed by a separator. For example, Java commonly employs dots as separators for system properties such as the *java.compiler* property that allows setting the Java Just in Time Compiler.

The attentive reader may be concerned that modeling the object state by properties violates two central tenets of object orientation: data hiding and encapsulation. However, we can reassure the reader that this is not the case. Rather, a system built using properties presents two different access interfaces: a compile-time interface for accessing the "known" parts of the object state, and a developer interface for accessing those parts of object state that were not anticipated at compile time.

Those attributes that the developer considers central to the propertied object would normally represent the object state and be modeled as private attributes. A pair of *get/set* methods are then usually used for allowing access to the object state from other objects. The return value may also depend on the object having a certain consistent internal state. This type of method can still be provided after changing the object state representation from attributes to properties. Where the old method simply returned the value of an attribute, it will instead retrieve and return the associated property from the hash table. This is in keeping with the concepts of data encapsulation and data hiding central to object-oriented systems: the user need not be aware how a given method is implemented, as long as it fulfills its promised contract [119]. This is also the main application of properties in Java, featured under the title of *Java Beans*, and widely used throughout the Swing API.

Developers of extensions, on the other hand, need to be able to access parts of the object representation that could not be anticipated when the application was originally built, or may simply not have been considered. This access requires both retrieving and modifying the object's state. In "classic" attribute-based systems, adding a new attribute requires code modification and recompilation. Properties, on the other hand, allow the easy addition of attributes by adding a new key to the properties and providing it with the target value. This is also possible at run-time. The attribute can always be accessed through the properties object, even if there is no convenience method for retrieving or setting it. Other parts of the system can easily query whether the object offers a given attribute by accessing the value.

Java properties also support the installation of default values which are returned if a requested property is unknown. The default property object provides the values for keys which are not included in the object's properties. Property access calls may also specify a default value to be returned if the property currently has no assigned value. The default return value is *null*, which may cause run-time problems. Providing a non-null default value makes many run-time checks unnecessary and also helps to prevent null pointer exceptions. Our specialized *XProperties* class always returns a default non-null value, freeing the programmer from checking whether the requested property is known.

Figure 4.3 illustrates the support for default properties. The default properties object *defaults* is stored as a separate *Properties* object in object *object1*. When *anObject* requests a property, *object1*'s properties are consulted first, as shown in the upper part of the figure. If the requested property key does not exist in *object1*, the internal default properties are consulted and the result of this request is returned. This process is invisible to the requesting object. The requester thus also has no way of detecting if the returned value comes from the object's internal properties or the

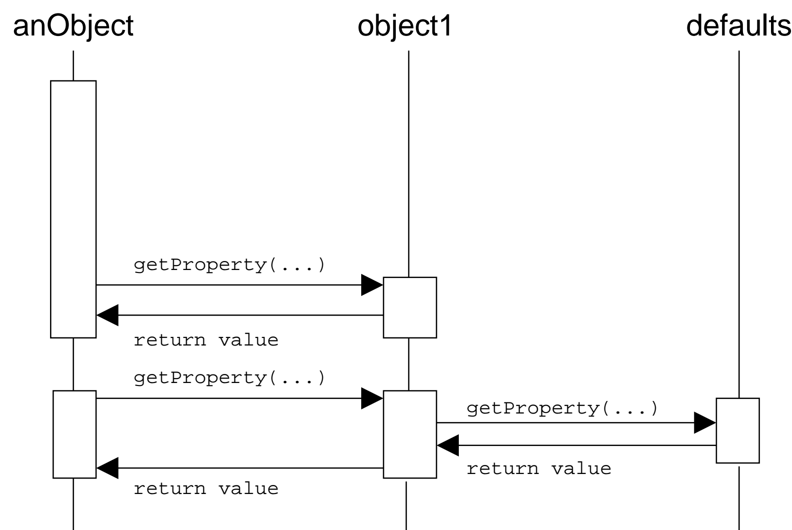

Figure 4.3: Default Properties Interaction

default properties.

Listing 4.1: Behavior of Java *Properties* objects

```
// Install the default properties to be used
Properties defaultProperties = new Properties();
defaultProperties.put("a", "d");
defaultProperties.put("b", "e");
defaultProperties.put("c", "f");

// Generate two properties objects with default properties
Properties p1 = new Properties(defaultProperties);
Properties p2 = new Properties(defaultProperties);

// overwrite the association of "a" in 'props'
p1.put("a", "g");
System.err.println(p1.getProperty("a")); // "g"
System.err.println(p2.getProperty("a")); // "d" (default)

// overwrite the assocation in the defaults
defaultProperties.put("a", "h");
System.err.println(p1.getProperty("a")); // "g": no effect
System.err.println(p2.getProperty("a")); // "h" (default)

// remove the assocation of "a" in 'p1'
p1.remove("a");
System.err.println(p1.getProperty("a")); // "h" (default!)
```

Note that the default properties are stored in a separate properties object and *not* copied into the properties table. Assigning a value to an entry thus does not affect the default properties object.

Such assignments therefore also have no side effect if the default properties serve as defaults for several objects. Direct assignments to a default properties object shared by several properties update the association in the default properties. However, this is only noticeable if the changed property is requested and unavailable in the base properties table. Alas, neither of these effects is very well documented in the Java class documentation of properties: the developer has to read very carefully to figure out the behavior. Listing 4.1 illustrates the behavior.
The *XProperties* class remedies the disadvantages of properties by providing customized wrappers for element conversion to and from String objects. A judicious use of properties instead of attributes helps in keeping class interfaces lean and also makes the code more readable. In addition, it makes it far easier to let the computer generate parsing or exporting support for objects by allowing the storage of the parsed entries into an *XProperties* object. Note that this may not be appropriate for structured notations where the order of elements is relevant.
Figure 4.4 shows two small support classes that provide additional functionality regarding editable object properties. The class *animal.main.PropertiedObject* provides methods for getting and setting a private *XProperties* object. The *setProperties* method copies a reference to the underlying *XProperties* object. Any assignments to the properties may therefore have side effects on other objects. The *clonePropertiesFrom* method offers a simple solution by inserting a clone of all properties contained in the *XProperties* parameter into the local properties. The boolean parameter determines if the local properties reference shall be newly allocated before the cloned entries are inserted.

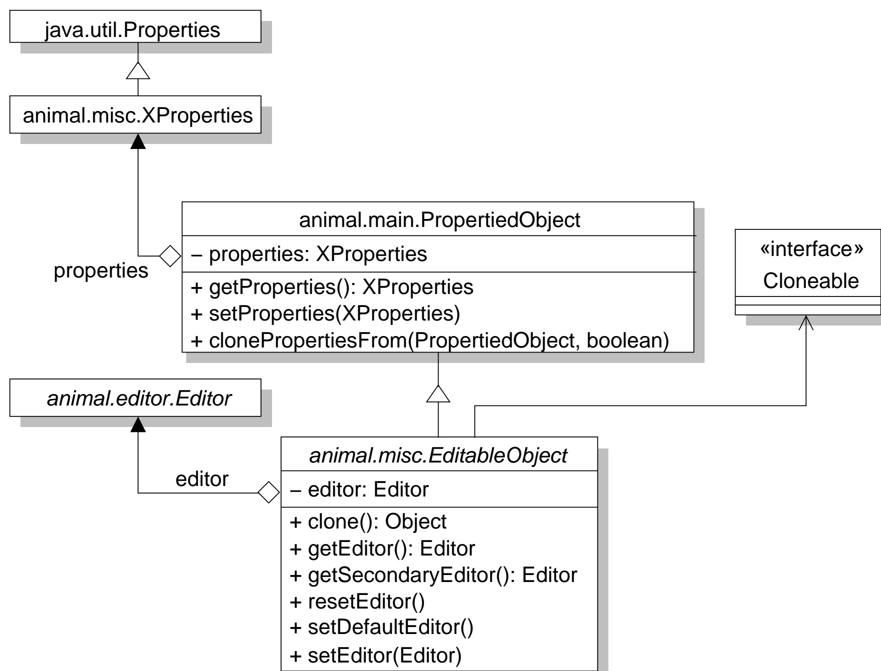

Figure 4.4: UML Class Diagram for *Propertied* and *Editable* Objects

The *EditableObject* class in package *animal.misc* extends the *PropertiedObject* class by adding a reference to an editor object. The editor is used in GUI-based animation generation or editing. Its description is deferred to the concrete system, here part of the ANIMAL prototype presented in

section 5.6. *EditableObject* instances can be cloned and provide methods for retrieving or setting the given object's editors. The *secondary editor* may represent different editing capabilities if the object can be edited in two different contexts.
Developers of a concrete AV system may consider graphical editing of the animation unnecessary or even unwanted. The editors shown in Figure 4.4 are not explicitly referenced throughout the framework. Therefore, their implementation may be left open. If editing components are wanted, they are relatively easy to implement either in specific manually generated windows or using the standard Java support for editing properties in the *java.beans* package, especially the *PropertyEditor* classes.

4.2.2 Flexible Import and Export

One of the target areas for extensibility and adaptivity common to many systems is import and export. Typically, each import or export format is represented by one class. Note that in some cases, a single class may act as the importer for several related formats, while other formats may require multiple classes or even packages. Import usually works on a *java.io.InputStream* or a String object, depending on what is more adequate for the chosen format. Export usually requires a String filename, as the concrete type of output stream may depend on the chosen format. For example, a *java.io.OutputStream* or subclass thereof is inapplicable if multiple files have to be generated.
Several of the requirements presented in chapter 3 stress the importance of the ability to import and export content. This concerns compatibilities between different releases of the system-specific proprietary format (**GR12**, **GR13**), the ability to save and export the actual content (**UR22**, **UR23**), and finally the ability to import foreign content into the framework (**GR24**).
Resolving file format compatibilities is beyond the scope of a framework, as the proprietary storage format used depends on the content type. However, the framework may be able to offer flexible components for importing and exporting content to a variety of formats. We therefore have to consider how the framework can ease the task for developers of a concrete system.
As an example, we investigate a system that can read and write its state to disk in six different formats. We assume that the formats are sufficiently different to make implementing common read and write code unattractive. We regard three approaches for solving this problem: incorporating read and write methods in each class, using the *Visitor* pattern [61, p. 331ff], or implementing a separate class or set of classes for each format.
In the first case, roughly a dozen methods per class have to be added for storing and retrieving the data from disk in each of the formats. Once a new format is introduced, all classes have to be touched by adding new storage and retrieval methods. Assuming that the number of formats is sufficiently large - think for example of word processing systems and their large selection of import filters -, the code for performing the actual operations represents a diminishing percentage of the class code.
The Visitor pattern [61, p. 331ff] offers a different approach that promises to eliminate code modifications on the underlying classes. The base visitor class declares a special visiting method for each concrete class. In our example, this could be a method *void storeXXX(XXX anXXX)*, where *XXX* stands for the concrete class to store. A second visitor hierarchy can be used for loading the contents, employing a set of *XXX loadXXX()* methods. The concrete classes now no longer need to be adapted to a new storage format, as they can simply invoke the method reserved for them in the visitor interface. The special *accept(Visitor v)* method is used for assigning the concrete visitor to

employ, allowing for example the replacement of an *ASCIIWriter* by a *XMLWriter*.
However, the Visitor pattern has one decisive disadvantage. If a new class is added to the class hierarchy - as would be expected in a dynamically extensible framework! -, an appropriate method has to be added to the Visitor base class, and implemented in all subclasses. Thus, instead of modifying each content class by adding for example *image* support when adding a new format, we now have to modify each visitor class - both for loading and storing! - if a new "content" class is added. Depending on the extent of the class hierarchy for storage and loading support on the one hand and the underlying content classes on the other hand, the situation may even be worse than it was before. For this reason, Gamma *et al.*warn against using the Visitor pattern if the underlying class structure is likely to change often [61, p. 331ff].
The third approach provides the storage and retrieval functionality in a separate set of classes. Storing or retrieving data is achieved by querying the attributes of the object to be stored in a format-specific order and dumping them to disk in the target format. Storing the attributes of a given object may require one method invocation for each attribute. Retrieving an object's state from disk also requires one method invocation for each attribute to be set.
The introduction of a new data exchange format in this approach no longer requires modifications to the original classes. However, it will result in a large number of method invocations, making it easy to forget one or more of them. The legibility of the code is reduced due to the large number of lines of code required to parse and initialize a moderately complex object, for example possessing twenty different attributes.
What can the framework do to counter these problems? First of all, the framework can define a pair of small open interfaces for importing and exporting content. Typically, import operations will work either on an input stream or come from a memory buffer that can be treated as having String format. The latter is for example the case when the content to be imported was typed in or pasted into a text field and shall be parsed from there.
If more than a single import filter is possible, there must be some way of determining the correct one. We want to keep the framework as open for extensions as possible. This also concerns the import and export components. Therefore, we have to anticipate that more than one filter each for import and export will be present. One practical decision support comes from the file extensions. However, there could be multiple different formats that have the same extensions, or formats without a default file extension. In this case, the user of the system shall be able to specify the correct import filter by selecting the appropriate MIME type.
We therefore have to slightly adapt the class *javax.swing.JFileChooser* by adding one special file filter for each known format specification. Each format specification shall consist of the default extension, a short description and the MIME type of the format. We introduce a new interface for this purpose, called *FormatSpecification*, that gathers these elements.
The abstract *Importer* class now has to offer only a small selection of methods. First of all, it must be able to store references to known concrete importers. These elements, which can be regarded as instances of the *Prototype* pattern [61, p. 117ff], have to be retrievable by either their MIME type or the associated file extension. In addition, the importer shall be able to provide a list of all format names it can handle, as there may be instances where one filter handles multiple input types. For example, the *gv* application on UNIX-based systems such as Linux is able to handle (at least) *Postscript (.ps)*, *Encapsulated Postscript (.eps)*, *compressed Postscript (.ps.gz)* and *compressed Encapsulated Postscript (.eps.gz)*. Instead of forcing developers to write four separate wrapper classes, an implementor of this functionality shall be able to act as a filter for all four types.
Figure 4.5 illustrates the structure of the import layer in the framework. Apart from an initialization

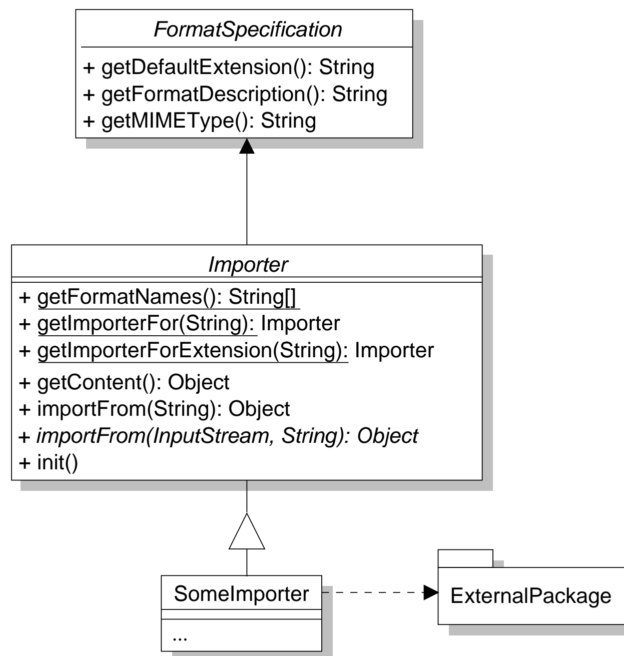

Figure 4.5: Import Layer Structure in the Framework

method that may be needed by some formats, the main work is performed within two methods. The *importFrom(String)* method tries to open an input stream on the String value passed in and invokes the abstract *importFrom(java.io.InputStream, String)* method. Here, the first parameter is a file stream which may be used for reading. The second parameter can be a filename or the actual content, depending on the input type. The result of the import operation is returned by the methods, but can also be queried later with the *getContent* method. Concrete subclasses of the abstract import framework class may add additional methods or even delegate parts of the import process to other packages.

Figure 4.6 illustrates the structure of the export layer in the framework. Except for exchanging *import* with *export*, the export layer structure of the framework looks much as the import layer. The two methods for importing content from a String or an InputStream and a String have been merged into the method *exportTo(String)* which returns a boolean value. Concrete subclasses have to open the appropriate stream or writer on the desired output and return a boolean value indicating success or failure.

The reason for not passing in a Stream to write on is that the different types of conceivable output are too different to subsume them under a stream class. For example, the output might be piped, or result in a set of files. The latter is for example the case if the chosen export format is an image format such as *BMP*, which may require the generation of multiple output files if dynamic content is to be stored. In this case, a sequence of files sharing the same basic output name must be generated. Having only a reference to one preopened stream instance is insufficient for this application case.

The FormatSpecification class covering the format name, default extension and MIME type, allows incorporating all import formats as special file filters for a customized *javax.swing.JFileChooser*. For example, if an importer for the MIME type *text/plain* is chosen, only files with the extension

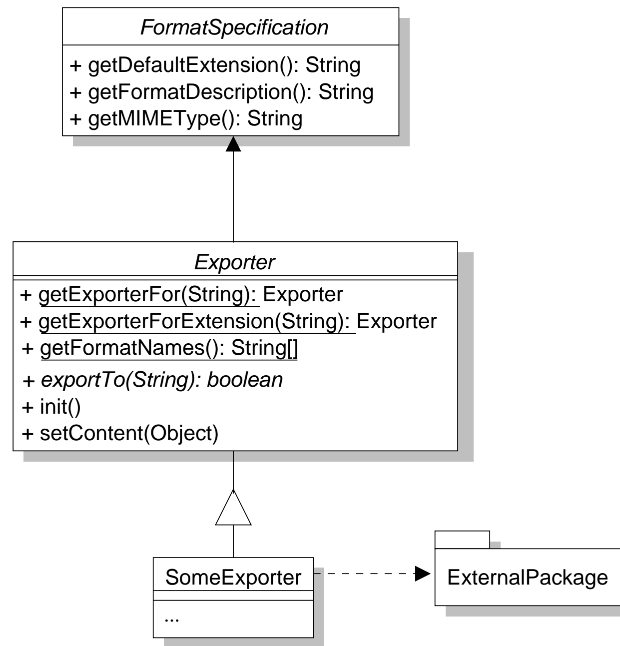

Figure 4.6: Export Layer Structure in the Framework

`txt` are shown in the file chooser.
At the same time, multiple different filters may have to use the same extension. For example, this may be the case if multiple different XML formats have to be parsed separately. Instead of hard-coding the association between an extension and a concrete filter, an implementing system may retrieve the class names associated with a given MIME type from a configuration file and use reflection to dynamically load in the associated class. This class can then act as a *Prototype* [61, p. 127ff] for all later instantiations of the particular format handler. Thus, the number and types of supported import filters can be configured at least before starting the system by adding or dropping entries of the configuration file. The same applies to the export layer.
Note that implementing systems or concrete frameworks will have to adapt the method interface by changing the return type of *getContent* and the *importFrom* methods, as well as the concrete *Importer* subtype to return from the *getImporterFor* methods, and accordingly for the export support.

4.2.3 Internationalization and GUI Creation Support

Two further related issues provide very interesting avenues for flexible and adaptive framework design: GUI component creation and internationalization in the GUI front-end. Two of the requirements placed in chapter 3 concern internationalization: **UR18** demands support for translating the content and the user front-end, and **VR4** for the generation front-end.
Java is in principle prepared for handling GUI component internationalization issues. However, the developer has to do all the work, as the extent of support in Java is limited to the inclusion of several classes in the Java API. *java.util.ResourceBundle* and its subclasses bundle a set of resources as pairs of key and value. *java.util.Locale* models the different locales on the earth, represented by their country code, language and possibly also application- or vendor-specific variants.

java.util.MessageFormat, *java.util.DateFormat* and *java.util.NumberFormat* provide different format options for entries depending on the chosen locale. For example, the notation for Christmas Eve in Germany is *24. Dezember 2002*, while the US-English notation is *December 24, 2002*.
The first and easiest application of these classes is to use a set of *ResourceBundle* elements for storing one language version each of all translatable messages. The current locale can then be used to determine the bundle from which a concrete message has to be retrieved. The disadvantage of this approach is that if a requested message is not present in the bundle, a *java.util.MissingResourceException* runtime exception is thrown. The resource bundle class also offers the retrieval of all possible keys in the form of an enumeration object.
The question we have to decide now is how a general framework can provide the best possible support for developers. On the one hand, we need support for internationalization in the GUI front-end, and on the other hand, provisions have to be taken for supporting content translation. The basic classes provided by Java support addressing these issues. However, most of the work is still left to the developer, as there is no systematic support. The example in [80] illustrates this problem, including much code for performing the translation to a single additional language.
From a design point of view, it would be best if we had two main participants for solving this problem. One participant should be able to translate a given message, possibly after modifying the message text by including current values. This is important for messages such as "Could not load *xxx*: file does not exist". Here, we have to be able to use one message as the template from which runtime messages containing a concrete replacement of the placeholder value *xxx* can be generated on demand. Note that the developer cannot anticipate the concrete replacement value.
The second participant shall provide support for the easy generation of Swing-based GUI components. Currently, the standard construction of many Swing components is cumbersome. For example, generating a *javax.swing.JButton* with all features requires four method invocations. The button is first generated together with its *label* and *image icon*. The *tool tip text, mnemonic* and *action listener* have to be added separately in one method invocation each. This process is always the same for buttons with these attributes.
When we try to provide internationalization for the button, the number of method invocations increases even further, as we have to update the *label* and *tool tip text*, and possibly also the *mnemonic* and *icon*. We therefore need a class that makes the generation of common GUI components including internationalization support as simple as possible - ideally, each component shall be generated with a single method invocation. This class requires the support class for text translation to be able to update the text components after internationalization.

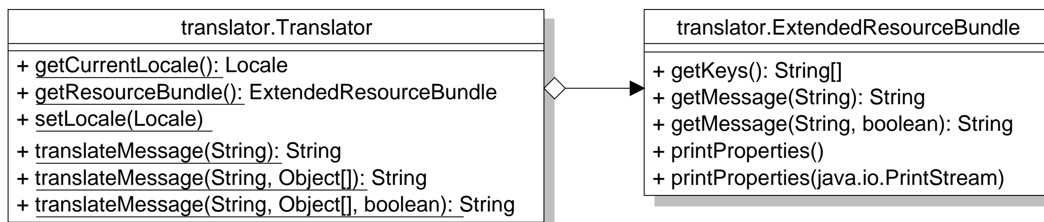

Figure 4.7: Class Structure of the *Translator* Class

Figure 4.7 shows the framework classes we provide to solve the first problem of translating messages. We have decided to provided an extended version of resource bundle support in a new

class, called *ExtendedResourceBundle*. As the purpose of this class is the translation of content, we have placed it into the new package *translator*. *ExtendedResourceBundle* offers accessing all keys present in a resource bundle as a String array and catches the possible runtime exception when translating a given message. An optional boolean parameter generates a warning if the requested message does not exist. Of course, this warning is also translated into the language determined by the current locale. To test the completeness of a given language resource bundle, two further methods allow printing the contents to *System.out* or a given *java.io.PrintStream* object.
Based on the *ExtendedResourceBundle* class, we can specify the behavior for a class that offers translatable messages. We call this class simply *Translator* and also place it in the *translator* package. The class shall act as a *Singleton* [61, p. 1287ff] by providing only class methods instead of instance-specific methods. The class can therefore be used from everywhere without having to maintain references to concrete instances. The class must allow querying and setting the current locale. Additionally, it must access the underlying *ExtendedResourceBundle* to retrieve the translated content.
The translation process is performed within three methods sharing the same name *translateMessage*. The *String* parameter passed in is the *ID* of the message to translate, not its concrete content. The optional *Object[]* parameter represents arguments to be included in the translated content. In the motivating example above, this is the concrete file name for the placeholder *xxx*. Finally, the boolean parameter determines whether a warning shall be generated if the translation process fails. This is typically the case if the message ID is not present in the keys of the current resource bundle. The first two *translateMessage* methods always map into the third method by providing default values for the missing arguments. The translation process is performed by requesting the target message from the *ExtendedResourceBundle*. If the result is not null, the message format for the current locale is applied to the returned message, incorporating the parameter values stored in the Object array. As defined in class *java.util.MessageFormat*, the process replaces all occurrences of numbers placed in curly braces by the object at the given position in the parameter array. Thus, the correct representation of the message "Could not load *xxx*: file does not exist" is "Could not load {0}: file does not exist". The developer only has to make sure that the array passed contains (at least) one element, namely the file name at index 0.
The *setLocale* method of the Translator class switches the current locale. Apart from storing the current locale within the Translator, the ExtendedResourceBundle is also requested to locate the associated language resource bundle. Each new interaction with the Translator class will now provide texts translated into the chosen language, provided that the target resource exists. The two classes shown in Figure 4.7 therefore address the problem of translating message texts well.
Based on the classes Translator and ExtendedResourceBundle, we can now design the more challenging class for supporting translatable GUI elements. To allow easy reuse, the method interface of the class shall be kept as simple as possible. Additionally, one method invocation shall be enough for generation of a new GUI element. Finally, a single method invocation shall cause the translation of *all* elements generated within the class.
To achieve this goal, we propose a class *TranslatableGUIElement*. Most of the class methods allow the generation of translatable GUI elements. To make sure that all components are translatable, they are registered in an internal hash table. When a translation of the components is requested, an iterator translates each registered component according to its type. For example, a *javax.swing.JLabel* can only have a text entry, while *javax.swing.JButton* instances may have to translate their text, tool tip entry, mnemonic and possibly also the icon shown.
Table 4.1 lists the capabilities of the methods supported by the *TranslatableGUIElement* class. We

refrain from including the UML diagram as this is too large to fit on the page due to a cascaded method interface that allows dropping several of the generation parameters. Note that some methods can also be customized further with parameters. For example, the programmer can provide a minimum, maximum and start value for *JSlider* elements.

Type	Translatable Components
translator.ExtendedAction	label, icon name, tool tip, name of method to invoke
translator.ExtendedActionButton	label, tool tip, mnemonic
javax.swing.JButton	label, icon name, tool tip, mnemonic
javax.swing.JCheckBox	label, icon name, tool tip, mnemonic
javax.swing.JComboBox	tool tip
javax.swing.JLabel	text
javax.swing.JList	tool tip
javax.swing.JMenu	label, tool tip, mnemonic
javax.swing.JMenuItem	label, tool tip, mnemonic
javax.swing.JRadioButton	label, icon name, tool tip, mnemonic
javax.swing.JSlider	tool tip
javax.swing.JTextField	tool tip, default text

Table 4.1: Internationalization Support for GUI Elements in *translator.TranslatableGUIElement*

The table includes two special classes we provide: *ExtendedAction* and *ExtendedActionButton*. Based on the *javax.swing.Action* class [80], these provide translatable buttons with built-in action listeners. The comparatively simple approach followed by the *Action* elements causes a direct invocation of the Action instance's *actionPerformed* method, which then has to be adapted by the developer. In contrast, the new classes directly invoke a target method passed by name. Note that this method name is also parameterizable and may thus change if the locale has changed.
The classes in the *translator* package are easily reusable in other projects. They provide full-fledged support for translating messages and even GUI elements on demand and on the fly. For example, the developer could provide a (translatable) *Language* menu containing the (also translatable) language choices as *ExtendedAction* elements. If the user clicks on one of these elements, the *setLocale* method of class *Translator* is invoked to change the current locale. This will also call the *translateGUIElements* method in the TranslatableGUIElement class and cause an update of all GUI elements generated using the class. Thus, translating the whole interface of a given application requires only one mouse click to perform, and only a few lines of code to embed! Of course, the developer also has to make sure that the appropriate language resources are available, as the translation process will fail otherwise.

4.2.4 Component Assembly and Administration

Achieving maximum flexibility in an extensible framework requires minimizing the number of "hard-wired" extensible objects. Thus, a concrete system may contain only a small selection of "fixed" elements which are explicitly referenced in the code, and add other extension components at the user's discretion. The decision which components shall be loaded can be encoded in a

configuration file.

This configuration file can in principle be in straightforward line-based ASCII format, or be available as a *XML* or property resource file, or come in a variety of other formats. For easy modification by users, we use the ASCII notation. Replacing the format by another one is a simple matter of rewriting the parsing method that retrieves the entries stored in the file.

One possible concern with keeping a configuration file is that editing its content may render the system unstable by adding unwanted or removing needed components. However, developers also need the possibility to experiment with new extensions. To resolve this dilemma, the framework looks for the resource file first in the directory where the system was started. If the file is not present there, the entries of the *CLASSPATH* and finally the *distribution jar* file are searched.

Thus, developers can experiment with new features by simply starting the AV system in a different directory. Note that this requires no code duplication or copying of the *jar* file. Classes which are present in the *jar* file but not configured for use act as if they did not exist. One possible application of this fact could be assigning students to programming tasks such as "implement the following extension filter" by giving them access to the *jar* file with a configuration where the extension filter is excised.

Another application area of adaptive and extensible systems is the support for changing the configuration of the system at run-time. The framework therefore requires a special component that supports the insertion of new elements. An example screen shot of such a *ComponentConfigurer* is shown in Figure 4.8. The screen shot was taken from the prototypical implementation of the framework presented in the next chapter. It therefore represents a certain configuration. For our discussion of component assembly and configuration, we can ignore the actual values shown in Figure 4.8 and focus on the existence and degree of support of the presented component.

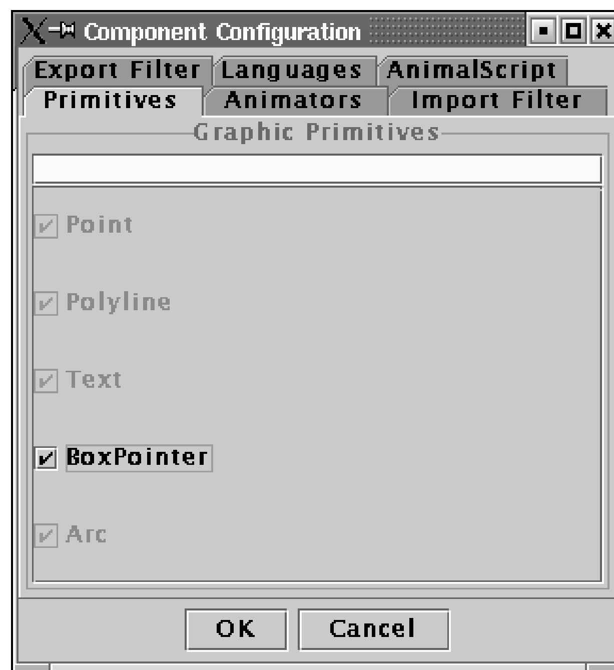

Figure 4.8: *ComponentConfigurer* Window for Adding Extensions on the Fly

Several entries are grayed out in each configuration tab. These are the *core components* without which the system may not run smoothly. They cannot be removed by changing the configuration dynamically. Other components can be removed by deselecting them in the *ComponentConfigurer*. In general, the framework should allow the dynamic addition and removal of all component types shown as tabs in Figure 4.8.
The general approach for adding a new component can always be essentially the same:

1. retrieve the class name of the class to add; for example from the text field in Figure 4.8 or as an entry in the configuration file;
2. if necessary, prepare the class name for component look-up by prepending the associated package name;
3. ask the default *java.lang.ClassLoader* object to locate the class definition for the given name by searching the entries of the CLASSPATH. If this does not work, cancel the addition;
4. try to instantiate the loaded class without performing any cast operation. The result is either an exception, in which case the addition process is canceled, or a new instance of the class;
5. test whether the object is of the expected type and conforms to the expected interfaces; if so, cast the reference accordingly and store it appropriately;
6. finally, add the object to the "appropriate" places in the running system.

The last operation obviously depends on the type of object loaded. For example, loading a new primitive class requires adding an appropriate entry to the tool bar for generating new primitive instances. Whenever a new instance of the loaded primitive is requested, the prototypical instance is cloned and returned. Removing a given component requires only the removal of the prototype from the storage.
The last enumeration item but one does not specify *how* and *where* the newly loaded object shall be stored. Similarly to the expressiveness gained by representing object state by properties, we advocate storing the instance in a hash table as a *Prototype* [61, p. 127ff]. This is the logical extensions of the application of hash tables for efficient look-up of dynamic values to dynamic *components*. It is inspired by the usage of hashing and properties in GUI assembly and Java beans. Using hash tables for storing each component as a Prototype consequently allows for maximum flexibility. Individual components can easily be added to the hash table and mapped to concrete instances by cloning the Prototypes. For this end, the hash key of each component must either be known or located in the enumeration of all keys stored in the hash table. Removing a given component from the system is achieved by simply deleting its entry from the hash table.

4.2.5 Decoupling Associated Objects Using Handlers

Handlers are used for decoupling components that have a different focus, where one component offers services which can be selected and requested by the other component. For example, think of a graphical object and a transformation on it. The graphical object, for example a point, does not need to be aware *why* a change in location is performed, as long as it properly adjusts its location.
Handlers provide the communication between these two separate elements. They offer a set of possible operations applicable to the first component to the second component. In this section, we

will call the component which offers services and updates its internal state to perform operations *ActualModel*. The second component which requests a list of possible operations and requests an update of the *ActualModel*'s state is called *Modifier*. Once a specific operation is chosen and the Modifier has determined the target state to assume, the handler invokes appropriate methods on the associated ActualModel that cause the desired effect.
Handler instances are always tied to a ActualModel *type* and therefore Singletons [61, p. 127ff]. They effectively decouple the ActualModel and Modifier instances. To provide dynamic extensibility without code modification, we also introduce *handler extensions*, which allow adding new capabilities to a system even at run-time. Handler extensions are stored in a dynamic look-up structure along with the type covered. For example, the class name of the covered type can be used as the key for storing the handler extension in a hash table. As there may be more than a single handler extension for a given type, they are stored in a vector. When users request an operation on a given type, the vector associated with the given type name is retrieved and iterated until the appropriate handler extensions has been located and executed.

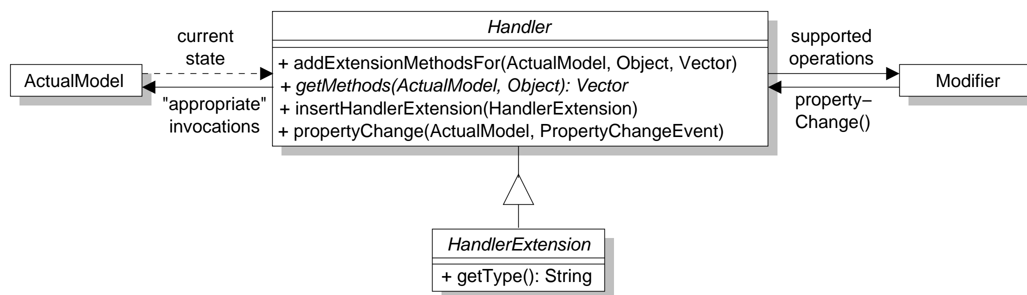

Figure 4.9: Abstract Schematic Handler Structure

Figure 4.9 shows an abstract view of the communicating parties in the handler concept. The *ActualModel* class in Figure 4.9 is the current point in our example. The *Modifier* role is taken on by the transformation on the point. The name of the handler has not been changed. As outlined in Figure 4.9, the handler provides the *Modifier* with a set of possible operations to perform on the *ActualModel* instance. This set of operations will often depend on the current state of the model, indicated by a dashed line.
Once the Modifier has decided on a concrete operation - for example, by querying the user for a choice -, it requests appropriate actions on the ActualModel by sending a *java.beans.PropertyChangeEvent* to the handler. This event encodes the name of the operation as well as the current and target value of the property to change, as explained in detail in section 4.6. The handler is then responsible for mapping the requested operation to the actual model, or possibly veto the change.
There are many possible areas where this concept can be of interest. Some examples include

Adaptive GUI components The *Modifier* represents the user who selects GUI operations. The *ActualModel* is the actually generated or modified GUI. Finally, the handler is the agent that offers a set of possible operations to the user and adjusts the actual GUI when needed.

Sandbox In this application, unwanted and possibly critical operations can be prevented from accessing the underlying object by being caught in the handler. The Modifier role in this example is the agent who requests operations, for example an application running inside the

sandbox. The ActualModel role is taken on by the object outside the sandbox, which only receives requests if the handler allows the operation.

Personalized Shops The Modifier is the Web interface of a given e-shop, offering different wares to a target audience. The ActualModel contains the wares offered by the shop, or alternatively the known information about the clients. The handler can then customize offers for clients based both on the *type* of client (preferred, occasional customer, first time, ...) and the concrete *attributes* of the client, such as the client ID. It can thus offer both general discounts for long-time clients, and special offers that are thought to be of interest to a specific client according to the information gathered so far.

Web Services Here, the requesting application takes the role of the Modifier. The Web Service is represented by the ActualModel. The handler as a go-between can provide a description of the features offered by the different Web Service types and instances available and negotiate priority access. For example, the service description can be coded in the Web Service Description Language (WSDL).

Consistency The Modifier role is assumed by the agent requesting an operation. The handler can check whether the operation is admissible based on the type *and* current state of the underlying ActualModel instance. If the operation is admissible, it may be forwarded. Alternatively, it can also be stored ("cached") locally and forwarded only after the Modifier sends a "commit" signal. In this case, the handler can also act as a caching transaction control or policy checker.

Filtering Proxy As a last example, the handler may act as a filtering proxy which shows only selected operations based on the value of the ActualModel. Here, the Modifier role is typically assumed by a user requesting an operation, and the ActualModel is a service or application. The filtering may be applied to the service type - for example, no FTP access - or based on the current state of the instance - for example, no access to pages containing sexual material. In this case, the filtering proxy can also serve as either a privacy guard or children filter.

The *handler* concept is one of the most important parts of the framework, underscoring the flexibility of the framework regarding adaptivity and extensibility. The concept is also far-reaching. However, it is difficult to classify handlers and handler extensions as design patterns. Each concrete handler and handler extension is implemented as a *Singleton* [61, p. 127], as they are tied to a specific primitive *type*, not instance. The method invocation sequencing of passing both the method name gathered by *getMethods* and the effect mapping within *propertyChange* is typical for the *Chain of Responsibility* pattern [61, p. 223ff].
In some respects, handlers also act as a variation of the *Bridge* pattern [61, p. 151], although the underlying problem is not one of bridging between an abstraction and an implementation as intended by the pattern. There are additional similarities to the general ideas underlying the *Adapter* [61, p. 139] pattern, although the focus here is not on inheritance, but rather dealing with code decoupling and support for unanticipated extensions.
The *Facade* pattern [61, p. 185] offers a unified interface to a set of interfaces in a subsystem. In a way, this is also what handlers do, as they offer a uniform way of determining and performing animation effects appropriate to the given primitive type. However, the match is again not quite accurate. There are also similarities between our handler concept and the *Mediator* pattern [61, p.

273], but also enough differences to show that they are not the same. The main difference here is the dynamic maintenance of the extensions which are not anticipated in the Mediator pattern.
As handlers and handler extensions are stored in a static hash table and looked up when needed, new extensions can easily be added or removed at run-time, without risking the stability of the system.

4.2.6 Summary of Framework Extensibility and Adaptivity Approaches

The goal of this section was the introduction of components for a highly flexible and dynamically extensible framework. We have placed special attention on specifying components in such a way that the implementation of extensions can be achieved without modifying existing system or framework code. The only exception to this rule are adaptations of the return types of the handler concept, which are necessary for supporting compile-time type checking and thus preventing casting and possible run-time exceptions.
In general, the approach we proposed in the section allows developers the implementation of extensions without deep system knowledge. Part of this is due to the fact that no existing code has to be modified. The dynamic configuration and addition or removal further helps in implementing extensions without having to read and analyze all existing classes in a given package.
In the following paragraphs, we briefly recap the key aspects of this section and outline strengths and weaknesses of the individual components.

- *Properties* offer a highly flexible approach for modeling object state. Basically, “fixed” attributes are replaced by a properties object that stores all values in a look-up structure. The standard pair of access and setting methods for individual “attributes” can still be provided, but now works on the associated properties entry.

  In general, properties can make the class interface leaner by not requiring the *get / set* method pair, and also provide a good central point of data encapsulation in each object. Developers can easily add or remove attributes even at run-time by inserting or removing a new key and associated value into the properties look-up structure. The support for default values enables consistent behavior even for incomplete data.

  As any other approach, properties also have disadvantages. First, the developer cannot see what “attributes” are covered as easily as with attributes. Note that from a data encapsulation and information hiding point of view, this is actually an advantage! Additionally, retrieving elements from the properties always returns a *String* object which has to be transformed in the appropriate type. The final disadvantage or liability is that extracting elements from properties, even using our support class *XProperties*, does not offer real type checking as would be the case with declared attributes. In the interest of extensibility and in view of the gained flexibility, however, we think that this disadvantage is acceptable, as we also make sure to prevent run-time errors due to incorrect casting.

  The main advantages of using properties are the great amount of additional freedom gained for dynamic ad-hoc addition or removal of elements and attributes thereof. A special support class specified in the framework addresses the disadvantage of having to transform entries from String to the desired type and vice versa by offering this functionality. Additionally, the base properties-based class in our framework supports flexible cloning of the properties values for new instances based on the Prototype pattern [61, p. 127ff].

- Content *import and export* can typically be achieved in three different ways. First, the code for reading and writing a given object can be embedded in the class code. Second, the Visitor pattern [61, p. 331ff] can be used for supporting diverse formats. Finally, the import and export code can be placed in separate classes or even packages.

  In the context of a dynamically extensible system, the first two choices are practically unusable. Embedded import and export code in the class results in continuously growing class size, with the actual modeled content representing a shrinking percentage of the full class code. Additionally, each newly introduced import or export filter requires modifications to the implementation code of all classes.

  Similarly, the introduction of a new element on a Visitor-based import and export layer is problematic for dynamically extensible systems. Each addition requires a new method to be declared and implemented in the Visitor and all subclasses - a direct violation of our requirement of code modification-free extensions.

  Placing the import and export code in a separate class or package thereof is the safest approach for dynamically extensible systems. Based on the *Importer* and *Exporter* classes defined by the framework, new filters can easily be implemented. The *FormatSpecification* interface defines the properties of the filter, essentially its default extension, a short description and the MIME type. Both extension and MIME type can be used for customizing the *JFileChooser* class used for file selection in Java. Additionally, the correct filter can be looked up based on the extension or MIME type. The import and export support is thus very flexible and allows for easy reuse.

- The *internationalization* components provided by the framework extend the translation services supported by Java. Base Java offers a set of classes that can be used for translating content. However, the developer has to implement the actual translation process manually. For this end, the framework extends the underlying modeling object for translatable texts in the *ExtendedResourceBundle* class.

  Content translation is achieved by a single method invocation in the *Translator* class. The method parameter is the ID of the message to translate. Additional parameters to be inserted into the message can also be passed in.

  Finally, the *TranslatableGUIElement* class provides internationalization support for Swing-based GUI elements. A single method invocation is sufficient for generating supported Swing components, in some cases even along with action listeners. The translation support covers displayed texts, tool tips, mnemonics, and where applicable, also the names of icons or messages to invoke on activation.

- The *dynamic assembly* of components used in the framework is based on a configuration file on the one hand, and dynamic configuration in a special window on the other hand. The configuration file can have different formats; for pragmatic reasons, we have decided to use a straight line-based ASCII notation. To allow reusing the distribution *jar* file in a variety of configurations, the framework looks for the configuration file first in the directory where Java was started, followed by the entries of the CLASSPATH and finally the jar file itself.

  There can be many different configurations of the same system, based on the directory where the configuration file resides on the one hand, and on the dynamic configuration on the other

hand. This allows developers to experiment with new features, as well as assigning programming tasks to learners by simply "removing" the implemented component in question from the configuration.

As described in section 4.2.4, adding a new component is very simple, stressing the extensibility of the system. All possible run-time exceptions due to dynamic loading are caught and lead to the removal of the component in question. Thus, the only run-time exceptions and errors possible are those caused by the code, *not* by the dynamic assembly. The same errors and exceptions would therefore occur if the components were "hard-wired" into the framework or concrete system.

The performance of the system also does not suffer measurably from the dynamic assembly, as cloning the requested elements is easy and efficient to achieve. The very slight overhead in running time incurred by the dynamic nature of the framework is easily acceptable when one considers the added expressiveness regarding dynamic assembly and extension implementation.

- The *handler* concept decouples two components that depend on each other for performing certain services. In the scenario, component A provides some services to component B. B can select the target service to execute and request it from A. The handler interacts between these two classes by providing the set of applicable services to B, based on the type of service requested and the current state of A. Once a service is selected, the handler executes the appropriate method invocations on component A.

  Handler extensions can be provided easily by implementing a new class and adding it dynamically. The extensions are examined whenever the standard extension for a given type cannot handle a request.

  Handlers are usable in many different application areas. They improve reusability due to the strict separation of concerns they provide. Implementing individual component extensions is also easier due to the cleaner focus of each component.

Combining the flexibility of the dynamic loading used for most central components in the ANIMAL framework with the power of the handler therefore yields a highly expressive, reusable and easily extensible framework. In fact, apart from the software engineering approach, this is to our knowledge the first published general-purpose framework for AV systems, as well as the first AV system that is explicitly geared for extensibility without intimate system knowledge, and more importantly, also requires no code modification.

4.3 A Brief Framework Overview

To specify an AV framework, we first have to define the participating entities. We regard both algorithm visualizations and animations, in the following abbreviated as "animations". It is easy to see that there must be at least two central components: *graphical objects* such as text or lines, and *animation effects* that modify the appearance of the objects over time. We will therefore adapt these two components for the framework.
Assuming that a visualization cannot be stopped or even rewound, these two components are sufficient. We could even skip the separation between animation effects and graphical objects and

replace the effects by explicit operations on the graphical objects. This approach is clearly insufficient if we plan to develop a general-purpose animation system that has no advance knowledge of the contents to present. Furthermore, an essentially stateless animation system built from cascaded or sequenced invocations of animation effects is too inflexible to allow advanced features such as animation rewinding. However, this facility is required for a deeper understanding of the animation content, as stated in **UR5** and several research papers, for example [118, 39, 55, 2, 21, 113, 134, 128].

We therefore have to incorporate a model of the animation content into our framework that represents the current animation execution state and also allows accessing previous or future animation states. To avoid the problem of having inadequate settings for memory used for undoing executed steps, we strive to find a solution that allows unlimited undo without excessive memory requirements for "large" animations. However, efficient rewinding represents a formidable challenge in AV systems.

If a given object transformation can only be modeled within either the graphical primitive it changes or the animation effect, developers will always be forced to touch existing (but possibly foreign) code when they prepare an extension. However, we do not want to force developers to read and understand the whole code of the system they want to modify. We are even less interested in forcing them to modify this code by finding just the right place to plug in their added functionality and basically hope that no side-effects will occur!

The framework shall therefore plug handlers between animation effects and graphical objects, as described in section 4.2.5. The responsibility of the handler will be to negotiate the services animation effects request from graphical objects by mapping them into a sequence of "appropriate" method invocations in the graphical object. Section 4.6 examines the adaptation of the handlers to the AV context in detail and outlines the advantages offered by the approach.

In addition, the ANIMAL-FARM framework offers a simple, efficient and elegant solution to the problem of efficient rewinding and context-free reverse playing, which was stated to be "one of the most important "open questions" in AV" as late as 2001 [2]. Briefly, forward playing is performed by initializing each primitive to the state to assume at the start of the current animation step. The primitive is then successively transformed by the animator (acting via the associated handler), based on the start time and the percentage stage of animation effect completion. For reverse playing, we modify the current time to lie in the future *after* initializing the animator. Instead of going forward in time, we step backwards until we reach the original start time, which marks the start of the step and therefore the end of the reverse execution. Accordingly, the animator requests modifications of the primitive that represent a move backwards in time.

The animation model contains three participating classes. The animation structure is encoded in an *Animation* instance. Apart from fall-back copies during animation import or export, this class acts as a *Singleton* [61, p. 217ff]. The Animation instance stores references to all primitives and animators, as well as the animation structure segmented in individual animation steps. The links connecting subsequent animation steps are modeled in *Link* objects. Apart from a link to the next step, Link instances also specify the operation to perform for advancing a step, for example a timed delay or waiting for a specific user interaction. Links may also be labeled, with each label acting as hyperlinks to the associated animation step. Finally, the current dynamic state of the animation is encoded in an *AnimationState* object.

Now the framework has four conceptual components: graphical objects, animation effects, handlers that act as negotiators between the graphical objects and the effects, and an animation model. Two more key components are obviously missing: facilities for displaying the animation and for editing

its contents. The animation can be displayed by requesting each visible graphical object to paint itself on a canvas for each state of the algorithm. Consecutive states therefore have to incorporate the changes or fragments thereof requested by the animation effects, for example by moving or recoloring a given graphical object. The editing GUI front-end shall allow the editing of both graphical primitives, for example by modifying their initial color, and the animation effects, for example by introducing different timing specifications or changing the type of effect to perform. As this functionality is currently not supported in most AV systems and might not be wanted, we provide only an interface for docking editing components. The interface may be left dangling without risking the stability of the framework.
In addition, the framework shall provide support for importing and exporting the animation. We therefore introduce two additional components for animation import and export. Several of the framework entities represent areas for extension ("hotspots") by developers. This concerns animation effects, handlers and graphical objects, as well as import and export facilities. Other areas, such as the GUI components, can be configured and extended to a lesser degree, as several of the requirements place implicit constraints on the design. For example, the control elements to support in the *AVPlayerGUI* component are basically fixed by requirement **UR5** and **UR12**.
Figure 4.10 on the next page provides a high-level overview of the central components of the ANIMAL-FARM framework: *import* and *export layer*, a drawing and editing front-end, and the central animation representation consisting of *primitives, animation effect* and *handlers* that negotiate between the two other class types. The optional editing framework component is included in the figure for completeness. As several AV systems may not want such a feature, we have not included it as mandatory in the framework. However, the *EditableObject* class has prepared the needed functionality for adding editing using direct manipulation.
The design of the framework places several research challenges. Apart from the general difficulty of defining a reusable framework, we also have to support dynamic extensibility without code modification. Users and visualizers shall be able to add and remove extension components at run-time using a configuration window. This goal has far-reaching consequences on the design approach taken, as it concerns most of the framework components. On the AV side, we have to support efficient rewinding and reverse playing on a scale hitherto not shown in general-purpose AV systems. Additionally, flexible and extensible import and export filters have to be introduced including the support for magnification both in the display and in appropriate storage formats.

4.4 Graphical Objects

The supported graphical objects are one of the central components of AV systems. They model the different types of entities that can be used for displaying the actual animation content, and thus have a large impact on the system's appearance and expressiveness.
The following issues have to be taken into account when designing the support for graphical objects:

Flexible The graphical objects should strive not to fix any context or place specific demands on their environment.

Widely applicable As demanded by requirement **GR10**, the application area and concrete topic used in a special instance of the framework shall be left open. Restrictions preventing reuse or fixing the context of usage to a certain topic area limit the applicability of the framework and therefore make it less attractive to use.

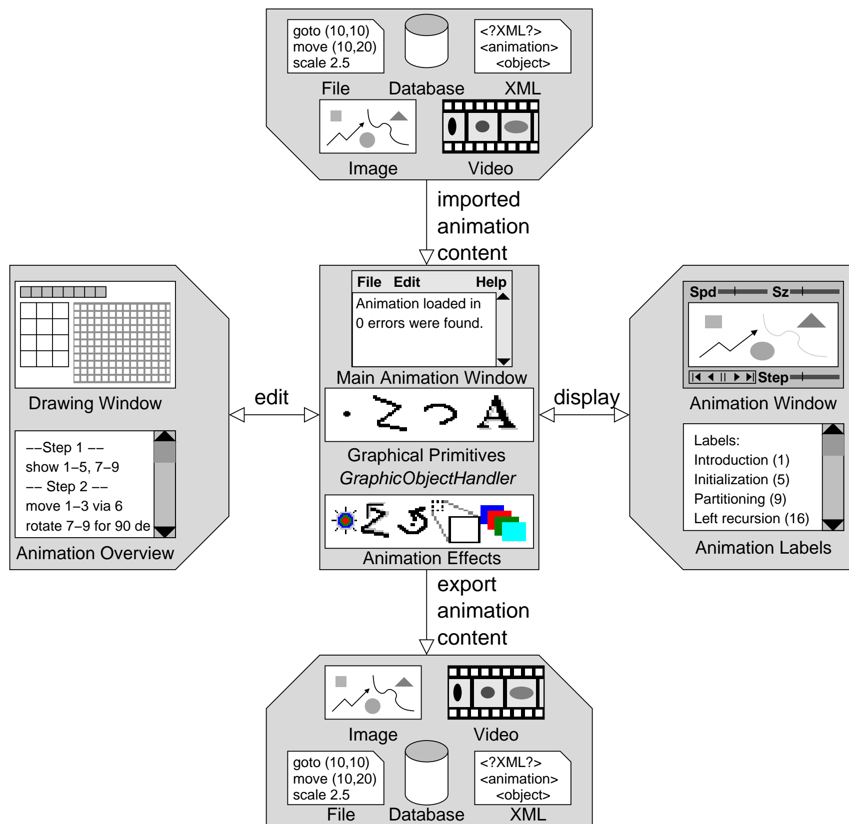

Figure 4.10: Abstract Overview of the ANIMAL-FARM Framework Components

Extensible Developers must be able to develop new graphical objects with ease. The concrete implementation of the framework shall be able to embed new objects even while it is running, without requiring a complete recompilation. Ideally, no source code of available classes shall have to be changed.

Easy to implement additions The developer shall find implementing new graphical objects as easy to do as possible (see **DR1** and **DR3**). Thus, the knowledge of implementation details required for providing an implementation should be kept as small and shallow as possible. At the same time, adding new primitives must be easy for the visualizer (**VR2**).

Separation of concerns Both reuse and extension programming are hindered if each instance of a new graphical object has to address a large number of issues. To be usable for animation purposes, graphical objects must be able to change over time. Packing this evolution process into the graphical object implementation results in large classes and less likelihood of reuse. We therefore have to decide on an approach that allows for a strict separation of concerns between animation effects and graphical objects. This will also make it easier to provide extensions without touching much if any existing class code.

Configurable As demanded by requirement **VR3**, certain aspects of the graphical object shall be configurable. This concerns the graphical representation, for example the color used for the object's outline.

Designed for reuse Apart from a clear separation of concerns as discussed above, composing available graphical objects into a new composite object must be easy. The amount of coding to be performed should be as low as possible to allow for effective reuse.

In the remainder of this section, we will discuss the modeling of graphical objects used within the ANIMAL-FARM framework. The discussion will show how the different demands are incorporated into the design.

The first design decision concerns the amount of knowledge that a generic graphical object shall have. The central issue in designing the graphical object is defining which entities shall be responsible for animating a given graphical object. As stated in the framework overview in section 4.3, we separate the animation into three concerns: determining the state to assume for the current point in time, mapping this state into operations on the affected graphical objects, and finally rendering the (updated) graphical objects the display front-end. The graphical objects only have to provide appropriate methods for updating the state, such as changing the color. The main part of the actual work in animating the object is shifted to the other two participating entities: the animation effect on the one hand and the handler on the other hand. These are described in more detail in section 4.5 and 4.6, respectively.

This separation of concerns makes reusing the graphical objects much easier, as developers can focus on implementing a graphical object without worrying about how it can be animated. In essence, this is similar to the task assumed by the *programmer* AV role, who implements an algorithm without knowing or worrying about later animations.

To make the extension of animation effects independent from the supported graphical objects, we decouple the classes using handlers. The animation effect shall not have direct access to the graphical primitive, but only refer to it by its ID. For the ID, we propose to use numerical values generated by a self-incrementing Singleton [61, p. 87ff] method, so that each allocated ID will be unique. By

not referencing concrete object instances, the development and compilation of extensions for both areas are decoupled.

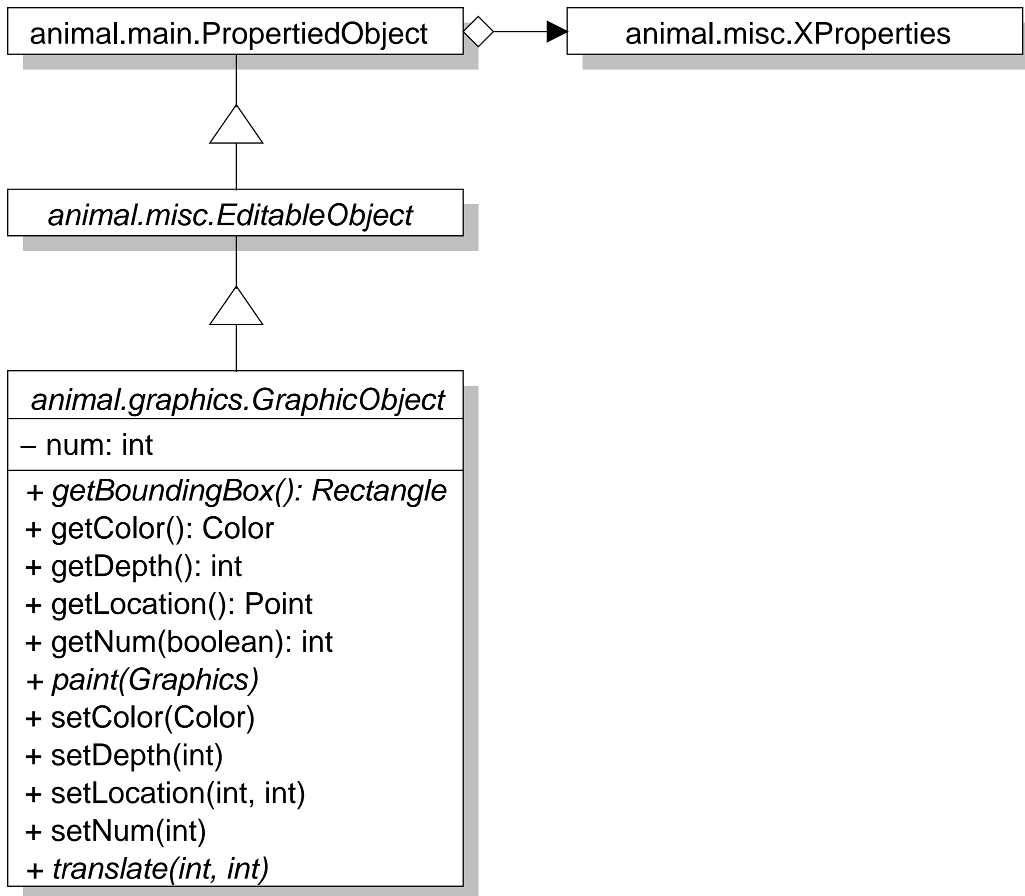

Figure 4.11: Architecture of Graphical Objects in the Framework

Offering the widest possible applicability of the available graphical objects means that we have to constrain the framework to offering primitive types. Primitives are simple conceptual graphical entities that can serve as building blocks for more complex entities. Typical primitives for building a graphical system are *points, polylines, texts* and *arcs*. The framework specifies the base class for the primitives. The degree of additional support for each primitive is left to the developers of a concrete implementation. For the rest of this chapter, we will therefore replace the somewhat imprecise term *graphical object* by *graphical primitive* or simply *primitive*.

Figure 4.11 shows the class structure of the proposed graphical object base class in our framework. As mentioned above, the graphical primitive has a private numeric ID field with a pair of methods for reading and modifying the value. When a graphical object is cloned, for example by a *copy and paste* operation in an editing GUI, the clone retains the same numeric ID as the original object. In some cases, this ID duplication may be wanted, as we will see in section 4.7. In all other cases, a new ID can be assigned explicitly by invoking the *setNum* method, or generated automatically by invoking *getNum(true)*.

Apart from a unique numeric ID, all generic primitives share some other characteristics: *color*,

depth, *location* and a *bounding box*, plus the standard methods for moving the element (also called translating) and displaying it using the standard Java *paint* method. Note that the interpretation of the *color* may differ between different primitives. In general, the most "basic" color should be used, for example by picking an object outline rather than its (optional) fill color.

The *depth* information is used for resolving the display of overlapping objects. The depth acts as a z coordinate in three-dimensional space, with the z axis vanishing into the monitor. Therefore, the greater the value of the depth information, the "deeper" the primitive. Primitives are painted from the background to the front in the reverse order of depth, so that primitives with a lower depth value may partially or fully overlap and thus occlude primitives with a higher depth value.

The *location* of a given primitive is represented by a point. By convention, it is the upper left corner of the object. The *bounding box* is the smallest rectangle that fully encloses the primitive. It is needed for determining if a given mouse click within the display or editing front-end could refer to the given primitive. It is also used for precise primitive placement based on the relation to other primitives ("relative placement").

The *translate* method moves the primitive to the target coordinate passed in. For some primitives, this can be achieved by simply adapting the primitive's location. In other cases, special adaptations may be necessary. Finally, the *paint* method uses the standard Java interface for rendering the current state of the primitive on the display, typically a drawing canvas. There is no adequate generic implementation for either of these methods. Therefore, they are declared abstract to force concrete primitives to provide an adequate implementation.

All primitives in the ANIMAL-FARM AV framework are subclasses of the abstract *GraphicObject* class, which itself is a subclass of *EditableObject*. This means that all primitives have default facilities for editing their state, and by default use properties for modeling their state, as shown in Figure 4.4 on page 74.

A short reflection shows that the primitive base class is not tied to a specific usage context, and certainly not restricted to AV applications. It is flexible enough to be applicable to diverse application areas. Also, the interface of the base class as shown in Figure 4.11 is so lean that it is very easy to implement extensions. The strict separation of concerns we propagate has removed all traces of animation purposes from the primitive implementation. In fact, the primitive could easily be used in any graphics context such as drawing programs or even games without having to change anything about its structure.

The appearance of the primitives can be configured by invoking their access methods, such as *setColor*. More importantly, the underlying properties also allow far-reaching configurability, as the insertion of new property values is as easy as modifying an existing entry. Of course, the *paint* method has to be modified to take the new properties into account when rendering the primitive.

In effect, the primitive base class *GraphicObject* acts as a *Template* pattern [61, p. 325ff]. Concrete primitives may also belong to the *Composite* pattern if they combine multiple primitives for added functionality [61, p. 163ff]. The design goals we placed at the start of this section - flexible, widely applicable, extensible, easy to implement, configurable and designed for reuse by using a strict separation of concerns - are fulfilled well by the chosen structure. The primitive base class, rooted as it is on editable propertied objects, can be reused in a large variety of different contexts. It thus represents a large but easy to use "hotspot" for future extensions.

4.5 Animation Effects

In section 4.4, we have seen that the graphical objects or "primitives" of the ANIMAL-FARM framework are designed to represent only a "static" view without any provision for animation. The framework therefore needs another set of classes for addressing the dynamical aspects of animations. For this purpose, we add two more types: *animation effects* ("animators") and *graphical object handlers* ("handlers"). In this section, we focus on the functionality desired from animators.
To avoid confusing the reader, we will first define some key terms used throughout the remainder of this thesis. An *animation effect* represents a family of related operations on generic objects. The implementation of a given animation effect is called *animator*.
Animation effect *subtypes* represent one concrete instance of animation effects. Finally, animation effects and effect subtypes may possess a timing specification consisting of an *offset* and a *duration*. The *offset* specifies the delay before the animation effect starts, while the duration specifies how many intermediate steps are used for generating a smooth display.
We use a short example to illustrate these distinctions. A typical animation effect is the *(generic) move* effect on selected primitives. A concrete animation subeffect could be *move the first node*, if "first node" is a valid concept for the selected primitives. A timed instance of this subeffect may specify both the offset – *after 100ms* – and the duration – *within 500ms* – of the underlying transformation.
Before we can start developing the structure of animator objects, we have to consider what we want to achieve. The design of the base animator class shall address the same requirements as for primitives. Thus, the animator must be *flexible* and *widely applicable* by not containing any special context, as achieved by a *design geared for reuse*. Developers shall find it *easy to implement extensions* and *configure* the available animators.
The separation of concerns underlying the concept of primitives can also be found in the animators. Here, we abstract in three different ways:

Decoupling from primitives The animator shall be fully decoupled from the primitives it works on, as it would otherwise have to import fixed class names. Dynamically loading of new primitives at run-time holds no advantage without this decoupling, as the primitives cannot be referred to. However, for performance reasons, we also do not want to perform a dynamic lookup and type cast for each primitive selected in a given animator.

Instead, the animator will simply work on the primitives' IDs. When necessary, the current state of the animation can be used for mapping this numeric ID to the actual underlying primitives. This way, the animator is unaware of the possible types of primitives it could work on, and therefore can easily animate primitives it knows nothing about. This especially includes primitive types that were not known at compile-time or even during system start-up.

Thus, the animator has to be defined just once and will then be able to handle arbitrary primitives, allowing for great flexibility in usage and animator reuse.

Animation effect timing All applicable animators shall possess a timing specification consisting of the offset and duration of their operation, as defined above. The timing information shall be specifiable both using real and virtual time units to address differences in processing power of the underlying hardware, as demanded by requirement **UR19**. As this may not be appropriate for all animators, we separate the animators in two base classes, with the second class adding the timing handling.

Genericity As stated above, each animation effect realized in an animator class shall represent a family of related effects. Thus, the animator encoding the *Move* animation effect shall be able to provide *all* possible applications of moving primitive or parts thereof. Obviously, the animator cannot anticipate all *move* types possible for primitives whose conception and implementation may still lie in the future. Therefore, the animator can only focus on representing the behavior of the animation effect over time, without actually modifying the underlying primitives. For example, the *Move* animation effect will focus on interpolating the point to reach at a given point in time, without trying to actually modify the selected primitives.

Separating the three aspects of decoupling, timing and genericity from the animator by shifting them outside of the animator class improves the chances for reuse of individual animators, and also makes implementing extensions much easier. However, we have to specify how animation effects are supposed to work. For example, how can a generic *ColorChanger* animator determine what color properties may be changed, if there is no direct access to the selected primitives? The answer is simple: the decision of which animation effect subtypes are applicable for a given primitive is shifted from the animator class to a handler, as introduced in section 4.2.5.
We decide to use a *handler* as a separate class for two reasons: first, the primitive as such is focused on its "static" appearance and therefore does not have to know the different ways in which this could be changed. Second, if the information were embedded in the primitive, providing a new animation effect would also require modifying the code of several primitives. This runs counter to our concept of code modification-free extension and reuse. Section 4.2.5 discusses the functionality offered by handlers and illustrates the benefits of this approach.
For our current discussion of animation effects, we will assume that we can request a list of all possible animation effect subtypes of the current animation effect for each affected primitive. The process of generating this list is described in section 4.6. The animator is then responsible for merging these subtype lists by building an intersection and presenting this to the visualizer for selecting the appropriate effect.
Finally, we have to define how animators can be executed to actually achieve animation in the visual front-end. We split this process into three segments: initialization, action, and finishing the execution. The *initialization* is responsible for setting up the animator. Timed animators that incorporate a duration, an offset or both must know the start time of the association execution to determine when their own execution can begin. Therefore, the initialization requires the current timing information, specified by real and virtual time. Additionally, the animator may depend on the current state of the animation, for example due to dynamic elements that have to adapt to the current situation. In some cases, the primitive will have to be accessed indirectly to retrieve the start value for the animator.
The *action* part of the animator performance is performed in a loop. During each loop iteration, the current time is determined. The animator then determines what its target state shall be. For example, a *ColorChanger* effect may determine an interpolated intermediate color based on the percentage stage of its execution and the start and target color. The type of the determined value depends on the animator: *ColorChanger* animators will use a *java.awt.Color* to represent the current state, while *Move* animators will use a *java.awt.Point* that represents the location to assume. Note that the calculation of this state is independent of the actual primitives on which the effect works. The resulting value of the calculation is passed along to the appropriate handler object, which is then responsible for mapping this into the appropriate operations on each primitive. See the discussion in section 4.6 for more details on handlers.

The *execute* operation lets the animator assume the final state of execution. It is reached when the animator is fully executed. It can also be invoked directly to get a static view of the end of the animation step instead of a dynamic display. Additionally, it can mark the animation effect as "finished" to allow the concrete AV system to skip it during the next loop iterations.

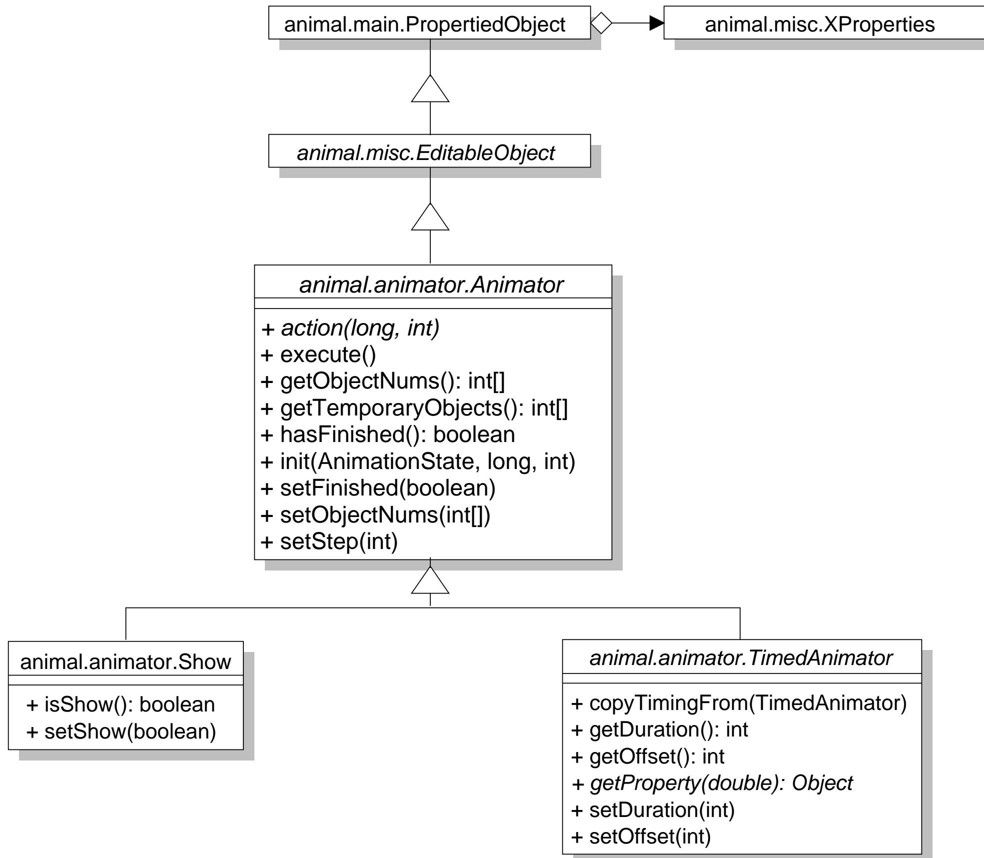

Figure 4.12: Architecture of the Framework Animation Effect Classes

Figure 4.12 shows the structure of the animators in the ANIMAL-FARM AV framework. The basic animator class *Animator* extends the *EditableObject* class presented in section 4.2.1 and thus prepares facilities for editing the properties. Apart from the already described methods *action, execute* and *init*, the animator can be assigned to a certain animation *step*. Steps represent administrative units for segmenting animations; they are described in more detail in section 4.7.
The *object nums* array stores the numeric identification of the primitives modified by the animator, while the *temporary objects* array contains primitives that steer the execution without being visible themselves. For example, moving requires a line or arc along which the primitives are moved. The numeric ID of this line or arc is stored in the temporary objects array. Finally, the *hasFinished* method allows determining if the animator has already finished its execution and can therefore be removed from the display update scheduler.
Two example subclasses of the animator base class are included in Figure 4.12. A *Show* animator, as shown, requires only a pair of additional methods for determining if the selected primitives shall be shown or hidden. The *TimedAnimator* class adds methods for assigning and retrieving the timing

information containing both the offset and duration. The *copyTimingFrom* method can be used to transfer timing information from one timed animator to another.

The abstract *getProperty(double)* method is responsible for calculating the concrete value to assume for the current point in time needed for smooth transitions **UR19**. The current time is passed as a double value and represents the execution state of the animator in the normalized interval `[0.0, 1.0]`. `0.0` refers to the start of the animator, while `1.0` represents the end of the animator execution. The returned value represents the target state to assume for the current point of execution; its type depends on the actual animation effect, as explained above.

Animators have two central hotspots: they can extend the *Animator* or the *TimedAnimator* class, depending on whether the animator shall possess a duration and offset. To implement an extension, the developer only has to implement the inherited abstract *action* and *getProperty* methods. In many cases, however, the developer may also want to overwrite the *execute* method that by default simply sets the animator state to "finished".

Note that animators only specify the animation effect subtype by name and possibly add timing information for the execution. They do *not* work directly on any primitive object. This is done by the handlers we examine in section 4.6. The primitives on which the animator works are present only by their numeric IDs. Therefore, implementing a new animator without also modifying or extending at least one handler instance will only produce results if the animator works without primitives, for example by linking to external documentation or tests (see **UR4** and **UR10**, respectively).

Implementing a new animator is used for introducing a completely new animation effect group, such as the family of all conceivable rotation types. To implement a new subeffect of an already available animator, such as *move nodes x y z* where *x, y, z* are node indexes, the developer instead has to implement a handler extension, as described in section 4.6. Animators have a very wide focus and are thus very well suited for reuse in many different contexts, as they do not actually perform operations that might clash with planned areas of use. Animators and TimedAnimators act as *Template* pattern instances [61, p. 325ff], just as the primitives themselves.

4.6 Graphical Object Handlers

We have already described the representation of primitives and animators in our algorithm visualization framework. As the reader will have noticed, there is no direct connection between primitives and animators. Instead of a direct link, we use a handler for this connection, as described in section 4.2.5 to further support reuse and extensibility.

Without the negotiating intermediate handler class, animators and primitives have to communicate directly. This means that the functionality the framework currently lacks would have to be placed in either or both of these classes. The tasks left are the determination of the animation effect subtypes available for a given animator and the mapping of a given animation effect to a modification of the primitive that reflects the animation effect intent.

If the determination of the offered animation subtypes for all primitives is placed in the primitive or the animator class, the introduction of a new animation subtype must always result in one of two alternative situations: touching the code of the class or introducing a new class that is only slightly different from the other one. Both solutions are inelegant, and the first solution is also highly error-prone due to possible bugs introduced in established class code.

To illustrate the problem, let us assume that there is a *Move* animator that can move whole objects, and a polyline graphical object that represents a sequence of linked line segments. The developer

now wants to introduce a new move effect subtype that only moves the first node of a polyline. We first regard the choice that the code for providing this new effect subtype is placed inside the *animator* code. Adding the appropriate code to the *Move* implementation may easily lead to bugs: the new effect subtype may only be used if *all* primitives selected for the Move animator are polyline (or possibly polygon) instances. If the developer forgets to place an appropriate conditional statement before the code, runtime errors due to invalid casts of for example text primitives to polyline instances may result.

Even if the conditional is added, consider how bloated the implementation code of the animator will become if the developer adds more subeffects. For example, moving multiple selected nodes of a polyline or changing the radius of an arc are very easy to conceive. The net result of just a few such additions will be a class code that becomes hopelessly tangled in casts and conditions, obscuring the clear and simple design of the animator structure.

Even worse, consider what happens if a new primitive is added and shall be provided with the "standard" effect. In this case, the developer has to touch *all* animator implementations to manually insert the code for handling the primitive! This is about as far removed from our intention of easy reuse and clear legibility as one can be.

Moving the generation of possible animation effects to the *primitive* does not improve the situation. The formerly clear and lean code of the primitive that was able to focus on representing the graphical properties of a "static" primitive is suddenly swamped with conditional statements. In this case, the code has to check which animator type requests the list of possible animation effects.

Developers of a new primitive have to embed the "appropriate" code for all currently available animators that shall be supported directly into the new class' code. Again, this reduces legibility, makes maintenance more difficult, and increases the likelihood of bugs or inconsistent treatments between primitives.

If a new animator is introduced instead, *all* primitive implementations have to be adapted. This may introduce bugs and side effects into previously well-tested and stable code. And of course, it also runs completely counter to our interest in reusability, legibility and simplicity.

The logical consequence of this dilemma is the introduction of a handler. In this context, the handler instance has three main tasks: listing the supported animation effect subtypes for a given primitive, mapping a concrete animation subtype at a specified state to operations on the underlying primitive, and maintaining its own extensions. Each handler instance is tied to one primitive *type* instead of a given primitive *instance*. It thus acts as a *Singleton* [61, p. 217ff]. Note that this also means that a concrete instance of the primitive has to be passed as a parameter for both determining the list of possible animation subeffects and actually modifying the underlying primitive.

Figure 4.13 on the next page shows the two participating classes *GraphicObjectHandler* and *GraphicObjectHandlerExtension*. They were adapted from the handler classes presented in Figure 4.9 on page 84 by replacing the classes *AbstractModel* and *Modifier* with the actual participating classes *GraphicObject* and *Animator*. As discussed in section 4.2.5, the handler extension class adds only a method that helps determining the primitive type to which it is associated. As stated above, the primitive instance to work on has to be passed to both methods, as the realization of handlers as Singletons provides no tie to concrete instances.

We will start our discussion of the handler usage with the first task. The scenario contains an animation effect which acts on a set of selected primitives and has to determine which animation effect subtypes are appropriate for this given set. To achieve this, the animator has to access the *getMethods* method in the handler associated with each of the selected primitive. Note that the decision which animation effects are possible may depend on the current state of the underlying

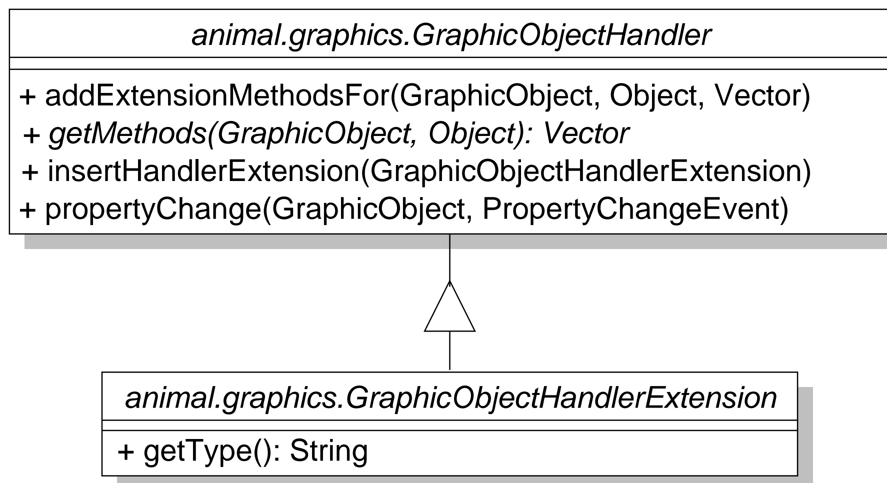

Figure 4.13: Handlers and Handler Extensions in the ANIMAL-FARM Framework

object. For example, changing the fill color of a polyline may be possible, but has no visible effect. The animator contacts the underlying primitive which determines its associated handler and invokes *getMethods*. This method receives two parameters: the current primitive for which the possible animation subeffects shall be determined, and an object that specifies the animation type. For example, a *java.awt.Point* object may represent both the underlying animator (*Move*) and the chosen target point to assume. In Figure 4.14, this second parameter is called *o*, as nothing more about the parameter is known *a priori*. Each invocation of *getMethods* returns a list of subeffect names. After gathering this list, the animator performs an intersection on the names to determine the set of animation subeffects that all selected primitives can perform.

In some cases, this set may be empty or contain only one element; in other cases, it may contain a large number of elements. The size of the result set is likely to become smaller the more different primitive types are involved, as each primitive adds effect subtypes specific to its type. For example, the option to move the second node is only relevant for primitives that are defined by more than one node, such as polylines, as opposed to text, point or arc components. The visualizer can select one of the elements in the result set as the target subeffect, or remove selected primitives to get a larger result set.

The second handler scenario deals with mapping a requested effect to appropriate operations on the underlying primitives that result in the wanted visual display. At each discrete step during the execution of the animation effect, the animator determines the current property of the effect. In our move effect example, this property represents the point along the move line to be reached “now”. As the method represents a change of one or more properties, we name the associated method *propertyChange*.

Each primitive then has to be updated to this new state. Again, this is performed by contacting the handler associated with the primitive type and asking it to invoke the appropriate methods on the primitive. From the primitive’s point of view, this is a “static” change, not a dynamic animation: there is no visible difference between handler-based and visualizer-based requests, as they might be issued when editing the properties of an existing object. Furthermore, as each given handler instance deals with exactly one primitive type, the mapping is typically very easy to perform. Figure

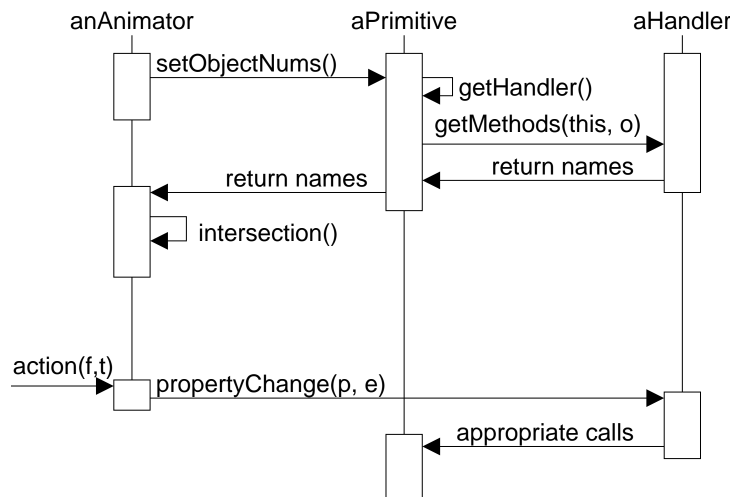

Figure 4.14: Cooperation between *Animator, Primitive* and *GraphicObjectHandler* Instances

4.14 illustrates the scenario.
Finally, the last handler scenario covers the maintenance of handler extensions. The additional indirection offered by the handler so far has improved reuse and the legibility of the code. Now let us assume that the capabilities of the handler itself shall be improved by adding a new animation subeffect for a given primitive. In this case, the developer will normally still have to modify existing code, with all the bad consequences this may have. While this modification is not as critical as the modifications previously undertaken in the primitive or animator class were, it is still unwanted.
The framework therefore adds a new class: *handler extensions*. The handler extension base class extends the standard handler class by adding a *type* information which can be retrieved by a *getType()* method. Extensions are added to the global repository of available handler instances by a special method. The type of the handler extension determines the association with a concrete primitive. At the end of the *getMethods* invocation, a special method is invoked that checks whether the repository of handler extensions offers additional subeffects. Figure 4.15 illustrates this process. Accordingly, the *propertyChange* method checks whether the repository contains a handler extension capable of handling a concrete request if the "standard" handler for the given type cannot handle it.
The *insertHandlerExtension* method adds the extension passed in to the repository. The *addExtensionMethodsFor* method checks the repository for extensions whose type information matches the GraphicObject instance passed in. It will then invoke *getMethods* for all matching extensions. All subeffect names resulting from these invocations are added to the *java.util.Vector* instance passed as the third argument. This vector is the same vector that was filled by the "basic" handlers within the *getMethods* invocation.
The net result of the operation matches the *Chain of Responsibility* pattern [61, p. 223ff]: invoking *getMethods* on a given handler will return the set of *all* appropriate subeffect names, whether they belong to the basic handler or any of the possible handler extensions. Note that the *Object* argument encodes the type and possibly further properties of the animator. For example, *Move* effects may use a *Point* object, while *ColorChanger* animators are likely to use a *Color* object.
Mapping the animation effect to the underlying primitive object works similarly. First, the handler

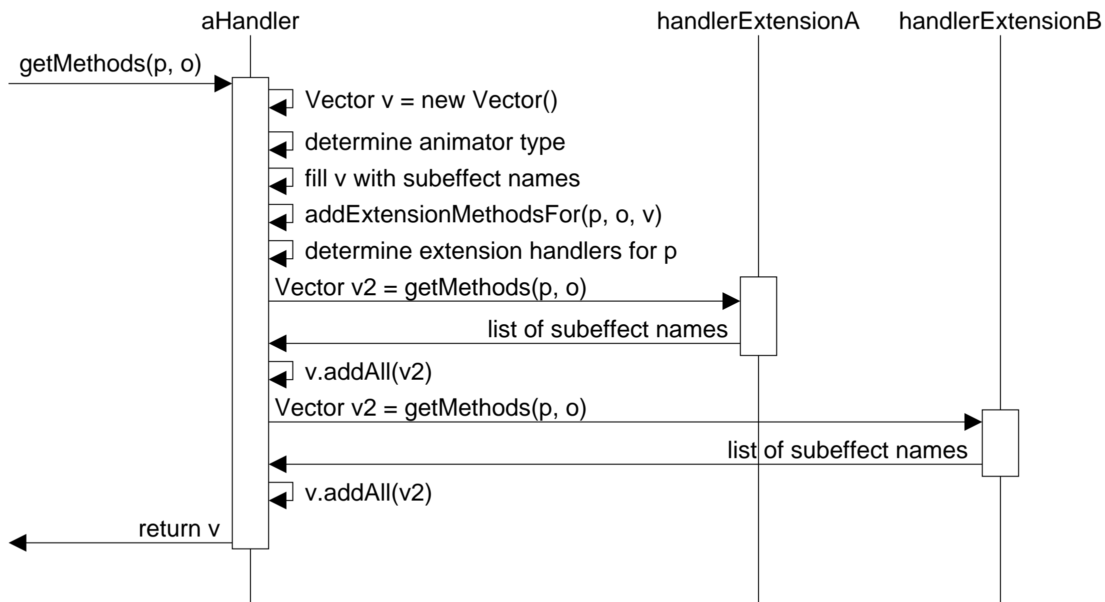

Figure 4.15: Incorporating Dynamically Added Handler Extensions

checks whether it is able to perform the requested concrete subeffect passed in as the "property name" in the *PropertyChangeEvent* argument. If so, the new value of the changed property is extracted from the event object and appropriate method invocations are performed on the primitive passed as the first argument. If the handler does not know how to treat the subeffect, the effect is passed along the Chain of Responsibility until it reaches the appropriate handler extension that is capable of handling the effect, or the end of the chain is reached and the effect is skipped.

4.7 Animation Representation

We have now described three of the central components of our framework: graphical objects ("primitives"), animation effects ("animators") and graphical object handlers and their extensions ("handlers"). We have illustrated how each of these components is structured and why they all support reuse and allow easy extensibility. The handlers act as a negotiating interface between primitives and animators and allow the separation of concerns for primitives which only have to provide a "static" representation of the object. Animators can focus on modeling one abstract animation effect family each, with both the responsibility for determining the concrete subeffect types for the selected elements and the mapping of the subeffect into appropriate actions deferred to the handler.

The structure presented is highly expressive, but also very flexible and context-free. However, the combination of the three class types only allows the modeling of a set of unlinked animation effects, as the structuring element that binds these isolated effects into a coherent whole has not yet been presented. In this section, we focus on defining an appropriate animation representation that performs this task.

The animation representation has to address some of the requirements discussed in chapter 3. Apart from providing an appropriate model for generic animation content, this especially concerns the

following requirements: labeling of steps (**UR12**) with the labels acting as hyperlinks, breaks between steps (**UR6**) and the possibility for skipping individual animation steps (**UR8**). In addition, the flexible animation controls required by **UR5** which also include reverse playing have to be kept in mind. While the animation representation is not responsible for actually displaying the animation - independent of direction -, it has to be prepared to address this issue.

Generally, animations can be structured into animation steps. Each animation step can contain an arbitrary number of animators which execute simultaneously or consecutively, depending on the specified timing. Requirement **UR6** demands the possibility to place breaks between consecutive steps. On the other hand, having a *mandatory* break between consecutive steps is disadvantageous, as this makes slide show-like presentations or subsequences thereof difficult if not impossible.

Therefore, the animation representation has to let the visualizer decide whether there shall be a break between two given consecutive steps. In general, there are two possible types of transitions between steps: a timed and an event-based transition. The timed transition starts displaying the next step after a specified delay has passed, with a zero delay resulting in immediate execution. Event-based transitions are usually triggered by user events, typically clicking on a "play / continue" button. However, other types of event-based transitions are also conceivable. For example, the next step may be shown only after the user has selected an arbitrary or a specific ("correct") element, performed a special operation, finished a test or answered an interactive prediction (**UR10**).

In addition to defining transitions between steps, the visualizer must also be able to provide labels for arbitrary animation steps. As stated in **UR12**, these labels can be gathered in a list and provide an overview of the animation structure. In addition they also act as hyperlinks: clicking on one of the labels in the list shall directly set the animation display to the associated step. Note that this requires facilities for moving backwards in the animation, as well as skipping animation steps (**UR8**).

Due to the different functions required from animation steps, we introduce a new class for modeling the animation steps. The new class may represent either the step itself, or the link between consecutive steps. The first approach is at first glance an obvious choice. Here, each step object contains the list of animators to be performed in the step, and also models the transition of the step. However, this modeling makes operations such as copying animators from one step to another or exchanging the order of steps more difficult. Note that the step itself is basically stateless: only the *transition* to the next step and the step label contain semantics.

We therefore model the transitions themselves in the new class, called *Link*. Links extend *EditableObject* and thus represent editable objects with built-in properties. Each Link instance is aware of its successor and the number of the animation step it represents. In addition, it stores the optional label belonging to the step where the transition starts.

A transition mode is used to describe how the user can reach the next animation step. As explained above, there are two default types of behavior: waiting for the user to activate a navigation button such as "play", or waiting for a specified amount of time. The transition mode is modeled as an integer value to allow for future extensions. The optional delay between steps as well as other conceivable transition information is placed in the properties.

We represent the link to the next animation step by an integer number instead of a Link reference. This makes merging two or more animations far easier, as we can easily provide a look-up table that translates the original animation step numbers to appropriately adapted values. Additionally, importing from and exporting to "flat" formats such as ASCII-based notations is easier if the next step to assume is encoded numerically.

Figure 4.16 shows a schematic structure of an animation illustrating the use of links. Each box

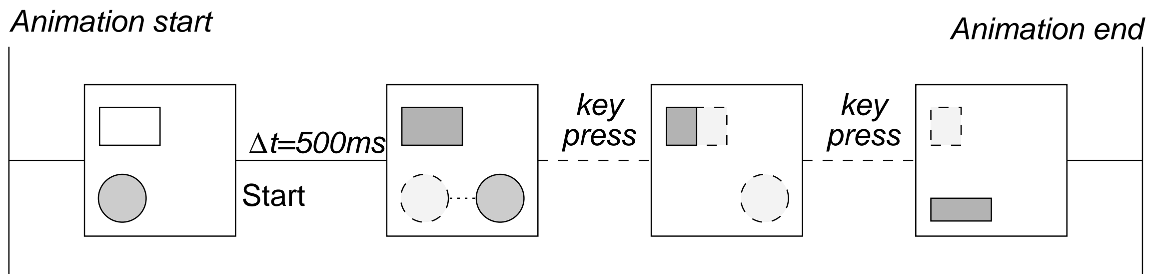

Figure 4.16: Example Animation Structure

represents an animation step, and the horizontal lines represent the step transitions encoded in Link instances. The example animation contains four steps with a total of seven animators on three primitives. The first step shows a rectangle and a filled circle. The step is labeled *Start*, as shown on the link connecting it to the next step. The transition between the first steps shall be performed automatically after a time of 500ms, unless the user explicitly halts or pauses the animation. The second animation steps changes the fill color of the rectangle and moves the circle. We use shading and dashes to illustrate the dynamic effects of the step. The third animation step is reached after the user has pressed a key. It moves two of the rectangle nodes, thereby shrinking the rectangle, and hides the circle. After another key press, the fourth animation step is reached, which hides the rectangle and introduces a new rectangle at the bottom.

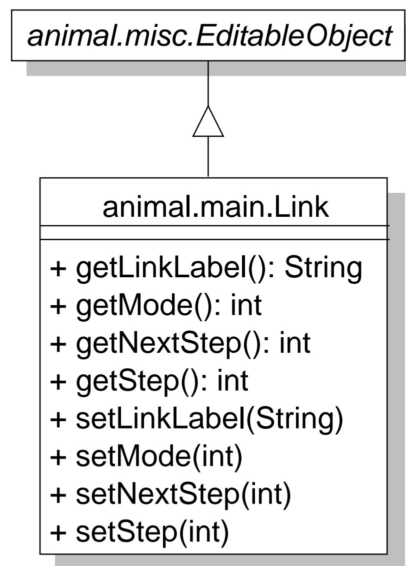

Figure 4.17: Structure of Link Objects for Modeling the Transitions between Animation Steps

Figure 4.17 shows the structure of the *Link* framework class. The modeling of the transition between steps provides support for step labeling as demanded by **UR12**. Note that further support is required in the animation display front-end to use the labels as hyperlinks; here, we can only

support their addition to links. Link objects also enable the visualizer to specify different transition types between steps (**UR6**).
We now have to define how we assemble the individual animation steps segmented by link objects into a coherent animation. We have to take into account that the representation shall offer support for skipping animation steps (**UR8**) and a variety of flexible animation controls including reverse playing (**UR5**).
For efficiency reasons and to support easier reuse of animators, links and primitives, we advocate placing all instances of the three types into one private vector each. Thus, the animation representation class shall contain three vectors. The main operations needed by the animation concern the insertion, deletion and retrieval of animators, primitives and links. Remember that animators cannot directly access individual primitives, as they only store their numeric ID. We therefore have to add a method for accessing a single primitive based on its numeric ID.
As the animation class as presented has a fixed connection to all animators, primitives and links, we need only one instance of it at run-time. Therefore, the animation class is basically an instance of the *Singleton* pattern [61, p. 127ff]. We say "basically", as there may be special cases when having two instances of the class for a few moments may be useful. For example, when importing a new animation, it makes sense to hold a reference to the current animation until the new animation has been successfully imported. In case of import problems, we can then easily revert to the previous animation; if the import was successful, we can discard the previous animation. Similarly, the generation of a subset of the animation as a new animation object may be helpful when exporting parts of the current animation.
The animation structure as presented so far represents only a static view of the animation, as it is completely stateless. We therefore introduce a second animation representation instance, called *AnimationState*. Animation states are responsible for representing a given state of animation execution. Contrary to animations, there may be multiple animation states at the same. To accurately represent the current state of the animation, the animation state object has to know the current animation step.
Similarly to its namesake, the *State* pattern [61, p. 305ff], the behavior of the primitives and animators depends on the current state of animation execution. The State object in the pattern and the AnimationState objects in the framework can be shared and co-exist without problems. The encapsulation of the dynamic aspects of the animation outside the main animation class makes it far easier to support advanced navigation controls that work in both directions, as a common fall-back point - the initial state of all primitives – is always preserved in the Animation Singleton.
To allow for easy navigation through the animation steps, as requested by requirements **UR5** and **UR8**, the animation state also has to be aware of the next and previous animation step, as well as the first and last animation step. Similarly to the Animation class, the animation state also holds one vector each for the set of current primitives and animators, as well as for the set of all primitives. The important aspect here is the word *current*: the vector of current animators holds only the animators to be executed in the current step. More importantly, the vector of current primitives contains *clones* of the primitives being animated in the current step.
The last aspect is central for the built-in support for reverse execution requested by requirement **UR5**. Most of the few AV systems that offer reverse execution to some degree do so by keeping an undo stack. The problem with this approach is that the developer of a system cannot anticipate how much memory has to be reserved for the undo stack. Note that the visualizer or user can also not specify the stack size, as this depends on the current animation and the user's interaction with it. Additionally, the undo stack fills very quickly if all intermediate animation states are also stored.

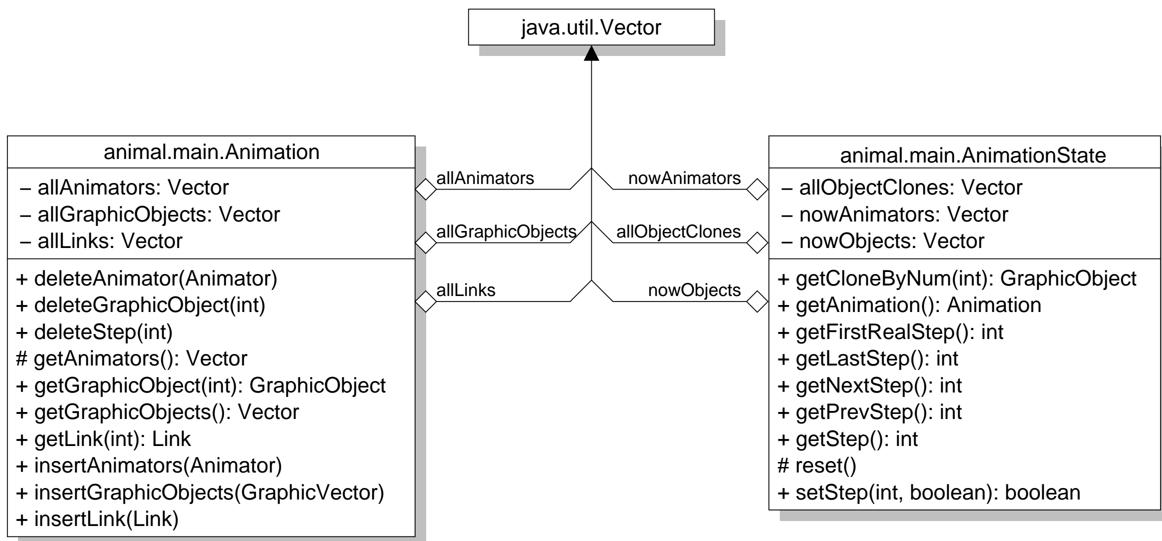

Figure 4.18: Animation Representation in the ANIMAL-FARM Framework

Pedagogically, there is a large difference between supporting *stepping back* (rewinding) and *playing in reverse* [2, 134, 181]. Stepping back simply returns the animation to the start of the previous step. Therefore, users have no visible clues to the effects performed within the step. Instead, they have to figure out the difference between the current and previous step by comparing the displays, which may easily lead to overlooking critical aspects. Playing in reverse, on the other hand, performs the same animation effect as playing the animation step in forward mode - only backwards. Thus, if an object moves over a certain time span to the right during the animation step, playing the step backwards will show the object moving to the left – and thus in reverse – over the same time span. Detecting the difference between successive steps is therefore much easier if playing in reverse is supported.
The described approach for setting the animation steps offers important functionality for two central didactic AV requirements: the ability to rewind the animation and even play it in reverse (see **UR5**), and the associated ability to skip animation steps (**UR8**). Note that the *Animation* and *AnimationState* classes do not provide support for dynamically performing the animation step. The AnimationState class only ensures the correct creation of primitives clones for the start or end of a given selected step. The actual dynamic execution of the animation effects that results in a smooth transitions on the display is only of interest in the AV player front-end, which we examine in the next section.

4.8 Animation Player Front-end

The ANIMAL-FARM algorithm visualization framework is not limited to specifying the components needed for assembling visualizations. Apart from the components already described - primitives, animators, handlers and the animation representation -, the framework also includes a prototypical definition of the animation display front-end. The reason for including this specification lies in

the fact that all concrete systems based on the framework shall be able to address a large selection of the requirements introduced in chapter 3. Accomplishing this is not trivial, and the developer may often find that decisions early in the implementation process block the way to achieving some goals. For example, flexible reverse playing and rewinding support require careful considerations and considerable effort in tailoring the components so that they interact precisely as needed. This is also part of the reason why most current AV systems offer only restricted rewinding or reverse playing, if at all.

In this section, we analyze the individual requirements that are important for an *abstract* animation playing front-end. We strive to support as many of them as possible inside the framework to make the work for the developers of a concrete application easier, and give them interesting and important functionality "for free".

We start our discussion with a few relatively easy requirements. The first issue we have to address is the independence of an *Internet connection* (**GR4**). As discussed in chapter 3, some AV systems require an Internet connection for executing operations on the central server. Requiring an active network connection may allow easy system updates, especially if part of the content is worked on at the server side, instead of just providing fixed code. However, we have to acknowledge that not all users may have an Internet connection. This is especially the case for applications of *learning on the move*, whether in trains or on a plane. Even in settings with free Internet access, we have to take possible network failures into account. In many countries, maintaining an Internet connection by dial-up also entails costs for phone calls, apart from fees gathered by the Internet access provider. Restricting the system to an Internet connection may thus limit the number of possible users, either due to technical or monetary considerations.

Luckily, the framework as described in this chapter does not require an Internet connection to run. There are only two places in the framework that actually profit from an Internet connection: downloading animations from the Internet, and acquiring new extensions and bug fixes. However, neither of these operations is essential to running the program. An inability or unwillingness to dial in to the Internet while applications based on the framework are running therefore has no adverse effects.

Another important consideration concerns the system performance (**GR8**), which must at least be "sufficient". As can be seen from the quotation marks, sufficiency is at least partially subjective. We can state that the performance of the framework depends largely on the performance of the installed Java runtime environment, as is the case for most Java-based software. This concerns mostly two areas: a shortage of free memory and just in time compilation.

The amount of memory available to the Java virtual machine has an impact on the performance. The Java virtual machine has an upper limit on the amount of RAM it can allocate. This limit can be modified by start-up parameters. It is important to know that the Java virtual machine will not allocate more memory than this limit states – even if much more free memory is available.

Just as the other central components of the framework, the content display front-end shall also be flexible in its application area (**GR10**). Offering highly specific operations with a special AV application in mind reduces the chances for reuse, as the application may not be usable for all AV contents. The front-end will therefore have to strive for a mostly "implementation-neutral" appeal.

One of the special features we required in chapter 3 is the ability to link to external content, typically in HTML format (**UR4**). The same application is used in many of today's tools when activating the built-in help function. Typically, the documentation is presented in HTML or the Microsoft Windows™ help format. Some of today's programs also connect to the Internet for selected features; this is for example the case with several Microsoft applications. Therefore, adding links to external

documentation shall be supported, although the program must also be able to continue working if no Internet connection is available.
One of the most prominent feature of the framework is the incorporation of extensions (**VR2**). Here, provisions shall be taken that allow the integration of extensions even while the program is running. Section 4.2 already explained how extensions can be added and removed on the fly.
Another important aspect in the flexibility of our framework is the ability to play animations in both directions using a variety of controls. In keeping with requirement **UR5** and in partial support of the requirement for repeated user interaction (**UR7**), we want to support the following control operations, as shown in Figure 4.19:

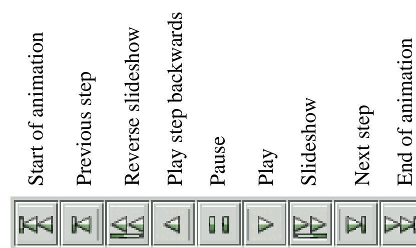

Figure 4.19: Example Control Tool Bar

1. jump to start of the animation,
2. jump to the beginning of the previous step, without a limitation on how often this can be done,
3. perform a reverse slide show mode, which respects all timed links and inserts a configurable time delay between steps which would normally wait for a user event,
4. play the current step in reverse to allow following the performed operation in reverse mode,
5. pause the animation,
6. play the current animation step and succeeding steps provided they are linked by timed transitions,
7. perform a slide show mode by stringing together all steps, respecting all defined delays between steps and treating event-based links as if they had a (configurable) fixed delay,
8. jump to the start of the next step,
9. and jump to the end of the animation.

We can state that supporting reverse playing is easily accomplished within our framework. Our design of the animator, primitive and handler components as well as the animation representation has been optimized to allow for smooth reverse playing and rewinding. Skipping ahead or backwards in the animation (fast forward and rewinding) are very easy to do.
Achieving a slide show mode is also easy if we assume that we can play the current animation step in the appropriate direction. Adding the slide show functionality then only requires checking the

link object that connects the current step to its successor for the transition mode. If the mode is set to a timed delay, or to user interaction-based events beyond clicking "play", execution proceeds normally. Otherwise, the visualizer-defined event is ignored and a predefined, user-configurable default delay is performed. The animation can then continue with the next step. The same process applies to a reverse slide show, if one replaces the (forward) playing with backward execution and the next step with the previous one.

One of the additional required features concerns the support for step labels that act as hyperlinks (**UR12**). We have already shown that the Link objects of the framework allow the addition of labels. Each link is also aware of the animation step it connects to its successor, as described in section 4.7. Therefore, the animation display GUI only has to offer a small component that can gather all defined labels in a list and waits for user selections. If a user clicks on an element in the list, the target step to assume can be extracted easily from the underlying Link object. The framework can then simply invoke the *setStep* method of the AnimationState class to reset the animation to the associated step, as described in section 4.7.

The adjustment of detail level as required by **UR16** cannot be achieved in a generic framework. The only partial support for detail adjustment possible for the framework is zooming the display canvas. As the primitives are painted on a graphical context, this can be achieved easily by scaling the underlying context before rendering it on the screen.

There are two main components in the animation display front-end specified by the framework. The *animation window* contains the slide rulers for controlling the display speed (**UR30**) and magnification (**UR16**), the animation control tool bar shown in Figure 4.19, a text field and a third slide ruler for controlling the current animation step by entering its number or dragging the ruler (**UR5**, **VR8**) and finally the animation canvas on which the primitives are painted.

The second component of the display front-end that can be fully defined in the framework is the *animation structure view* which presents those labels which are present in the current animation in a list view and listens to user clicks. All of these entities including the canvas and the animation window itself are instances of the *Singleton* pattern [61, p. 127ff].

We will now look at the process of *reverse animation playing* in more detail. In general, the execution of smooth reverse playing is mostly the same as for smooth forward playing. Therefore, we will first briefly examine how smooth forward playing is performed.

Instead of starting with a vector of primitives which are iteratively transformed by each animator working on them, we keep the original primitives separate from their animated clones. Even irreversible animators that destroy the primitive structure, such as scaling an object with a factor of 0 in either direction, can thus be reversed easily. Compared to the standard approach of keeping the previous instances on an undo stack, our approach allows for unlimited reversibility.

Cloning primitives requires additional memory for each generated clone. This memory can in principle be freed once a new set of clones is requested. However, Java does not support explicit memory deallocation by developers, so that both user and visualizer are at the mercy of the garbage collector. However, note that each cloning operation will at most clone as many primitives as are present in the whole animation. The amount of memory required for cloning thus depends on a predictable value – the number of primitives used within the animation for the worst case scenario –, instead of a highly dynamic and unpredictable value depending on the precise timing and content of the stored animation effects. Our approach therefore normally consumes far less memory than an elaborate undo stack that stores all intermediate dynamic states of the last few animation steps, for example as images. Additionally, the highly improved functionality makes the consumption of memory highly worthwhile.

The first operation to perform when executing a timed animator is determining the current system time and display frame number ("ticks"). Ticks refer to the number of frames already displayed and are a useful virtual time measurement, as they adapt to the processing power of the current system. All animators are then initialized with the current animation state and time. Note that the animation state also contains the current state of the objects. The initialization also sets the start time and ticks for the animator.
We then enter a loop that is performed until all animators for the current step are finished. The body of this loop is shown in Figure 4.20. First, the current point in time is determined. For each active animator, we then invoke the *action* method with the current time as a parameter. The net effect of this invocation is that each animator determines the target state to assume for the current point in time based on the current time and its execution start, as passed by the initialization method. For timed animators, the *getProperty* method as described in section 4.5 is used to determine the target state, and an appropriate *PropertyChangeEvent* object is sent to each primitive.

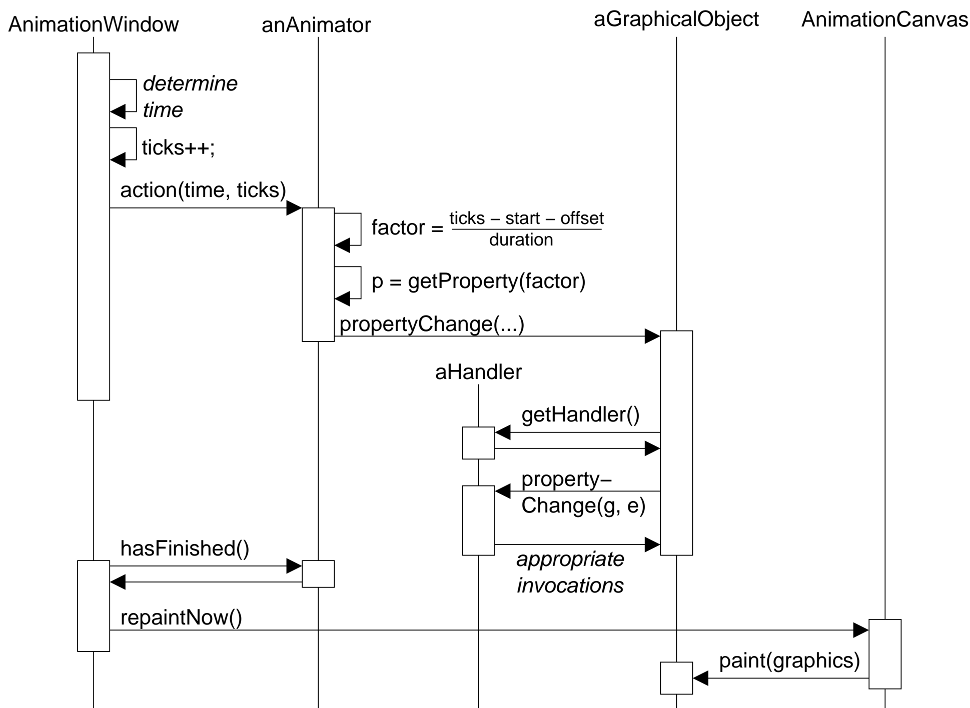

Figure 4.20: Timed Animator Execution

The event is directly forwarded to the *Singleton* handlers for the underlying primitive types by adding the current primitive. The generic handlers map the PropertyChangeEvent into a set of invocations that cause the appropriate transformation of the primitive.
After performing this operation, each animator is checked for its finish status. If finished, it is removed from the queue of current animators. In any case, execution proceeds with the next animator. Once all animators have been executed, the time is incremented. The animation window invokes the standard Java *repaint* method which causes all primitives to paint themselves in their new state on the animation canvas. The loop over the animators continues until all animators have finished executing. The animation step is then set to the start of the next step.

This approach is flexible enough to be used after a small modification to also address reverse playing. In effect, there are only three places where we have to alter the code slightly to achieve fully dynamic reverse playing: in the initialization, the *action* invocation and the changing of the animation step at the end of the execution.

The initialization of all currently active animators for *forward* execution consists of two activities, as outline above: determining the current system time and display frame number (or “ticks”), and initializing each animator with the current animation state and the current time information. For *reverse* execution, we also initialize all animators with the current animation state and current time. However, we conceptually have to treat the “current” time – which marks the execution start of the step – as if it lay in the past. To do so, we determine the duration of the current step as the maximum of the sum of all offset and duration pairs of contained animators. This value is added to the current time value. The new value represents the moment in time when the current step would finish executing if we started it in forward playing mode “now”.

We now have two pieces of timing information: the *actual* time, which we take to be the start of the animation step in *forward* mode, and the *modified* time, which represents the *end* of the animation step in forward mode. Therefore, we will use the *actual* time for the animator initialization, but start the execution of the *action* method with the *modified* (“end”) time. The main execution loop is performed *backwards* by replacing the *time increment* by a *time decrement*, so that the execution of the animators starts with their end state and successively approaches their initial state. The net effect of this is a fluent reverse execution which can even handle the reversion of destructive animators, such as scaling a primitive with a factor of zero.

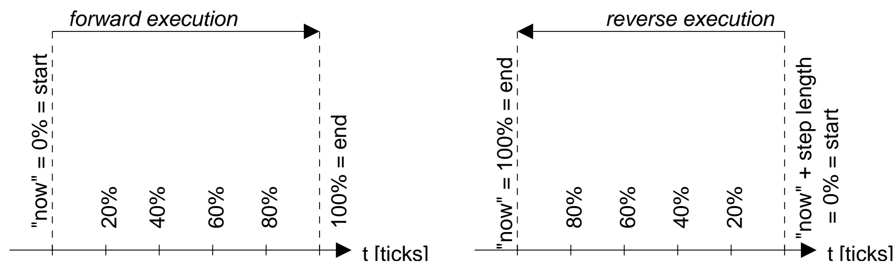

Figure 4.21: Treating Reverse Execution in the ANIMAL-FARM Framework

Figure 4.21 illustrates this process. The left side of the Figure shows how forward execution proceeds. The current time (“now”) is used as the reference point for the start of the current animation step. The execution proceeds over multiple intermediate stages until all of the step is executed (and thus 100% of the execution is reached). For simplicity reasons, the figure only shows the 20%, 40%, 60% and 80% stages.

The right side of Figure 4.21 shows the same process for *reverse* execution. The current time also serves as the *start* point of the animation step. However, the animators start executing at the *end* of the animation step, determined from the start time of the step and the duration of the step. Therefore, the timing information passed along to the individual animators is initialized to the time resulting from adding the step duration to the start time of the step. After 20% of the reverse step execution, the animation state will be the same as after 80% of forward execution, as the internal execution time is diminished in each loop iteration. Thus, after 100% of the reverse step execution is performed, the animation has gone back far enough in time to end up at the start of the step.

In a different vein, we can also enable the user to modify the display speed by providing a slide ruler. The value of the slide ruler represents the amount of time to be added to the current time after each loop iteration. We only have to replace the increment operator for forward playing by adding the chosen value to the current time. For reverse playing, we replace the decrement operator with a subtraction of the chosen value. Thus, adjusting the display speed as required in **UR30** is trivial to achieve in our framework. This feature can also be used to address the issue of graceful degradation (**UR31**) on slow or fast machines by dragging the slide ruler appropriately to speed up or slow down the interpretation process.

4.9 Animation Import and Export

As stated in section 4.2.2, the AV framework has to adapt the import and export layer classes slightly to offer properly typed methods. However, this is easily accomplished by replacing the *Object* parameters and return types by type *Animation*, as shown in Figure 4.22 and 4.23 for the import and export layer, respectively. No other additions or modifications to the import and export layer are necessary. The abstract framework components are placed in a dashed box.

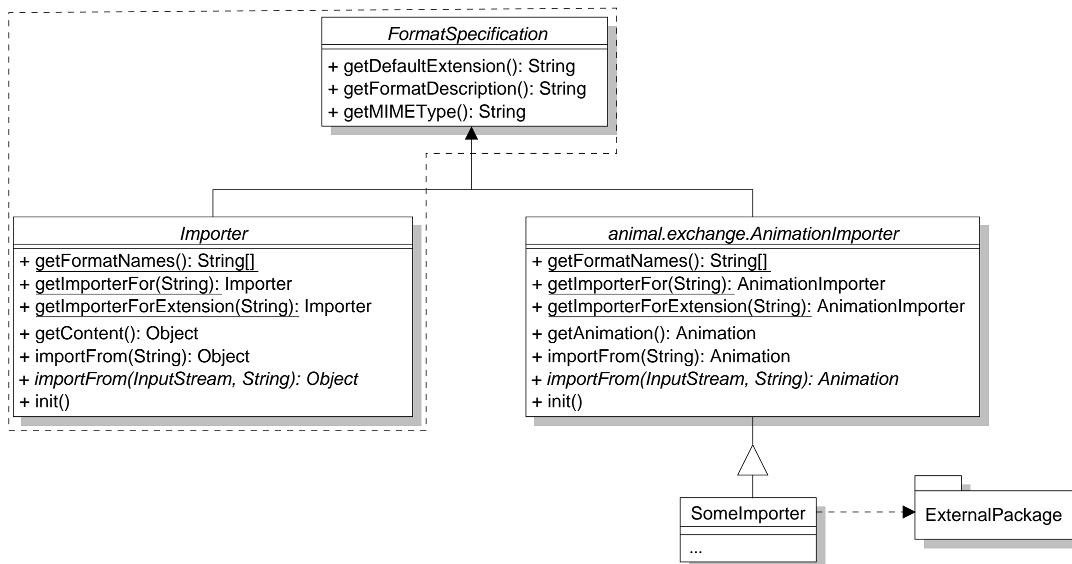

Figure 4.22: Import Layer Structure

4.10 Summary

The main goal of this chapter was developing a reusable, dynamically extensible framework for AV. This challenging task can be split into three different research areas: specifying components for a dynamically extensible framework (of arbitrary content), defining a solid framework structure for AV, and embedding useful relevant AV features into the framework. We will regard the achievements in these three segments in the order given.

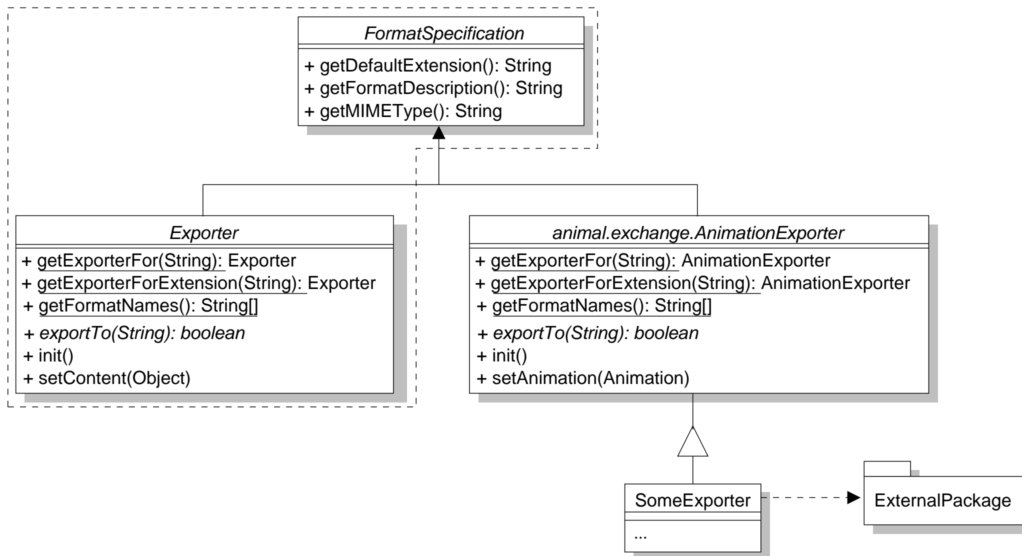

Figure 4.23: Export Layer Structure

The incorporation of properties for representing the state of the participating entities allows dynamic adaptation and extension, as new keys and values can easily be inserted into the properties. At the same time, data hiding and encapsulation purposes are still met by the changed representation of object state. As a further bonus, the properties can provide default values, replacing the standard "zero" or null value returned when accessing "uninitialized" attributes by the specified default value. Properties can easily be shared between objects, but can also be cloned to provide a separate copy of each entry to the participating entities.

The flexible MIME type-based import and export layer offer simple facilities for supporting an arbitrary number of different import and export filters. This is especially of interest when the actual file filters are added or removed dynamically.

To achieve maximum applicability of our framework, we have also included support for the translation of messages and even the whole GUI into a set of different languages. The only constraint of this support is the number of language resource files present. In theory, our framework can support an unlimited number of languages and allow fast switching by a single mouse click. However, this depends on (other) users and developers helping us translate the language resource files. As these are encoded in a line-based ASCII notation as used by Java properties, this task is easy to achieve if time-consuming, provided that one has enough familiarity with the target language.

As an added bonus, the *TranslatableGUIElement* class for generating translatable GUI components also makes the generation process much easier by internalizing most of the otherwise necessary method invocations. It also offers dynamic elements such as *Action* buttons that can directly invoke a method by name, as specified in the configuration file. Changing the language used in the GUI may therefore also automatically change the association of some buttons to specific methods.

The framework and concrete subsystems are assembled dynamically using dynamic loading. Most entities loaded in this way are stored as Prototypes [61, p. 127ff] in a dynamic look-up structure. Thus, adding or removing new entities is very easy. As outlined in this chapter, implementing extensions typically requires neither source code modifications nor deep system knowledge. At

run-time, the system can be reconfigured using the *ComponentConfigurer* component, both by adding and removing components. Components that are marked as "core" cannot be removed so as not to endanger the system's stability.

The dynamic configuration of the framework during a session depends on a configuration file listing the individual classes to be made available for the user or visualizer. The configuration can be entered manually by editing the ASCII-based configuration file, or dynamically using a special GUI component.

The configuration file is searched in the directory where the system is started first, followed by the entries of the *CLASSPATH* and the distribution *jar* file. Therefore, users can keep multiple copies of the configuration file representing different interests or views in separate directories. The concrete system will then look slightly differently depending on the directory it is started in. Note that all users can use the same jar file, even if their configurations vary. Of course, additional extensions that lie outside the jar distribution have to be placed inside the *CLASSPATH* so Java is able to locate them. Using the same jar file is especially helpful in a network setting. Here, the jar file can be provided as a shared network resource, together with the default configuration file. Users who want to modify their local configuration can then copy only the configuration file and modify it accordingly.

The introduction of handlers as agents for negotiating services between two entities adds even more flexibility to the system. As each handler or handler extension is linked to a fixed type, they are represented by Singletons [61, p. 217ff]. Within the course of the framework, the handlers are placed between primitives and animators. Animators can query the handler for the set of concrete animation effect subtypes applicable to a given primitive or set thereof. The handler generates the returned list from its own knowledge about the underlying primitive class, as well as from the primitive's state. When the animator has to perform an effect, it resolves the current timing information and sends the concrete effect and the animator's previous and current state to the handler. The handler then examines the data passed in and invokes the appropriate methods on the primitive.

Handler extensions are lightweight classes that act in the same way as the "fixed" standard handlers, but are maintained as Singleton instances in a dynamic look-up structure. The "standard" handlers are required for animating a given primitive, and are thus automatically loaded in whenever a new primitive is added to the system at start-up or during run-time. Extension handlers can be added and removed at any time and are automatically included in the animation effect resolution process.

The framework thus offers a very strict separation of concerns, especially regarding handlers. Several advantages spring from this approach. Firstly, it is very easy to develop new entities, whether primitives, animators, handlers or handler extensions. The developer can focus on *one* aspect during the development. For example, implementing a new primitive requires absolutely no knowledge about animation. The animators completely abstract from the underlying types. Finally, the handler instances are tied to a specific primitive class and act as *Prototypes* [61, p. 117ff]. Implementing a handler or extension thereof on a known primitive is therefore also simple, as only *one* special class has to be kept in mind.

As can be seen, the specified framework is reusable and employs an open and lean class structure. As such, developing a concrete system on the basis of the framework design is not difficult. The framework contains hotspots for extensibility in nearly all areas. In fact, there are only two areas where extensions are not planned: in the *animation representation* and the *display components*. These central components should not be subject to arbitrary change, as this may have catastrophic consequences for the concrete system. Of course, the developers of a concrete system based on the framework can add further extensions facilities to their own system.

Two base classes are employed widely throughout the framework: *PropertiedObject* and *EditableObject*. The former represents an object whose state is represented by properties, with the added facility for cloning the properties from another object. The latter adds editing capabilities to PropertiedObject instances. The editing process has to be implemented by the developer of the concrete system to take individual preferences and system-specific demands into account. One example implementation may follow the Java Beans approach for editing properties.

Animations in the framework consist of a set of animation steps. Each animation step may contain an arbitrary number of animation effects. Steps are linked by *Link* objects which encode the transition between steps. For example, the transition may be performed automatically after a visualizer-set delay, or wait for a specific user event. Links can also be labeled. These labels are gathered in a list, where each label acts as a hyperlink to the associated step, updating the display accordingly.

The usage of an *AnimationState* object that provides clones of the primitives present in the current animation state offers advanced rewind and reverse playing functionality. Even in 2001, experts in the field of AV stated that "currently no AV has rewind" capabilities, and considered an "efficient rewind one of the most important 'open questions' in AV" [2]. We submit that less than a year has passed between this published statement and the finishing of our framework that offers full-fledged rewinding including dynamic reverse playing.

Graphical Objects ("primitives") as used within the framework offer a clean separation of concerns. They are only aware of their current state and thus see themselves as "static" entities. Methods for retrieving and especially changing their status allow changing the way the primitives are displayed. Each primitive also implements the *paint(java.awt.Graphics)* method for rendering itself on a graphical context, typically a canvas. As no knowledge about the dynamic behavior during the animation is encoded in the primitives, they are highly reusable and could for example easily be transported into a straightforward non-animating drawing program.

Animation effects ("animators") represent one general animation effect each. For example, a single animator is responsible for *all* translation (move) transformations. The animator has to be able to store timing information such as the current time, start time of the effect execution, duration and offset from the start of the animation step. Based on this information, the animator has to determine its current state during each point in its execution.

This state only represents the value to be assumed by the animated primitives, and is thus completely independent of the actual animation subeffect. For example, if the animator represents a move effect on a line, it is only responsible for calculating the point on the line reached at the current time. The animator does *not* modify the animated primitives.

Handler prototypes introduce a powerful general concept for service negotiation. Within the framework, they are used for two operations: determining the list of animation subeffects possible for each primitive type and mapping property changes to primitives. For example, a handler can inform a *Move* effect that the selected primitive can offer three different move effect types: moving the whole object, only the first node, or only the second node. The animator gathers this information from all handlers and then builds an intersection to determine the subeffect types applicable to all selected primitives. The visualizer can then select from these types.

When performing an animator, the animator determines its current state as defined above. This state, together with the primitive worked on, the concrete subeffect and the previous state, is then sent along to the handlers of each animated primitive. The handlers extract the information and map it to the specific method invocations that achieve the requested effect. Additionally, handlers can forward requests to *extension handlers* which can be added dynamically for further animation

effect functionalities.

The display features the ANIMAL-FARM AV framework offers go beyond what most other comparable systems can offer. On the one hand, ANIMAL-FARM offers the same functionality as we are used from video playback devices, including fast forward and reverse, dynamic reverse playing, jumping to the start and end of the animation and a slide show mode that strings all animation steps together. Additionally, the user can adjust the display speed and magnification as well as the percentage of the animation show using sliders. Finally, the optional labels of individual animation steps provide a “table of contents” view of the animation structure [66] and act as hyperlinks for directly jumping to the associated animation steps.

Chapter 5

The ANIMAL Animation System

5.1 Introduction

In chapter 2, we have explored a large selection of currently available animation systems and applets. The evaluation yielded an elaborate set of 69 requirements for an "ideal" AV system in chapter 3. Based on some of the requirements, we have introduced a framework for an extensible and configurable AV system in chapter 4.
The framework fixes some aspects of extensible AV systems, but leaves others open to the individual developer. For example, the basic components of the target system are fixed, including the principal components. However, the precise implementation details are deliberately left vague to allow for personal design decisions. In this chapter, we present the prototypical extensible AV system ANIMAL built on the proposed framework. We also highlight which design decisions were left open for the individual components, and what decisions were taken and why.
As explained in section 4.1, ANIMAL is an acronym for *A*dvanced *N*avigation and *I*nteractive *M*odeling of *A*nimations for *L*ectures. The name expansion indicates that ANIMAL is specifically geared to be usable within lectures and other educational settings. As ANIMAL is built on the ANIMAL-FARM AV system framework, we can focus on the more salient design issues in this chapter. Our presentation of ANIMAL follows the structure of the previous chapter in describing the individual components.
Section 5.2 describes the general architecture of ANIMAL. Section 5.2.1 presents the graphical primitives supported by ANIMAL, followed by a discussion of the animation effects in section 5.3. Section 5.4 presents the coordination of the graphical primitives and animation effects using transformation handlers. ANIMAL's graphical front-end for displaying and editing animations is presented in sections 5.5 and 5.6, respectively. Finally, section 5.7 describes the import and export layer implementations. Section 5.8 summarizes the chapter.

5.2 ANIMAL Architecture

ANIMAL consists of a core program, an import layer, a graphical generation and editing component, a graphical front-end for animation presentations and an export layer. The core program covers the main animation window and other base classes, as well as the currently available graphical primitives and animation effects. The import layer contains a set of import filter components which are loaded on demand, allowing for dynamic addition or removal. The same applies to the export

layer components. The graphical generation and editing front-end allows adjusting the properties of the currently loaded animation, as well as adding, modifying or removing both graphical primitives and animation effects. Finally, the animation playing front-end offers flexible controls for viewing and rewinding the animation.
The *ComponentConfigurer* introduced in Figure 4.8 on page 82 is used for dynamically assembling the individual components usable during a given ANIMAL session. It contains a tabbed pane with a set of tabs each addressing one area of extensibility or "hotspot". In most cases, the user of the system can simply enter the name of the component in the text field shown at the top and press Return. ANIMAL then determines the "real" class name of the component from the user input and the currently active tab and tries to load the associated class. If loading was successful, the class is dynamically inserted into ANIMAL's runtime environment, added to the list of components shown on the associated tab, and can be used immediately. Thus, adding new components is very easy to achieve if the name of the component is known.
The *ComponentConfigurer* shown in Figure 4.8 on page 82 can also be used for removing extensions by unchecking the box placed before the component's name. Note that this is only possible for extensions; core components are selected but grayed out to prevent their removal. Further configurations can be performed in the ANIMAL *Preferences* window, as well as in the individual editor windows. The last value settings used for a given primitive, effect or step are always treated as the default values for the next instance of the same type. Configuring individual elements is therefore also easy.

5.2.1 Graphical Primitives

The framework includes a precise definition of the graphical primitive base class. Section 4.4 specified the super class for all primitives and the set of operations offered therein. However, Figure 4.11 on page 93 specified only a single data attribute for all graphical primitives, namely the numeric identification. Obviously, more information is needed for actually encoding the primitive representation. Even the methods presented in the framework, such as *getColor()*, require an appropriate underlying data representation.
As described in section 4.2.1, we have decided to encode the basic representation of graphical primitives, such as color, font and depth information, as *properties*. Where necessary for efficiency reasons, we use direct encoding with attributes for selected properties. This is especially the case for object nodes, for example in a polygon object, which have to be referenced often. The type conversion required by using properties may introduce an unacceptable delay if a large set of transformations is performed, as is likely in larger animations.
All ANIMAL graphical primitives are placed in the package *animal.graphics* and extend the abstract class *PTGraphicObject*, where *PT* stands for *Portable Toolkit*. As shown in Figure 5.1, *PTGraphicObject* instances extend the framework class *GraphicObject* shown in Figure 4.11 on page 93. They are therefore editable and propertied. In this and the following figures involving the framework, the framework components are placed in a dashed box if present. Implementors of specific framework components are linked to their ancestor by standard inheritance arrows. Note that in keeping with standard UML notation, methods inherited from the framework are *not* repeated in the implementing subclass.
The class attribute *registeredHandlers* is a hash table used for mapping animation effects to intermediate *handler* agents. The functionality of handlers was already introduced in section 4.6. The actual mapping is described in more detail in section 5.4. The private *num* adopted from the frame-

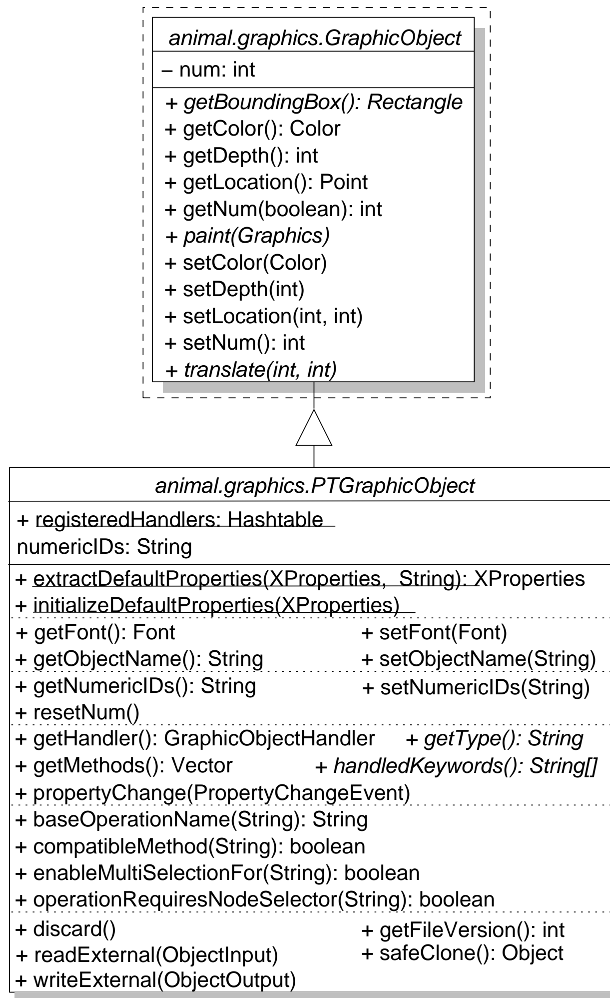

Figure 5.1: *PTGraphicObject* as an Implementation of *GraphicObject*

work has a unique value for each graphical primitive and thus acts as a reference for the object. The *numericIDs* attribute is a convenience storage object that prevents multiple conversions of the numeric object ID to String objects.
For better readability, the large interface of class *PTGraphicObject* has been segmented using dotted lines. The first segment contains the two class methods *extractDefaultProperties* and *initializeDefaultProperties*. They are used to initialize the current graphical primitive from a set of predefined properties. The object properties are extracted from a set of general properties. This process employs the *getElementsForPrefix* method of class *XProperties* as described in section 4.2.1.
The second segment contains several convenience methods for setting and retrieving attributes commonly found in most primitives. The framework already provides methods for accessing and setting the *color* and *depth*. ANIMAL's *PTGraphicObject* adds methods for retrieving and modifying the *font* and *object name*. Note that the precise interpretation of the *color* depends on the object's type. ANIMAL chooses the most basic color for the *getColor* method. Therefore, the method will return the *outline* color for a polygon rather than its fill color.
The *depth* of a primitive is used for resolving overlapping primitives. It acts like the z coordinate in three-dimensional geometry, with higher values indicating a position further to the background. Note that *negative* values represent positions in front of the display and are regarded as illegal. The *font* property is applicable to all graphical primitives that may contain a text component and covers the complete font information including font name, size and style. Finally, the *object name* is an optional *String* assigned to the primitive for easier reference. The primitive's *num* attribute is sufficient for uniquely accessing the object, but only allows for natural number "names" which may be difficult to remember.
The third segment lists the methods for accessing the primitive's numerical identity. The framework provides the methods *getNum* and *setNum*, as introduced in section 4.4. ANIMAL adds the method *resetNum* for resetting a given object's number to 0. The boolean parameter of *getNum* indicates whether a new unique number shall be assigned before returning the current numeric ID. To prevent careless use, a new unique identification is only assigned if *resetNum* has been invoked. This is a necessary safety measure as the animation effects presented in section 5.3 all work on the numeric IDs of the primitives and are not updated automatically if a new number has been assigned. The methods for accessing the String representation of a set of numeric IDs are convenience facilities which are mostly used by animation effects.
The fourth segment offers methods required for performing animation effects on the graphical primitives. They are discussed in more detail in section 5.4. The following segment offers general methods needed within the GUI-based animation editing. They are discussed in section 5.6 focusing on GUI-based generation and editing.
The methods *paint* and *translate* taken from the framework are abstract, as they cannot be fully specified for abstract generic primitives. The *paint* method is responsible for displaying a given primitive's current state on the graphic output context, while *translate* adjusts the primitive's location. Note that *translate* is invoked internally from *setLocation*.
The final segment of methods deals with object versioning. It also includes support for the *java.io.Externalizable* mechanism. Although externalization has been deprecated, the support is still present to allow the parsing of older animation files. See section 5.7 for a discussion of export and import handling.
The method interface of *PTGraphicObject* is relatively large for an object-oriented design. This is due to the different interests that have to be covered within the base class: convenience property access and setting, animation effect interface as well as storage and retrieval facilities.

Note that the framework class *GraphicObject* is a subclass of *EditableObject*, which again extends *PropertiedObject*, as described in section 4.2.1 and 4.4. Therefore, all subclasses of *PTGraphicObject* also inherit the methods and attributes of the classes *PropertiedObject* and *EditableObject*. They can thus access an internal state representation using properties and have a link to graphical editors. Furthermore, each subclass has a class attribute representing the default properties for all objects of the given type.

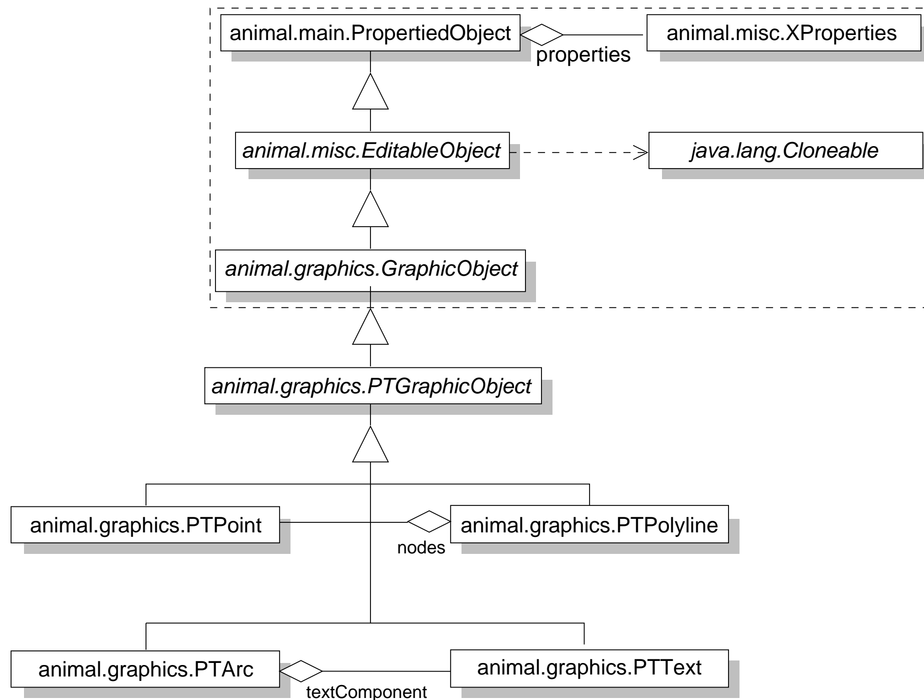

Figure 5.2: Inheritance Structure of the Graphical Primitives in ANIMAL

The inheritance structure of the graphical primitives provided by ANIMAL is shown in Figure 5.2. The individual subclasses shown are discussed in the following subsections.

5.2.2 Point Primitives

The *PTPoint* class represents a single point as a special instance of a *PTGraphicObject*. Points are the smallest geometric unit. A *PTPoint* object occupies exactly one pixel on the display and is therefore practically invisible. However, points are needed for assembling more complex objects such as polygons. They are also needed for specifying the center of a rotation effect. We do not provide example screen shots of ANIMAL *PTPoint* instances, as their nature prevents them from being very interesting or surprising to look at.

Figure 5.3 shows the added methods provided by the class. Note that for efficiency reasons, the location of a point instance is represented by a *double* array. The location is also stored in the properties and included in the class *DefaultProperties*. For a list of methods inherited from the base class *PTGraphicObject*, please refer to Figure 5.1 on page 119 and Figure 4.11 on page 93.

The methods placed above the first dotted line represent property setting or accessing. The standard

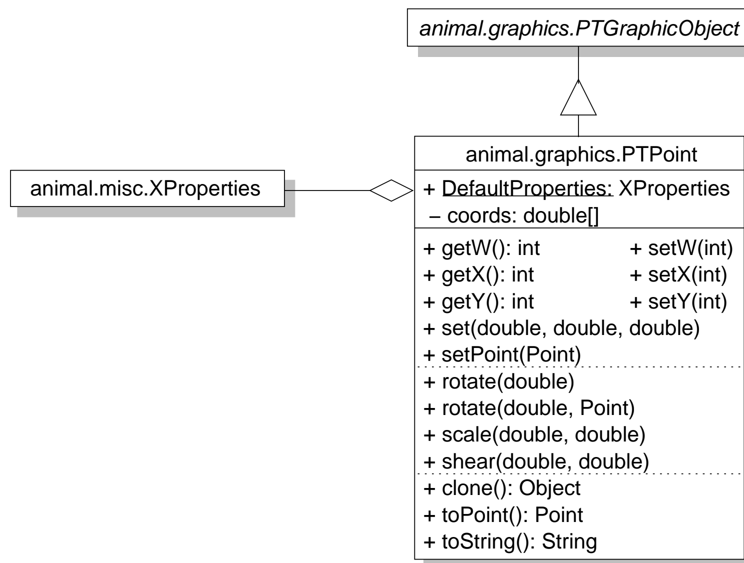

Figure 5.3: Class Diagram for *Point* Primitives in ANIMAL

graphic operation *translate* inherited from the *GraphicObject* framework class is implemented in PTPoint. In addition, PTPoint provides support for *rotating, scaling* and *shearing* point primitives. The last set of methods cover the conversion of a point to a *java.awt.Point* or String representation, as well as a secure *clone* operation that prevents unwanted side effects such as duplicated references to primitive properties.

ANIMAL point primitives are specified using double precision homogenous coordinates [59]. Note that although *double* precision coordinates are used for points, only integer coordinates are actually retrievable. This is due to the basic graphic operations in Java, which are based on integer pixel coordinates.

Homogenous coordinates represent a two-dimensional coordinate by three values labeled *x, y* and *w*. For $w = 1$, (x, y) represent the coordinates of the point. For $w \neq 1$, the value of w represents a scaling factor applied to the x, y coordinates. Note that a value of $w = 0$ is invalid. Figure 5.4 summarizes the interpretation of homogenous coordinates.

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} \equiv \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ 1 \end{pmatrix} \equiv \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \end{pmatrix}$$

Figure 5.4: Interpretation of Homogenous Coordinates

The advantage of using homogeneous coordinates lies in the consistent representation of graphical operations [59]. Standard operations such as *rotate* are implemented using matrix multiplications. However, this is not possible for *translations* which add to the value of the *x, y* coordinates. Using homogeneous coordinates, this addition can be achieved by multiplying the *w* entry with the target

summand while assuming that $w = 1$. Although there are faster ways to represent the graphical operations, matrix multiplications allow for easy accumulation of effects.

5.2.3 Polyline and Polygon Primitives

ANIMAL's *polyline* or *polygon* primitives consist of a vector of nodes linked by edges. Figure 5.5 shows the class diagram of the *PTPolyline* class that represents both polyline and polygon objects. Note that each node is represented by a *PTPoint* primitive. *PTPolyline* primitives also implement the *animal.animator.MoveBase* interface, and can thus be used as the base object for arbitrary move effects.

The main difference between a *polyline* and a *polygon* is that the *polygon* is closed, that is, its last node is equal to the first node. Additionally, only polygons may be filled and possess a fill color. Polylines, on the other hand, may possess a forward or backward arrow that points away at a right angle from the last or first node, respectively.

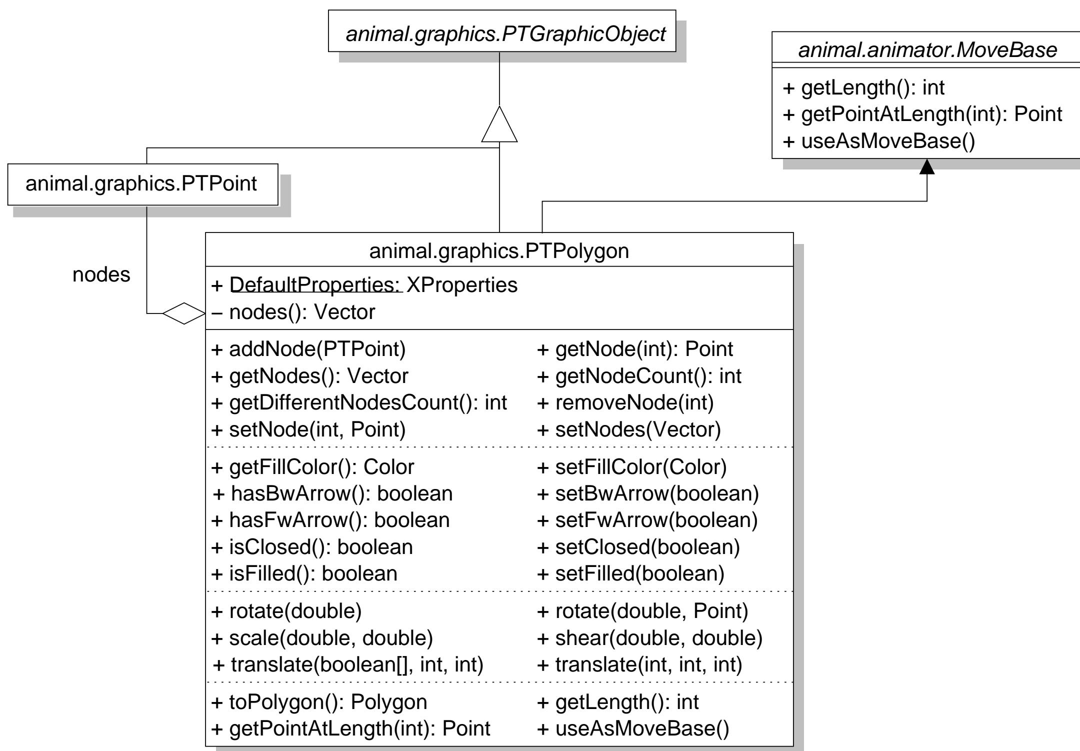

Figure 5.5: Class Diagram of *Polyline / Polygon* Primitives in ANIMAL

ANIMAL places no upper bound on the number of nodes used for a polyline object. A polyline must consist of at least two nodes and thus represent a simple line to be valid. A polyline with only one node is simply a point and has to be represented as such.

The first set of methods available for polyline or polygon primitives support the adjustment of nodes. Apart from setting and retrieving a node at a certain index, ANIMAL also allows adding a single node and setting or retrieving the set of nodes using a *java.lang.Vector* of *PTPoint* entries. In addition, the *getDifferentNodesCount* method can be used to query the nodes of a polygon primitive.

For polylines, the return value is equal to the return value of the *getNodesCount* method. However, the implementation of polygons in ANIMAL sets the last node of a polygon identical to its first node. The two methods will therefore have different results for polygon objects.
The methods for querying and setting display properties are shown in the second method segment. PTPolyline instances provide methods for retrieving and setting the fill color of a polygon, as well as testing or setting the existence of pointers at the start or end of a polyline. The *closed* property is important for the distinction between polyline and polygon primitives. The *filled* property is only meaningful for closed polylines, that is, polygons.
The methods for querying and setting the display state access the internal properties of the object. They could be removed from the class without losing expressiveness. However, this may cause problems for developers who wish to access a given value and would have to resort to guessing the associated key. Note that neither of these methods is applicable to all instances of *PTPolyline*: the fill color is only meaningful for closed polyline instances (and thus polygons), while the forward and backward arrows are only meaningful for "open" polylines.
The implementation of the *translate* transformation as the third set of offered methods is almost trivial. The standard *translate(int, int)* method inherited from *PTGraphicObject* translates all nodes of the structure at the same time. The special *translate(boolean[], int, int)* is used to translate a subset of nodes at the same time, instead of all nodes. Only those nodes where the boolean array at the node index contains *true* are translated. The *translate(int, int, int)* method translates only the node at the index passed as the first argument. All translation requests are mapped directly to the set of *PTPoint* nodes and executed within that class. The other transformation methods behave as expected. They perform the requested transformation by mapping the request to the set of nodes.
The final set of methods offers a conversion to a *java.awt.Polygon* representation as well as methods needed for animation effects. The *getLength* method returns the geometric length of the object. *getPointAtLength* returns the point on the object reached after traversing it for a specified length. The *useAsMoveBase* method opens the object, activates the forward and deactivates the backward arrow.

5.2.4 Text Primitives

The *PTText* class represents text objects consisting of a single line of text. Figure 5.6 shows the UML class diagram for this class. *PTText* instances share a *DefaultProperties* object and the *escapeText* method. The method is used to escape unsafe text elements such as the single or double quotation marks. The text and font information is stored in the internal properties representation. All other inherited methods from either the framework or the *PTGraphicObject* class therefore provide only a short-hand notation for accessing the value from the properties. The relevant methods from the *PTGraphicObject* class include accessing or setting the text color, font and depth properties. *PTText* adds methods for retrieving and setting the text component, as well as for performing the graphical transformations *rotate, scale* and *shear*.
A special problem occurs when implementing the *scale* and *rotate* operations. Standard Java's *java.awt.Graphics* class does not support scaling or rotating text entries. Text scaling could in principle be accomplished by adapting the font size. However, this implementation only works if two conditions are met: first, the x and y scale factor must be identical, and second, the chosen font must be available in the calculated size. As the scaling factor is a *double* value, the resulting font size is highly likely to be a non-natural value. However, Java only allocates fonts with a natural font size, so certain compromises have to be made. Additionally, rotating the text is not

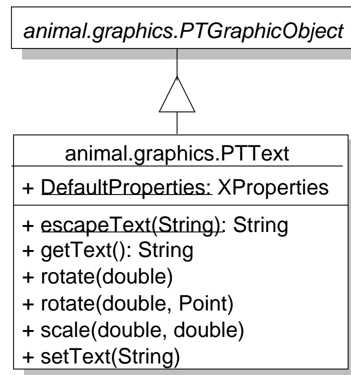

Figure 5.6: UML Inheritance Diagram for the *PTText* Class

easily accomplished due to lack of direct support. We have therefore decided to use the scaling and rotation support in Java's *java.awt.Graphics2D* class available in current JDKs.
For maximum portability, ANIMAL only supports the three font families *SansSerif, Serif* and *Monospaced*. These font names are substituted with a system-specific font, typically Helvetica or Arial for fonts without serifs (SansSerif), *Times Roman* or *Times New Roman* for Serif fonts, and `Courier` or `Dialog` for `Monospaced` fonts. ANIMAL does not support the inclusion of system-specific fonts such as *Helvetica*, as such fonts may not be present on a given client system. In those cases, the display of texts has to revert to a common basis, which typically results in problems for the layout. The generic fall-back font is a plain 16-point SansSerif font.

5.2.5 Arc Primitives

The *PTArc* class represents an arbitrary arc component with an optional text entry. Arcs cover arbitrary circles and ellipses, as well as segments thereof. Figure 5.7 shows the class diagram for *PTArc* components. *PTArc* primitives implement the *animal.animator.MoveBase* interface, and can thus be used as the base object for arbitrary move effects.
Most of the explanations for *PTPolyline* objects also apply to *PTArc* instances. The large set of access and set methods shown in the diagram maps requests to the local properties of either the arc itself or its text subcomponent. If present, the text is centered on the arc's center.
The access methods are basically unnecessary but allow developers easier access to the properties. A *closed* arc is also called a *pie wedge*. The *setCircle* method allows switching from a circle representation to an ellipse. The difference between these is that circles use an integer as the radius, while ellipses have a *x* and *y* radius represented as a point.
Closed arcs may possess a fill color, while open arcs may exhibit the forward or backward arrows already described for polyline objects. The *angle* and *start angle* determine the shape of the arc in conjunction with its orientation. The orientation may be *clockwise* or *counterclockwise* and can be determined or set using the *isClockwise* and *setClockwise* methods, respectively.
The second set of methods addresses animation effects on arcs. The *getLength* method returns the length of the arc. The *getPointAtAngle* method returns a point on the arc at the given angle. *isAngleInside* determines if a given angle is part of the arc, while *useAsMoveBase* opens the arc and installs a single forward arrow. The *rotate* methods allow the rotation of an arc component

around the point of origin or the passed point. The *translate* method is also implemented but not shown in the diagram as it was inherited from the superclass.

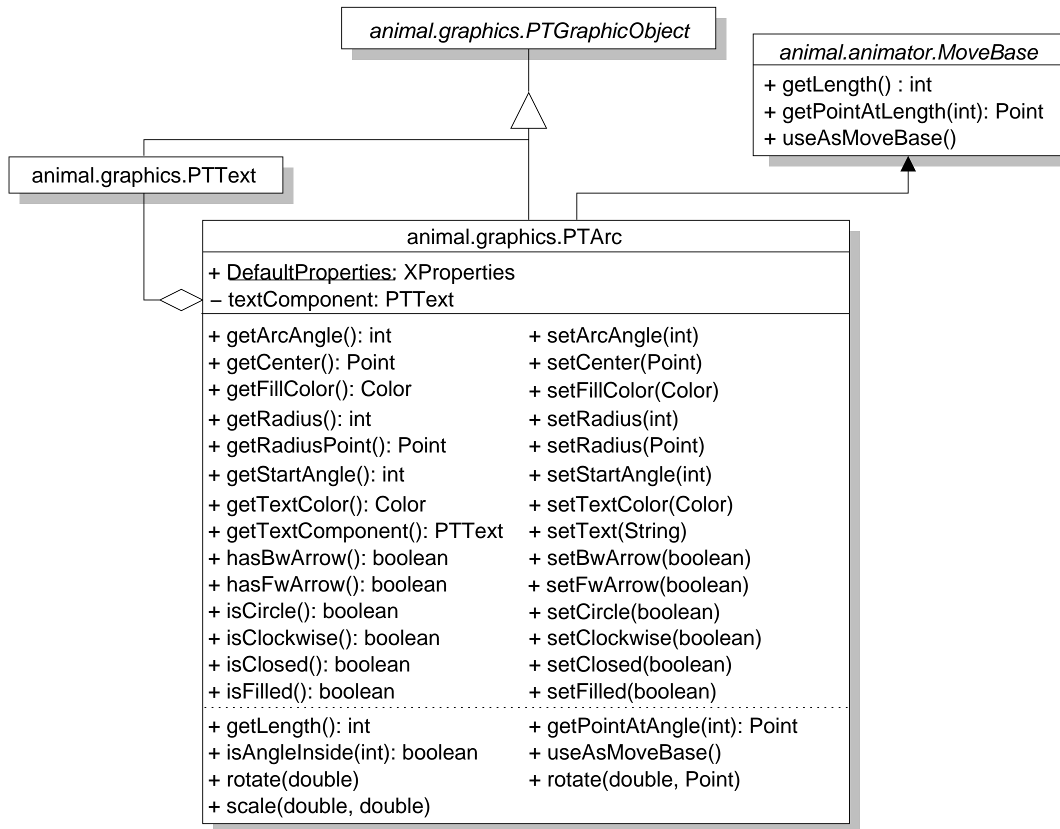

Figure 5.7: UML Inheritance Diagram for the *PTArc* Class

5.3 Animation Effects

All ANIMAL animation effects or "animators" are placed in the *animal.animator* package and extend the abstract class *animal.animator.Animator*. Animation effects that allow offset and duration specification instead extend the abstract class *animal.animator.TimedAnimator*, which is a subclass of *animal.animator.Animator*. The UML class diagram for the *Animator* class in relation to the framework is shown in Figure 5.8. Note that the framework class has been placed in a dashed box, as it was directly extended for added functionality within the prototype system. *Animator* is a subclass of *EditableObject* and therefore also of *PropertiedObject*. It contains three static attributes and a large collection of methods.

The *DefaultProperties* class attribute represents the properties used as default properties for all animator instances. On initializing a new animator, this attribute is declared as the default property set. As outlined in section 4.2.1, this has the effect of providing a common base for all animators as a fall-back position. The *infoFrame* attribute is an extension for adapting ANIMAL to integration into JHAVÉ and is described in section 6.7. The class attribute *registeredHandlers* is a hash table used

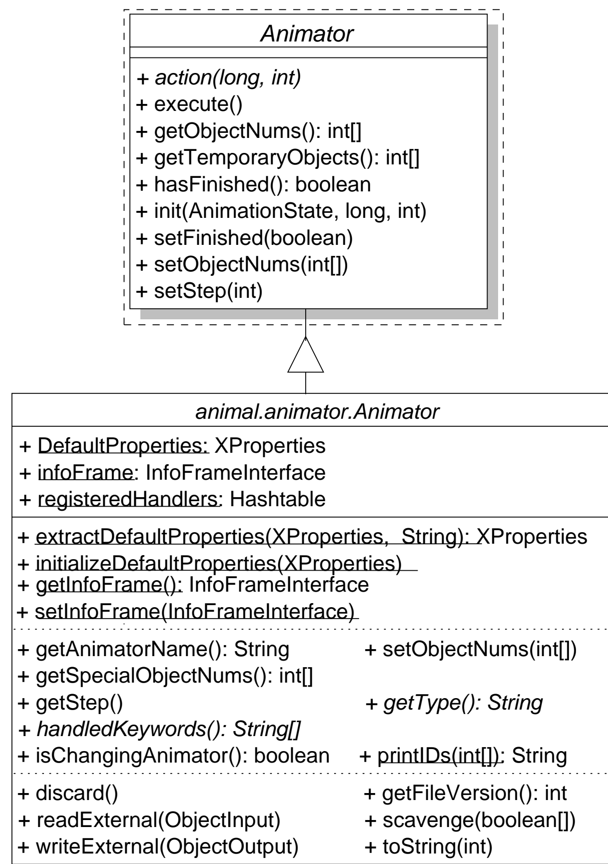

Figure 5.8: Architecture of the Animator Base Class *Animator*

for mapping animation effects to intermediate agents called “handlers”. This process is described in more detail in section 5.4.

For better readability, the large interface of class *Animator* has been segmented using dotted lines. The first segment contains the two class methods *extractDefaultProperties* and *initializeDefaultProperties* used for accessing the *DefaultProperties* class attribute described above. The *getInfoFrame* and *setInfoFrame* methods are used for retrieving or assigning the *infoFrame*. Please refer to section 6.7 for more information about the *infoFrame*.

The second block of methods provides read access to the basic properties of the current animation effect: *name*, setting the array of *objects* on which the effect works, the array of *special objects* used by the animator, and the animation step in which the animator is placed. The *type* of the animator is used for extracting the animator’s properties from the global dictionary using the *getElementsForPrefix* method of class *animal.misc.XProperties*, as described in section 4.2.1. The *handledKeywords* are required for dynamically adding, removing and looking up animation effects. Finally, the *isChangingAnimator* returns true if the animator modifies the objects it works on. The *printIDs* method provides a convenience access to a String representation of the numeric IDs of the animated objects.

The final segment of methods deals with object versioning. It also includes support for the *java.io.Externalizable* mechanism. Although externalization has been deprecated, the support is still present to allow the parsing of older animation files.

The method interface of *Animator* is relatively large for an object-oriented design. This is due to the different interests that have to be covered within the base class: convenience property access and setting, animation effect interface as well as storage and retrieval facilities. Note that all subclasses of *Animator* also inherit the methods and attributes of the classes *PropertiedObject* and *EditableObject*. They can therefore access an internal state representation using properties and have a link to graphical editors. Furthermore, each subclass has a class attribute representing the default properties for all objects of the given type.

The abstract class *TimedAnimator*, shown in Figure 4.12 on page 97, provides the basic functionality for timed animation effects. The class method *copyTimingFrom* for easy transferal of timing information has already been defined in the framework, as well as methods for accessing and setting the *duration* and *offset* of the current timed animation effect can be retrieved and set. The *getMethod* method allows the determination of the concrete subtype of the animation effect. Most animation effects provide more than only one type of effect. For example, a *move* effect may move the whole object or only parts thereof, depending on the underlying type of the object.

The *getStartTimeOrTicks* methods can be used to retrieve the start time of the animation effect. The *isUnitIsTicks* and *setUnitIsTicks* method pair is used for resolving whether the underlying base time unit is milliseconds or internal ticks corresponding to displayed image frames. The abstract *getProperty* method is the main method for resolving timing information. It must determine and return the state of the animation effect at the current point in time. This is then used to determine the state of each affected graphical primitive.

The inheritance structure of the animators provided by ANIMAL is shown in Figure 5.9. The *Show* effect is currently the only available animation effect which does not use any timing information. The *PTGraphicObject* reference belongs to an array of the primitives on which the effect works. As the array is transient, the reference has been omitted from Figure 5.8.

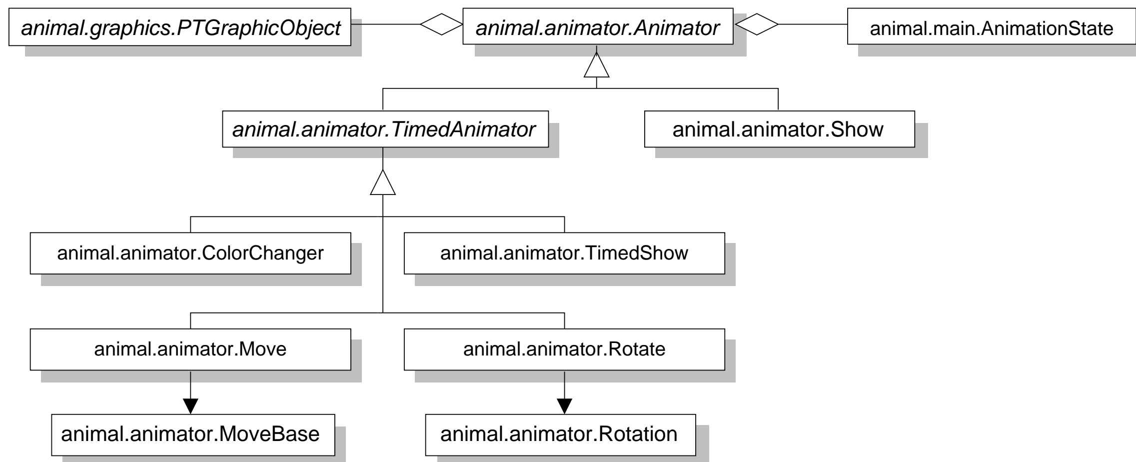

Figure 5.9: Inheritance Diagram for ANIMAL Animation Effects

5.3.1 Show Effect

The *Show* effect implemented in class *animal.animator.Show* is arguably the simplest animator. Its sole task is modifying the visibility of graphical primitives according to the method used. Figure 5.10 shows ANIMAL's *Show* animator in relation to the framework. Apart from the methods inherited from *animal.animator.Animator*, the class has a private boolean attribute that determines the visibility of the selected primitives, with `true` for visible and `false` for hidden. The value of this attribute can be retrieved by invoking *isShow* and set by *setShow*.

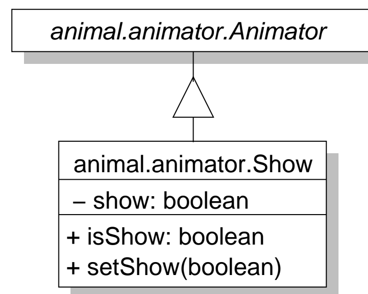

Figure 5.10: Inheritance Diagram for *Show* Animators

Figure 5.11 presents a small example of the *Show* effect. In the first step, a coordinate axis with a mathematical function is displayed by invoking the *Show* effect with the object numbers, which we assume to be 1, 2 and 3. In the second step, two of the three primitives are hidden by a second *Show* effect that has been set to *hide* mode by invoking `setShow(false)`. Finally, the third step shows the objects again by adding another *Show* effect. The boxes around the diagrams are inserted to show the boundary of each step.

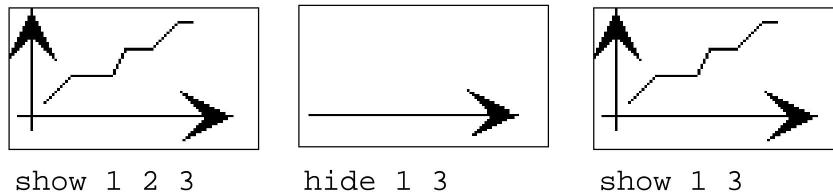

Figure 5.11: Example *Show* Effect

5.3.2 Timed Show Effect

Figure 5.12 shows the structure of the *TimedShow* animator in relation to the framework. Just as the *Show* animator, the simple structure of the requires only two additional methods for setting and retrieving the visibility state of the selected primitives beyond those introduced in the framework super classes *animal.animator.Animator* and *animal.animator.TimedAnimator*.

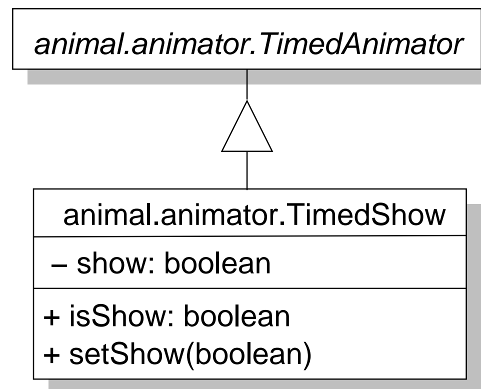

Figure 5.12: Inheritance Diagram for *TimedShow* Animators

For an example of the *TimedShow* effect, please refer to Figure 5.11. Note that the default *TimedShow* effect offers no fading, so that the *duration* has no visible effect. Thus, there is only one difference between a *Show* and a *TimedShow* animator: the *Show* animator is always executed at the start of the animation step, while the *TimedShow* animator is only executed when its start time has been reached. This fact can be used for building animation steps that show a number of items which appear one after another, as commonly done in Microsoft PowerPoint™ presentations.

5.3.3 Color Change Effect

Figure 5.13 illustrates the relationship between the color changer animator in the framework and in ANIMAL. To present a good interpolation of intermediate color values for non-instantaneous color change effects, the original colors of all object must be stored. Alternatively, the visualizer may also use a common start color for all animated primitives. Another pair of methods is used for retrieving and setting the target color.
The color interpolation is achieved by calculating the intermediate color that bisects the line connecting the start and target color at the current execution point for each animated object. Figure

5.14 illustrates this process for a change from black to white for an arbitrary color property. The figure includes the interpolated color values for the following animator execution states, with the double precision timing given in parentheses: 0%, equal to the animation effect start (*0.0*), 20% (*0.2*), 50% (*0.5*), 85% (*0.85*) and 100%, equal to the animation effect end (*1.0*). The marked colors are returned by the *getProperty* method defined on page 98.

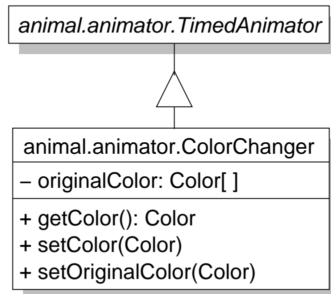

Figure 5.13: Inheritance Diagram for *ColorChanger* Animators

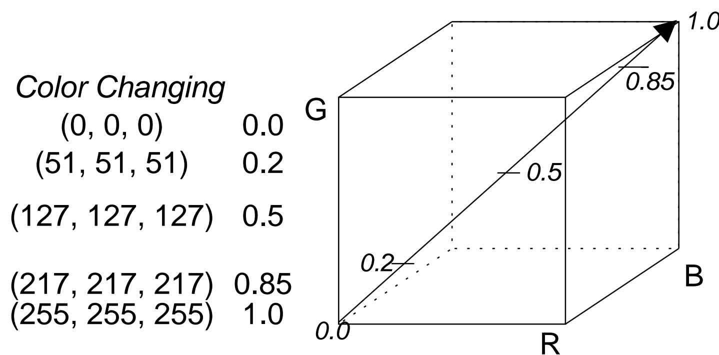

Figure 5.14: Color Interpolation in the *ColorChanger* Animator

Figure 5.15 shows an example color change application grabbed from ANIMAL. The color of the text embedded in the box changes over time from black to white, assuming appropriate grey shades during the operation.

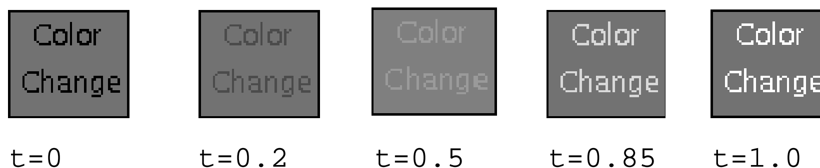

Figure 5.15: Example Color Interpolation in the *ColorChanger* Animator

5.3.4 Move Effect

Figure 5.16 illustrates the *Move* animator in ANIMAL. Apart from declaring the animator as a timed animator, the only information needed are the length of the underlying move base and its numeric ID. ANIMAL's Move animator offers a fluent interpolation of move points for animators possessing a duration. This is achieved by calculating the intermediate move points along the move base object once for the current execution point. Figure 5.17 illustrates this process for a move along a polyline. The figure includes the interpolated points for the following animator execution states, with the double precision timing given in parentheses: 0%, equal to the animation effect start (*0.0*), 20% (*0.2*), 50% (*0.5*), 85% (*0.85*) and 100%, equal to the animation effect end (*1.0*). The marked points on the polyline in Figure 5.17 are also the returned value of invoking the *getProperty* method for the associated values, as defined on page 98.

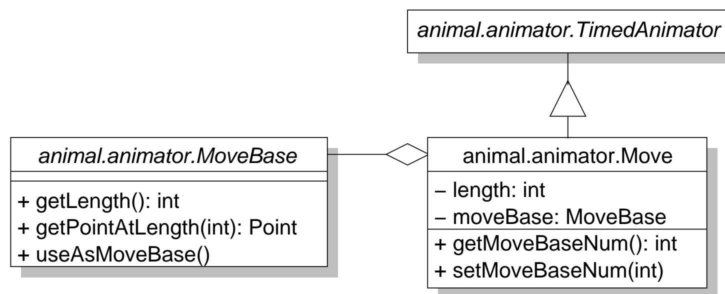

Figure 5.16: Inheritance Diagram for *Move* Animators

The *MoveBase* interface is responsible for determining the point on the underlying move base at a given point in time. For this end, the execution state of the effect measured in the interval $[0,1]$ is multiplied with the length of the move base, as retrieved by *getLength*. The rounded result value is then inside the interval $[0, getLength]$ and used for determining the target point. Note that this approach is *not* limited to using lines as a move base; the developer can easily also implement the functionality for arc components. This is done in ANIMAL, so that moves can be accomplished along polylines or arcs.

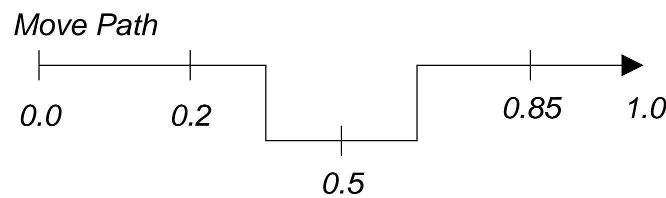

Figure 5.17: Timing of *Move* Animators

Figure 5.18 shows an example move application that moves a square along the path shown in Figure 5.17.

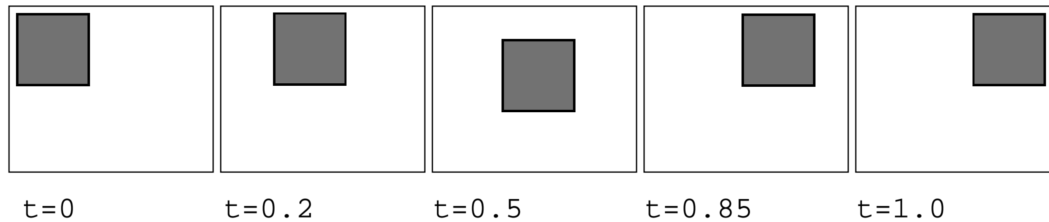

Figure 5.18: Example *Move* Animator

5.3.5 Rotate Effect

Figure 5.19 illustrates the *Rotate* animator in ANIMAL. We added methods for setting and retrieving the center point and the rotation angle. The rotation itself is encapsulated in a *Rotation* object that allows accessing the rotation angle and center. ANIMAL's rotate animator offers a fluent interpolation of rotation angles for animators possessing a duration. This is achieved by calculating the intermediate rotation angles along the circle spanning the full rotation angle once for the current execution point.

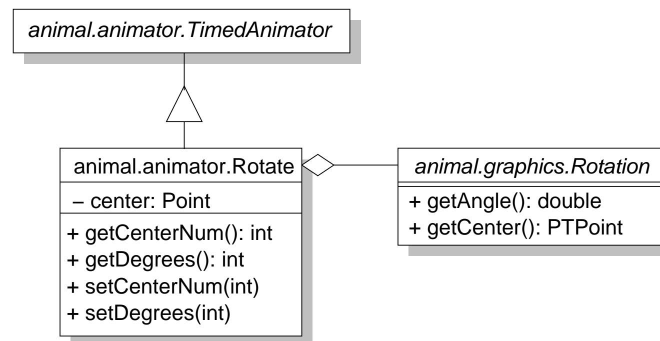

Figure 5.19: Inheritance Diagram for *Rotate* Animators

Figure 5.20 illustrates this process for a move along a polyline. The figure includes the interpolated points for the following animator execution states, with the double precision timing given in parentheses: 0%, equal to the animation effect start (*0.0*), 20% (*0.2*), 50% (*0.5*), 85% (*0.85*) and 100%, equal to the animation effect end (*1.0*). The marked angles and thus also the associated points on the angle are returned by the *getProperty* method for the associated values, as defined on page 98. Figure 5.21 shows an example rotate application that rotates an arrow on its end point along the angle shown in Figure 5.20.

5.4 Transformation Handlers

As stated in section 4.6, transformation handlers act as agents for negotiation between primitives and animation effects. The primitives provide a straightforward implementation of the underlying graphical element. This especially includes methods for accessing the state using *properties*, the

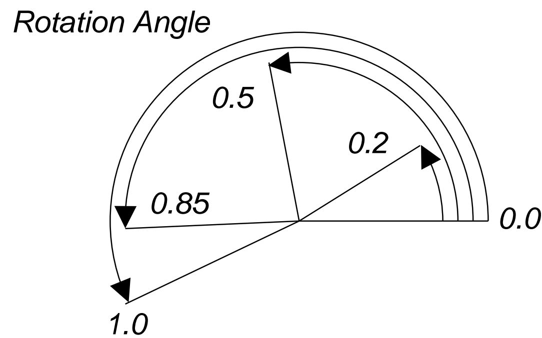

Figure 5.20: Timing Resolution for *Rotate* Animators

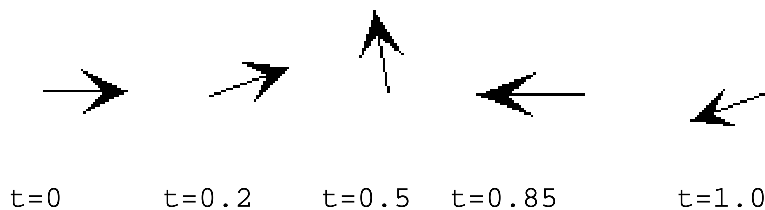

Figure 5.21: Example of *Rotate* Animators

determination of the primitive's *location* and *bounding box*, and the *paint* method for displaying the primitive on the graphics context. Additionally, they provide at least the *translate* method for changing the location, and may also offer the other basic transformations *scale* and *rotate*.
Animation effects model the execution of generic animation effects. They can retrieve and set the affected object and may possess a timing specification containing both an offset from the associated animation step's start and a duration. The core functionality is wrapped in one method each for initializing, performing and finishing the effect, respectively. Section 5.3 examined the animation effects in more detail. From the handler's point of view, it is sufficient to know that the animation effect can determine the percentage state of its execution and retrieve the appropriate intermediate value, such as a target color or move location.
The transformation handler is responsible for two central aspects of performing animation effects. Firstly, it is responsible for determining the list of concrete transformation subtypes available for a given primitive and a given animator. Secondly, it maps the given effect to a set of primitive transformations for each affected primitive.
The first task is achieved within the *getMethods(PTGraphicObject, Object)* method present in all handlers. Given a concrete instance of the handled object type and an effect-parameterizing object, the handler returns a vector of possible animation effect subtypes, as shown in Figure 4.14 on page 101. Note that the list of available effects can depend on the properties of the primitive. For example, if the nodes of a polygon can be moved individually, the set of possible method names depends on the number of nodes of the primitive. By passing the primitive as a parameter, such properties can be determined easily.
The mapping of animation effects to primitive method calls is performed in the *propertyChange(PT-GraphicObject, PropertyChangeEvent)* method. Given the primitive on which to work and the

transformation encoded as a *java.beans.PropertyChangeEvent*, a set of method invocations on the primitive are performed that result in the execution of the animation effect. The *Property-ChangeEvent* encodes the name of the animation effect subtype, as well as the current and target value of the target object state. The precise data stored in the target state depends on the animation effect. For example, *Move* effects encode the point reached at the current transformation time.

The transformation handlers as defined in section 4.6 are perfectly up to the task set them. We have not found it necessary to provide any extension or customization within the course of implementing the prototypical AV system, apart from filling in the actual transformation types possible for each element. Therefore, we will not delve into the implementation code for handlers.

ANIMAL also faithfully follows the animation representation structure described in section 4.7 using a single *Animation* object modeling the animation built on the interaction of animators, primitives and primitive handlers. The *animal.main.Link* class represents the links between successive animation steps.

In the following sections, we will examine the GUI front-end offered by ANIMAL for displaying and editing animations.

5.5 ANIMAL's Animation Display GUI

ANIMAL's animation display GUI is by definition the most "visible" component for users. We have therefore taken special care to offer relevant and helpful features for users and visualizers employing the display GUI for testing their animations.

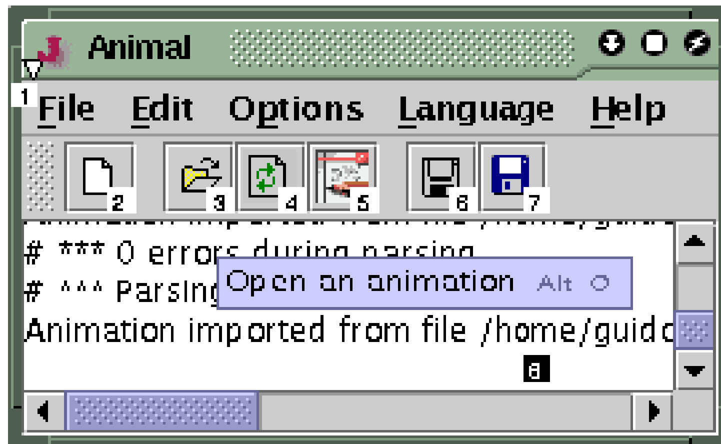

Figure 5.22: ANIMAL's Main AV Frame

Figure 5.22 shows the main ANIMAL frame. The numbers placed in colored squares are not part of the frame; they were inserted here for better referencing of features. Similar numbering will appear in most other GUI components under consideration. The menu bar (1) contains a *File* menu with the usual operations for loading, storing or creating a new animation. The *Edit* menu is used for showing or hiding the individual elements for the display and editing GUI. *Options* provides a central access for configuring ANIMAL, including switching between two- and three-button mouse types. The *Language* menu contains a list of localized GUI front-ends available for the installed

system. By default, it contains *English* and *German* language support. Finally, the *Help* menu offers the usual access to the built-in help facilities.

The tool bar below the menu also contains buttons for the most common file operations: *new animation* (2), *load* (3), *reload* (4), *enter scripting code* (5), *save* (6) and *save as* (7) which stores the animation under a different file name. The text area (8) below the tool bar lists status messages for ANIMAL, especially the load and save status.

Figure 5.23 shows the animation display window used in ANIMAL. The top of the window contains a pair of sliders for controlling the display speed (1) and magnification (3), as well as two buttons for resetting the values to 100% (2, 4). The main part of the display is taken up by the animation canvas (5) which presents the animation at the chosen speed and magnification scale. The animation shown in the screen shot is a joint effort of four students of the University of Wisconsin Parkside in animating a branch-and-bound dynamic knapsack algorithm. The display has been scaled to slightly less than 50% to fit the window on the page, causing some font artifacts.

The bottom part of the animation window contains the central animation control bar (6) and the animation step controls (7, 8). The text field (7) can be used for entering an arbitrary animation step number. If the animation step exists, the animation jumps to it; otherwise, it remains at the current step and resets the displayed value. The slider (8) displays the percentage of the animation currently shown and can also be used for fast-forwarding or rewinding the animation. Both components (7, 8) are squeezed due to the page constraints, as the images used in the control tool bar are not resizable.

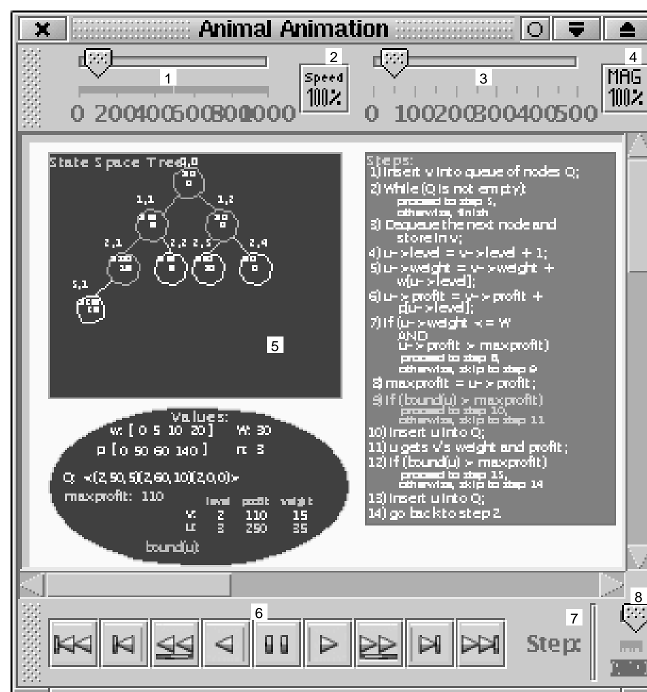

Figure 5.23: ANIMAL's Animation Window

Figure 5.24 shows the animation control tool bar included in the animation window shown in Figure 5.23. The buttons, from left to right, carry the following meaning, in keeping with Figure 4.19 on page 108:

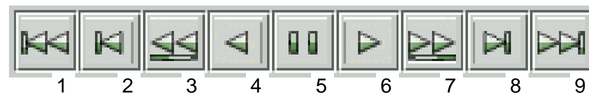

Figure 5.24: ANIMAL's Control Tool Bar

1. jump to the start of the animation,
2. jump to the start of the previous animation step,
3. play the animation in reverse slide show mode. The timing delay between successive steps is respected if present; otherwise, the delay set in the configuration is used,
4. play the current animation step backwards by employing all timing information for animators,
5. pause the animation, especially relevant when employing either slide show mode,
6. play the current animation step by employing all timing information for animators,
7. play the animation in slide show mode, respecting all defined delays between subsequent steps and using the configured default delay otherwise,
8. jump to the start of the next animation step,
9. jump to the end of the animation.

Finally, Figure 5.25 shows the implementation of the Animation Structure View in ANIMAL. As the structure represents an animation time line, we call the window *Time Line Window*. All available step labels are gathered and displayed in a list inside the window, along with the associated step number. If the user clicks on an entry in this list, the animation jumps to the associated step. Note that it does not matter if the current animation step lies before or after the selected labeled step, as ANIMAL can easily handle reverse animation playing. If the selected labeled animation step is identical to the current animation step, no effect will be visible.

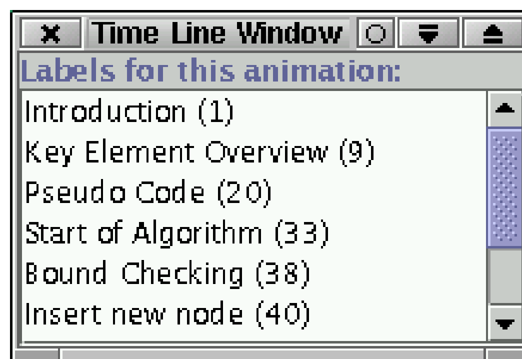

Figure 5.25: ANIMAL's Implementation of the Animation Structure View

5.6 ANIMAL's Editing GUI

For our prototypical AV system, we have decided to support GUI-based animation generation and editing. This feature is relatively uncommon for AV systems, especially when combined with other generation or editing approaches, as demanded by requirement **VR5**. Therefore, adding such a front-end is a good exercise and also adds a special flavor to our prototype.

The editing front-end of ANIMAL consists of four windows: the *drawing window* for generating primitives, the *animation overview* for managing animation effects and animation steps, the *Main AV* window shown in Figure 5.23 on page 136 and the *Animation Structure View* shown in Figure 5.25 on the page before. As the latter two windows have already been described, we focus on the drawing window and the animation overview.

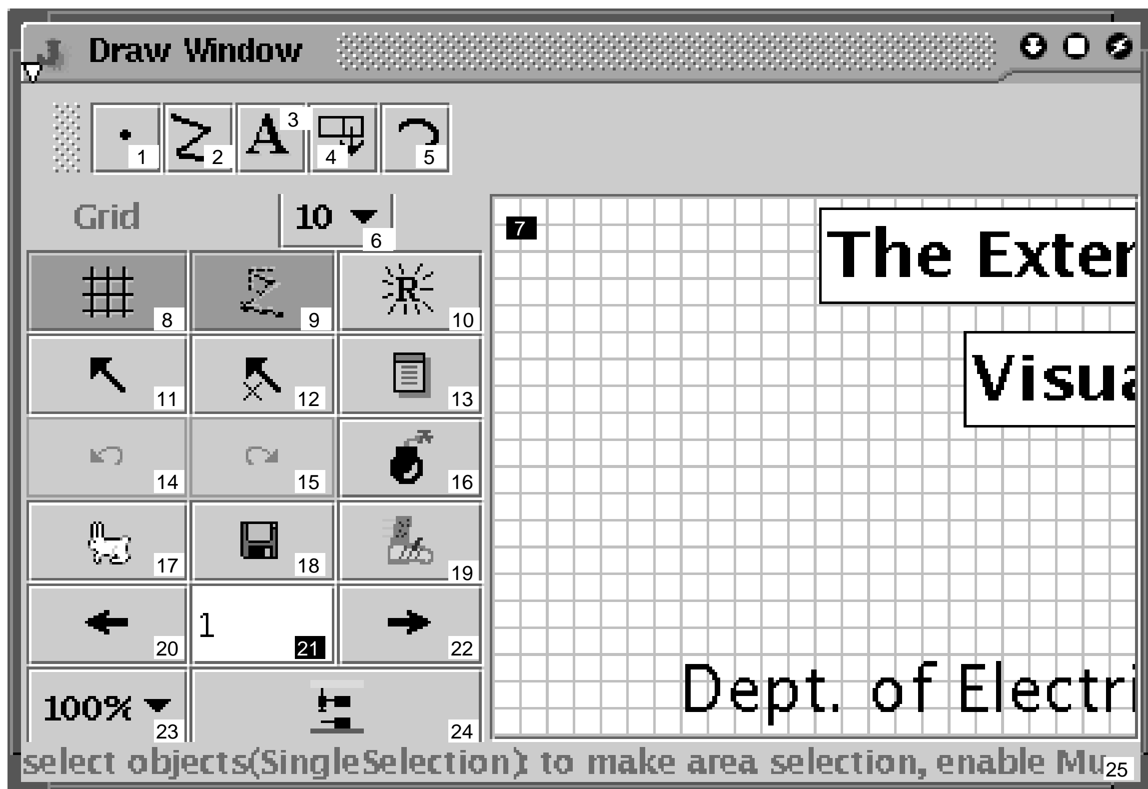

Figure 5.26: ANIMAL's *Drawing Window* GUI

Figure 5.26 shows the drawing window used by ANIMAL. The top button row offers access to specific editors for the currently supported primitives *point* (1), *polyline / polygon* (2), *text* (3), the *list element* extension (4) described in section 6.2.2, and *arc* elements (5) including circles, ellipses and segments thereof. The grid control (6) toggles the display width of the grid between the values *none, 5, 10, 20, 25* and *50*, measured in screen pixels. The main canvas (7) is populated by the primitives visible at the current animation step, as well as the optional grid. The canvas uses drag and drop for each primitive. Drag points for dragging the full object are placed at the middle of

each line. Editing points for moving individual primitive nodes are placed at each node and allow easy manipulation of node locations.

The first button row contains two buttons for toggling grid lock mode (8) and the display of temporary objects, such as move base primitives (9). Grid locking automatically adjusts primitive locations selected by user actions to the closest intersection of grid lines. The refresh button (10) causes a redraw.

The second row starts with the *select objects* button (11) used for toggling the usage mode from generation to editing. The remaining two buttons toggle the selection of multiple objects (12) and the editing mode (13). The latter, if active, pops up special editor window for the currently selected primitive or primitives. Figure 5.27 shows an example editor for *arc* primitives.

Figure 5.27: *Arc* Primitive Editor Window

The buttons (14, 15) offer *undo* and *redo* operations. Button (16) deletes the currently selected primitive or primitives. If at least one of the selected primitives is used in an animation effect, the user will be prompted to confirm the deletion. The hare (17) offers a cloning operation that is very useful if a set of similar objects has to be generated. The cloned object is placed slightly below and to the right of the original object. The disk button (18) is used to write the modified animation content back to the animation and all associated windows. It can be used to update the animation window, as this is not performed by default for performance reasons during the editing process. Button (19) runs the current animation step in the animation window, opening the window first if necessary.

The buttons (20) - (22) let the user select the current animation step to be edited, by going backwards (20) or forward (22) or by entering the target step number (21). The selection list (23) offers five different magnification scales for drawing: 50%, 71%, 100%, 141% and 200%. The general configuration menu that also allows adjusting the canvas background is activated by a button (24). Finally, the status line (25) describes the actions currently possible.

The *Animation Overview* window shown in Figure 5.28 is used for generating and editing the structure of the animation. It thus contains buttons for adding new animation effects (1-5) and animation steps (8, 9). The animation effect buttons shown are, from left to right, *Show* (1), *Move* (2), *Rotate* (3), *Color Change* (4) and *Timed Show* (5). The main canvas of the window contains two different types of entries: animation steps (6) and animation effects (7). Double clicking on an element brings up the associated editor. The right mouse button can also be used to activate a context menu. The maintenance buttons at the bottom row allow inserting a new animation step

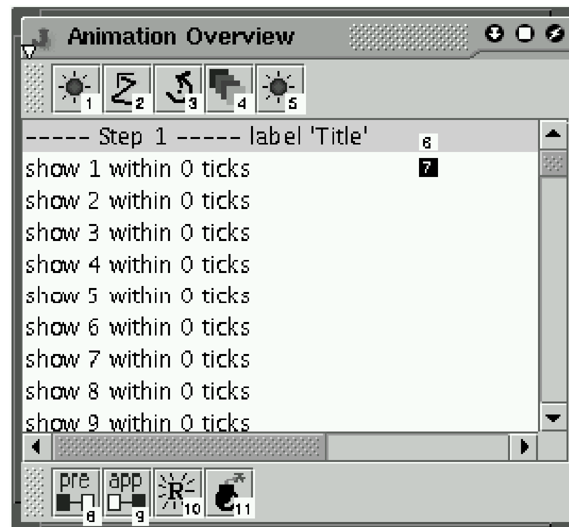

Figure 5.28: ANIMAL's *Animation Overview* GUI

before ("prepending", 8) or after the current step ("appending", 9), refreshing the display (10) and deleting the selected element (11). If the element to be deleted is an animation step with at least one animation effect, the user has to confirm the deletion before it is performed.

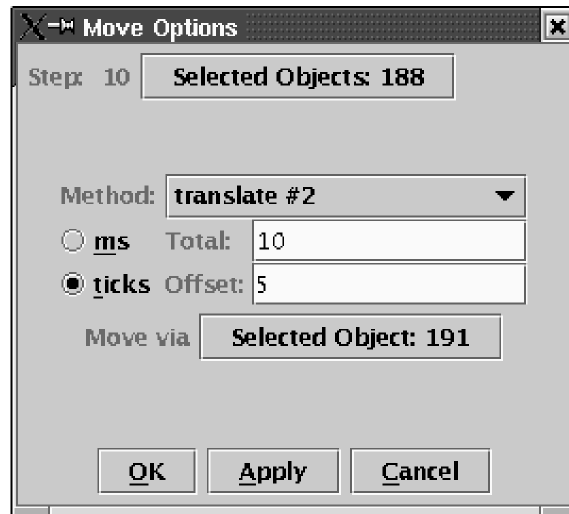

Figure 5.29: ANIMAL's *Move Options* GUI Editor

Figure 5.29 shows an example editor window for the *Move* effect. The name of the animation effect and the current animation step are shown at the top of the window, along with the selected animation objects. The currently selected animation subeffect method *translate #2* translates only the second node of a polyline or polygon primitive. The list of possible subeffects is generated by the transformation handlers of the selected object(s), as described in section 5.4. The two radio

buttons at the middle of the window toggle between real time (*ms*) and virtual frame-based time (*ticks*). The total duration and offset of the animation effect using the current time unit can be specified in the two adjacent text fields. Finally, the temporary object used as the move base is shown, along with the standard trio of window buttons *OK, Apply* and *Cancel*. *OK* acts as *Apply* by applying any changes to the animation, but additionally also closes the editor window.

5.7 Import and Export

The import and export facilities offered by ANIMAL closely follow the specification of the framework in section 4.9. The standard Java file selector with adapted filter selections is used for querying the input or output file name and the target format. The appropriate import or export filters is then instantiated and initialized by dynamic loading according to the configuration and accessed using its MIME type.

Exporting can be configured to the user's tastes. The exact degree of configurability depends on the chosen target output format. Figure 5.30 shows the standard export mode choice dialog. The user can choose between exporting the complete animation or only those steps selected in the list at the bottom. Dynamic export as opposed to static snapshots contains the full dynamics of the animation effects. If the export format is static, a sequence of appropriate elements may be generated, for example resulting in a set of images written for a single animation step. The magnification can be adjusted similarly to the drawing window to 50%, 71%, 100%, 141% or 200%, as well as to the magnification used in the animation canvas. The selection list at the center right also allows the entry of arbitrary percentage values. Keep in mind that the precise export capabilities and the visual quality of the output depend on the target format and the export filter implementation, rather than on ANIMAL's display capabilities.

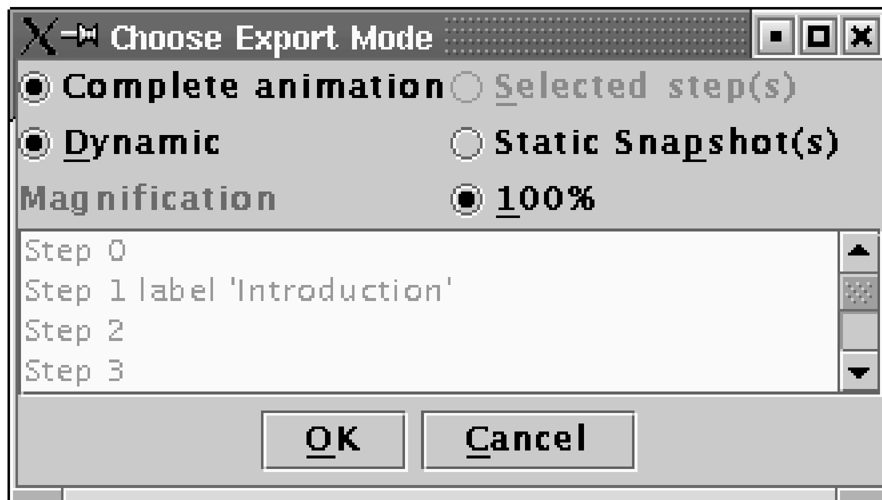

Figure 5.30: ANIMAL's Export Mode Chooser Window

5.7.1 ANIMALSCRIPT

As a special elaborate case of an import filter, we regard the support for the built-in scripting language ANIMALSCRIPT. ANIMALSCRIPT animations are specified in a simple ASCII-based format. Each line of a valid ANIMALSCRIPT animation contains exactly one command or comment.

ANIMALSCRIPT is extensible and thus allows easy adaptation and extension if desired features are not provided. The concrete extent of the features supported by ANIMALSCRIPT is retrieved at start-up from a configuration file and used to dynamically build the whole parser component for the scripting language. This process is the same as described in section 4.2.

Apart from supporting the base primitives provided by ANIMAL, ANIMALSCRIPT also supports linked list elements, arrays and source code embedding including highlighting and indentation. Modular ANIMALSCRIPT files can be dynamically included using a special *embed* command. To make full use of the possibility of placing runtime information into a reusable module, ANIMALSCRIPT allows the declaration of variables which can be placed in text components and are replaced by their current value at load time.

All coordinates used for specifying primitives or animator parameters can be relative to other, previously defined locations. One application area where this is important is the internationalization of animation content offered in ANIMALSCRIPT. Here, the whole animation may be translated into the target language chosen by the user. Provided that internationalization is embedded in the animation file and that all primitives use relative placement, the screen layout will still be fine, even if the length of the components differs between languages.

ANIMALSCRIPT provides facilities for performing interactive predictions (**UR10**). Currently, the feature is employed only in conjunction with the JHAVÉ system [131, 181]. ANIMALSCRIPT can be configured easily, as all components are loaded dynamically based on a configuration file. The configuration file is first looked for in the current directory, followed by the *CLASSPATH* entries, the *jar* file containing the ANIMAL distribution and the ANIMAL home page. Therefore, users and visualizers can easily switch between different ANIMALSCRIPT configurations by simply changing the directory from which ANIMAL is started.

ANIMALSCRIPT is simple to use but powerful. ANIMALSCRIPT input is typically a simple ASCII file consisting of a number of lines. Each line may contain either a comment marked by a hash mark #, or exactly one ANIMALSCRIPT command. The first word of each command line is the *keyword*; this must always be unique to allow both easy parsing and dynamic extension.

ANIMALSCRIPT commands typically have a number of parameters, some of which may be optional. The syntax of ANIMALSCRIPT is defined by a set of extended Backus-Naur rules using [ ] for optional parameters and {} for elements that may be repeated an arbitrary number of times. Further details about the syntax of ANIMALSCRIPT can be found in [179, 178, 177] and on the ANIMAL Home Page [169].

ANIMALSCRIPT provides some extended functionality that offers more comfortable access to "hidden" features of ANIMAL. The simplest extended command is the *group / ungroup* directive which tells ANIMALSCRIPT to group all object IDs passed in as one object for the next steps, and thus frees the user from much typing.

List linking allows the user to set or clear the pointers of list elements with a single command. In keeping with pointer semantics, the user only has to specify the target object by its ID, and does not have to worry about the exact location of the pointer. This takes a lot of administrative book-keeping from the user, compared to other systems where the precise target location must be given.

ANIMALSCRIPT provides full array support for both vertical and horizontal alignments. This includes the installation and moving of *array index pointers* with an optional label as well as the *put* and *swap* operations. As the last two operations may have an effect on the width of the array cells following the first affected element, the layout of the array is updated, possibly including moving existing array pointers. Thus, sorting an array using ANIMALSCRIPT is very easy to do, as it consists mostly of a sequence of *arraySwap* and *movePointer* commands for changing the elements and moving the (optional) index pointers.
ANIMALSCRIPT also provides full *source code embedding*. For this purpose, the user declares a *codeGroup* with some formatting options, such as the font to be used, and gives the upper left corner. Individual code lines or code line elements can then be added to the code group, including an optional indentation depth. The code is laid out dynamically. *Source code highlighting* is easily accomplished by commands that allow the user to highlight a given line or line segment as either the current command or the current context.
All elements may be positioned in one of the following ways:

- using absolute coordinates such as `(10, 20)`,
- relative to an edge of another object's *bounding box*,
- relative to an *arbitrary node* of a given (polyline) object,
- relative to the *baseline* of a given text,
- or relative to the *last* location used.

One of the most helpful features of ANIMALSCRIPT during animation development is the *echo* command. This command can be used to print the location of objects (usually the upper left corner) or their bounding box. It can also be used to print the numerical IDs of grouped objects, or show the IDs of all currently visible objects. For assertions, *echo* can print the current value, and it can also print out any given text to the standard output. Thus, *echo* is a very powerful help during animation development in pinpointing the precise reason why something did not go the way it was supposed to. Of course, *echo* can also be turned off.
In order to address the widest possible target audience with animations, all ANIMALSCRIPT text entries may be given either as a literal string or as localized text. The localization can be embedded in the file or stored in separate language resource files. Embedded localized text is placed in parentheses (), and each component is prefixed with the *language code* followed by a colon, for example as (de: "Ja" en: "Yes" fr: "Oui"). On loading the animation, ANIMALSCRIPT prompts the user for the language to be used for the animation's display, and will layout all components accordingly. Due to the possibility of using relative coordinates, it is usually sufficient to generate *one* animation with multiple languages embedded. Additional extended features of ANIMALSCRIPT include *assertion checking, block charts, variables* and *stop-and-think questions* as used in the JHAVÉ [131] system now also using ANIMALSCRIPT.
While ANIMALSCRIPT already provides many helpful features for animation generation, users are bound to find some features they consider necessary or at least helpful which are not yet covered in the extended ANIMALSCRIPT functionality. For this purpose, we have designed ANIMALSCRIPT in a way that allows easy extensions by other programmers.
Adding new graphic primitives or animation effects requires a certain knowledge of Java and programming in general. ANIMALSCRIPT comes with an extensive set of parsing support methods, so

that parsing the input data is comparatively easy to accomplish. This includes support for parsing keywords, optional parameters, comments, color specification, absolute or relative coordinates and localized texts. Usually, each element in a new command definition requires one method call in the parsing support class. The ANIMALSCRIPT components can also be configured dynamically using the *ComponentConfigurer* shown in Figure 4.8 on page 82.
The auxiliary *animalscript.sourcerer* API generates the parser code from a given extended Backus-Naur-Form. Using this API, programmers of ANIMALSCRIPT therefore only have to program the actual graphical and display operations.

5.8 Summary

Both graphical primitives and animation effects in ANIMAL employ properties for modeling most of the object state. One exception to this rule are the numeric identification of graphical primitives and the internal representation of primitive locations. Both are accessed very often in a typical session, including mathematical operations in the context of location transformations. Converting the location between *Point* and *String* format for each access introduces an significant but completely unnecessary overhead. Recall that the location of every visible primitive must be accessed at least once for each *paint* invocation that updates the graphical display.
The graphical primitives supported by ANIMAL are *point, polyline / polygon, text* and *arc*. The visualizer can toggle between polylines and polygons by inverting the value of the *closed* property. Contrary to some other systems as *JSamba* [196], polylines are not restricted in the number of nodes they have, apart from requiring at least two nodes to form a valid line. Polygons can be filled with any color, while polylines may posses arrows at either or both ends. Text primitives are restricted to the three Java system font families *Serif, SansSerif* and *Monospaced* to avoid portability problems arising when specific fonts such as *Arial* are not installed on the user's system. Finally, arcs may be elliptical or circular. If closed as a pie wedge, they may also be filled; open arcs may possess forward or backward arrows. Additionally, a text can be placed inside the arc and, if present, is automatically aligned on the arc's center.
All graphical primitives have a color, depth, name, numeric ID, location and bounding box that completely encloses the primitive. They are also required to implement the *translate(int, int)* method that moves the object by the offset $(\Delta x, \Delta y)$. The *paint* method is invoked for generating the display of a given primitive.
ANIMAL animators are separated into timed and untimed animators. The only untimed animator is currently the *Show* animator for toggling the visibility of the chosen objects. All animators take an arbitrary non-empty set of objects as target objects. Timed effects specify the animator's duration and offset from the animation step, using either real time measured in milliseconds or virtual time measured by the number of frames to display. Base ANIMAL provides the timed animation effects *TimedShow*, *Move*, *Rotate* and *ColorChange*.
With the exception of *TimedShow*, the intermediate state of the animated objects is interpolated for each frame. For this purpose, ANIMAL employs three methods within each animator, as well as a separate handler agent that maps the operations to the underlying primitives. The *init(AnimationState, long, double)* method initializes the animator and sets the start time to the time passed in as the current system time and frame number. The *action(long, double)* method determines the target state of the primitives during the execution of the animator and forwards this state to the handler. Finally, the *execute()* method is invoked when the animation effect has finished. It displays the final

state of the animator and removes the animator from the list of animators to examine.
All animators are completely independent of the graphical primitives on which they work. Their flexibility thus allows for easy reuse and recombination. The coordination of animators and primitives is the responsibility of the *handler* agents.
Handlers provide two methods, called *getMethods* and *propertyChange*. The *getMethods(PTGraphicObject, Object)* method receives the concrete primitive on which an effect is to take place and an object that represents the animation effect parameters. Based on this second object, the method builds a vector of appropriate animation effect subtypes. This generation may also take specific properties of the primitive into account. For example, if a *Move* effect is requested, the vector returned may include the following move targets: full object, single specific node, or set of nodes. The latter two subtypes require the knowledge of the node count encoded in the primitive. Animation effects receive a vector of animation subtypes for each object on which they work. The vectors are pruned by building an intersection. The resulting elements represent the possible animation subtypes applicable to the selected objects.
The *propertyChange(PTGraphicObject, java.beans.PropertyChangeEvent)* method receives the current and target state of the animation effect, as well as the animation subtype, in the PropertyChangeEvent parameter. Based on the animation subtype, the new state is examined and mapped into a set of method invocations on the primitive passed in so that the primitive's display is adapted to the requested effect.
ANIMAL thus employs a strict separation of concerns. The graphical primitive classes represent only the current state of the underlying primitive including display capability in the *paint* method. However, they have no knowledge about animation per se. The animators, on the other hand, only determine appropriate animation subtypes on a String base, and can interpolate intermediate animation states. However, they do not possess any knowledge of the primitives on which they work. Finally, the handlers may not need to access the current concrete primitive instance in most cases when determining the possible animation effects. Exceptions are animation effects which depend on the current state of the primitive, for example the number of nodes or whether the underlying object is closed. Mapping the requested animation effect onto the primitive requires one or more primitive method invocations that typically require little if any knowledge of implementation details.
The front-end used by ANIMAL for displaying the animation consists of the main AV frame, an animation window and a structure view of the animation. Apart from the usual file operations, the main AV frame centrally controls the visibility of the other windows and provides internationalization for all GUI components. The animation's structure or "time line" is shown in a separate window. Clicking on a labeled step shown in this window updates the display to the associated animation step. The animation window offers facilities for changing the display speed and magnification on a percentage scale. Additionally, the video player-like control incorporates playing, stepping and displaying in a slide show mode for *both* animation directions, as well as jumping to the animation start or end.
The animation generation and editing front-end contains a drawing window for generating primitives with full drag and drop support, grid control, cloning, redo and undo. The animation effects and steps are controlled in a separate animation overview window. Alternatively to visual generation in the GUI, the animation may be specified using the built-in animation scripting language ANIMALSCRIPT. This language incorporates special features such as relative placement, animation content internationalization and debugging. Special subtypes such as arrays and list elements are also embedded in ANIMALSCRIPT, together with the typical operations used on the data types.

The import and export capabilities of ANIMAL are built from independent components that act as a file filter for one format each. The concrete selected filter instance is determined from the MIME type of the input and loaded dynamically. Export capabilities currently include the file format specified by ANIMAL, ANIMALSCRIPT and image or XML export. If the user request exporting the animation with full dynamics, a set of images is stored for each animated step, instead of a single snapshot showing the start of the step.

The main contributions of the ANIMAL AV system prototype can thus be summed up as follows. ANIMAL is dynamically extensible and uses the concepts of the underlying framework to configure several component types at start-up or even while the system is running. This concerns primitives, animators, import and export filters, and ANIMALSCRIPT components. ANIMAL also presents a (for AV systems) highly unusual combination of multiple generation and editing approaches, namely *visually* using the GUI front-end, by scripting in ANIMALSCRIPT, and by API invocations in a separate API that is nearing completion. Thanks to the highly expressive framework base, ANIMAL is to our knowledge also the only general-purpose AV system that can handle random access navigation through any animation, including rewinding and dynamic reverse playing.

All GUI components can be dynamically translated on demand and on the fly by a single menu item selection, thanks to the powerful operations provided by the underlying framework, as described in section 4.2.3. The import and export filter interface is kept deliberately open for later additions. It may also be used to add another animation generation approach by writing a code interpretation "import filter" that translates code of a certain programming language into ANIMALSCRIPT or directly into ANIMAL animation primitives and effects. An example project of this type was recently performed in a Compiler Construction course at the University of Zittau / Görlitz, Germany; alas, the results have not been published.

ANIMALSCRIPT offers the standard features visualizers can find in other AV scripting language, but also several advanced commands. Among these are the *echo* command that is very helpful for debugging the code, powerful relative placement directives, internationalization support for the animation code, and interactive prediction. Variables and loadable modules are also offered and allow for example the extraction of common front- and back-ends for sorting algorithms including placeholders for the actual number of calculation steps performed.

The display front-end of ANIMAL offers dynamic adjustments of the display speed and magnification, hyperlink labels that illustrate the animation structure with the ability to directly jump to certain animation steps, and full-fledged video player controls. A slider for adjusting the current animation step as an execution percentage state can also be used for a "fast forward" navigation.

For visualizers who are not yet satisfied with the extended features offered by ANIMAL and for developers interested in implementing extensions, we discuss example ANIMAL extensions in the next chapter.

Chapter 6

Extending ANIMAL Using ANIMAL-FARM

6.1 Introduction

We have already discussed how the ANIMAL-FARM framework presented in chapter 4 supports the dynamic addition and removal of components. The ANIMAL system built on the framework therefore also supports the integration of new components at run-time, as well as their removal. Note that there is a difference between extensions of the ANIMAL system and the framework: the former type of extensions provides added functionality for a concrete system, while the latter type modifies the framework. Here, we are concerned with extensions to the concrete implementation prototype to better illustrate how easily new extensions can be developed. The design and implementation process does not require modifying any existing code (**DR1**) or intimate system knowledge (**DR3**). The developed extensions can be added or removed using the *ComponentConfigurer* introduced in Figure 4.8 on page 82.

We have already discussed the individual components that make up the ANIMAL AV system. Only a small number of components are tightly connected and explicitly referred to. These classes mainly address the core animation modeling and GUI interactions. Other components are loaded dynamically and instantiated at run-time. This especially concerns the graphical primitives, animation effects and import / export layers. This dynamic embedding is performed both at start-up and when the user alters the system settings while it is running. Thus, ANIMAL is dynamically extensible.

However, the fact that components can be added or removed at run-time does not state how easily this can be achieved. This concerns both the developer and the user or visualizer who wants to embed components. In this chapter, we discuss some example extensions that cover the range of extensibility provided by ANIMAL. Where necessary, some implementation details will also be provided. A precise implementation guideline is available online [169].

The chapter is organized as follows. In section 6.2, we discuss two example extensions for graphical primitives. Extended animation effects are address in section 6.3. Adding sub-effects to already implemented effects requires the extension of object handlers. This process is illustrated in section 6.4. Section 6.5 shows how a new language support can be installed for the whole ANIMAL GUI. Additional animation import and export filter implementations are discussed in section 6.6. Section 6.7 discusses additions for interactivity added within the course of embedding ANIMAL to the JHAVÉ system. Section 6.8 summarizes the chapter.

6.2 Extending Graphical Primitives

There are two principal types of extensions for graphical primitives applicable to AV systems. The first type, and probably the more obvious one, provides a fully new primitive that is not connected to the other available primitives. Alternatively, an aggregate primitive that combines multiple other primitives with modified or extended semantics can be introduced.
In this section, we discuss one example each for both approaches. For the completely new primitive, we address embedding support for images. The aggregate extension presented here describes support for list elements as they are used throughout computer science.

6.2.1 Example Extension: *Image* Support

In order to develop a completely new primitive, a set of standardized steps has to be followed, which can be summarized as follows:

1. determine and define the properties of the primitive,
2. decide which properties if any should be modeled as attributes for performance reasons,
3. implement the basic functionality for displaying the primitive, including the abstract methods inherited from the framework,
4. implement a handler object for the new primitive,
5. optionally implement a graphical editor component for visual adjustment of the displayable properties,
6. optionally provide classes for im- and exporting of the new primitive, as the new primitive cannot be saved and loaded otherwise,
7. and finally embed the new primitive into ANIMAL using ANIMAL's *ComponentConfigurer* component.

To illustrate these steps, we regard the *Image* primitive extension. Images can be a great boon to many application areas in algorithm animation. For example, geometric algorithms may benefit from embedded images in appropriate format, rather than hand-drawn objects based on the standard primitives. The accompanying explanation of the visual content of the animation may also employ images, for example by presenting a scanned flow-chart of the algorithmic structure.
First, we have to define the properties of images. Images possess a *location* inherited from the framework. In keeping with the standard usage, we pick the upper left corner of the image as its location. The image source is described by its URL and an internal *javax.swing.ImageIcon* for rendering it on the graphical context. Additionally, the width and height of the image are important. We will treat width, height and URL as properties, and store the ImageIcon and location as attributes for efficiency reasons.
The basic functionality to be implemented is easy to accomplish. The inherited *getLocation* method simply returns the location of the image, while *getBoundingBox* returns a rectangle using the location as the upper left and the location plus (width, height) as the lower right corner. *translate* is

implemented by modifying the location of the image. Displaying within the *paint* method is performed by using the special *drawImage* method of the *java.awt.Graphics* class. *Scale* and *rotate* can be achieved within the *java.awt.Graphics2D* environment introduced by JDK 1.2.

The *ImageHandler* class responsible for handling the animation effects for image primitives is very easy to implement. Details on the process are given in section 6.4 which focuses on implementing handler extensions. Implementing the graphical editor is straightforward using the provided basic functionality residing in class *animal.editor.GraphicEditor*. This includes support for editing the primitive name and depth, as well as adding the three standard buttons *OK, Apply* and *Cancel* to the bottom of the displayed frame. Appropriate elements for entering the URL of the image to embed and its width and height are easy to define. The main task left is then to make sure that the communication flow between the GUI elements and the underlying primitive is maintained.

The implementation of import and export filters for image primitives is discussed in more detail in section 6.6. In the near future, the developer will be able to make the extension available using a special platform for World-Wide Web-based ANIMAL distribution, the *J-Updater* package implemented by Matthew Smith [193]. For this end, a 20×20 *GIF* image representing the new primitive must also be provided. As the *J-Updater* project is not fully finished yet, the user currently has to download extensions manually. Independently of how the extensions have been acquired, end users, typically adopting the user or visualizer role, can easily embed the functionality by starting the *ComponentConfigurer* and typing in the base name of the extension, that is, *Image*. The extension is then instantly usable.

6.2.2 Example Extension: *List Element* Support

List elements are commonly encountered in computer science applications and especially introductory programming courses. Many authors use a similar notation for list elements that shows two boxes, one containing the value of the list element and the other the pointers. List elements thus can be generated as an aggregation of available primitives. Instead of doing this manually for each instance, we examine how we can implement a list element support extension.

Figure 6.1 shows an example list element with two pointers. As shown, it can be assembled from two filled polygons, a possibly empty set of polylines with a forward arrow and a text entry. In ANIMAL terms, the resulting elements are mostly instances of *animal.graphics.PTPolyline* with appropriate values for the *closed, filled* and *fwArrow* properties. The color of the box outline and the fill color of the polygons should be adjustable, requiring access to the respective properties. Due to the visual appearance, we have decided to call the extension *animal.graphics.PTBoxPointer*.

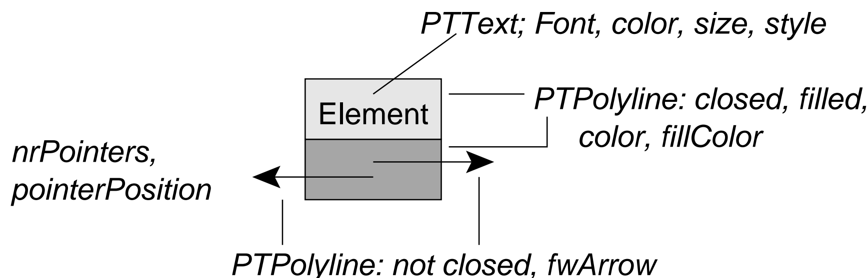

Figure 6.1: Schematic View of List Element Primitives and Specific Components

We have decided that the text element and all pointers shall share the same color to provide a "cleaner" look and feel. Thus, only the color property of the *animal.graphics.PTText* component is accessible, and not the color properties of the individual pointers. In addition to the color, the text entry itself together with the font name, style and size shall be specifiable. Finally, assembling the object requires properties that describe the pointer position and the number of pointers. We have decided to support the five pointer positions *none, top, left, right* and *bottom*, with Figure 6.1 showing the commonly assumed *bottom* position. The number of pointers in the example is 2 and dictates the size of the array storing the pointers in the code, as well as the dimension of the pointer box on the display.

Implementing the extension is comparatively easy based on these definitions. Basically, we declare the aggregated primitives and implement the basic operations from the framework. For example, the *paint* method implementation simply invokes the *paint* method of all components in the appropriate order. The bounding box is generated by building a union over the bounding boxes of all elements. The location of the object is the upper left corner of the topmost box. Note that this is not necessarily the same as the upper left corner of the list element's bounding box, depending on the position of the pointers.

One small issue lies in initializing the layout for the new primitive. We proceed as follows. First we place the text at a small offset from the location in such a way that the upper left corner of the text lies slightly below and to the right of the target upper left corner of the complete primitive. This is accomplished by determining the display properties of the text using the *java.awt.FontMetrics* object for the chosen font and determining the dimensions of the given text. The text baseline is then placed at the location plus a slight offset and a vertical distance equal to the font's maximum ascent.

After the text component is placed, the text box is wrapped around the text, based on the text's width and height. The lower box starts at the lower left of the text box and has a sufficient height to contain all pointers. Finally, the individual pointer start points are calculated and the pointers drawn to the target location. If the pointer box is not placed at the bottom position, the calculation of the pointer box is adapted appropriately. The whole object can be translated by translating the object's location and re-initializing the object.

One implementation issue has to be resolved when dealing with aggregate primitive types: the question of separate or shared properties. Normally, the $3+n$ primitives encapsulated by the new primitive type will each have their own properties object. However, this makes it much more difficult for the new primitive to gain and maintain control over the display of the subcomponents, for example by preventing the modification of the *closed* attribute for the two polygon boxes. On the other hand, if we simply share the same properties object in all primitives, all *animal.graphics.PTPolyline* primitives used in the list element will receive the same color when looking up the associated property.

We already prepared for such problems when we developed the graphical editor interfaces. A supplementary class *animal.misc.PropertyNameMapper* is used for mapping lookup requests according to the rules built into *animal.main.PropertiedObject* instances. Listing A.10 on page 190 shows the code of the name mapper.

The approach taken is as follows. When a polyline tries to determine its color, it will normally access it through the properties object using the special access key, in this example *"Polyline.color"*. The property name mapping mechanism is employed by rerouting direct property access through a *mapKey(String)* invocation, where the parameter represents the original request key. If the internal hash table of the mapper does not list an entry for the passed parameter, the parameter is returned

unchanged; otherwise, the associated entry is returned.

Name Mapping Rules: Text Box	
Key	*Value*
Polyline.color	BoxPointer.color
Polyline.fillColor	BoxPointer.fillColor

Name Mapping Rules: Pointer Area Box	
Key	*Value*
Polyline.color	BoxPointer.pointerAreaColor
Polyline.fillColor	BoxPointer.pointerAreaFillColor

Properties Values	
Key	*Value*
BoxPointer.color	red
BoxPointer.fillColor	green
BoxPointer.pointerAreaColor	blue
BoxPointer.pointerAreaFillColor	black

Table 6.1: *PropertyNameMapper* Example Usage

Table 6.1 shows a small example. The two tables on the top contain the mapping for the text box and the pointer area box, respectively. Thus, the *color* of a list element's text box is stored in the property *BoxPointer.color*, and the fill color of the pointer area box is stored as *BoxPointer.pointerAreaFillColor*. When one of the components requests a specific property, the actual property key to use is looked up in the table associated with the object. Thus, the list element will have a red text box filled with green, and a blue pointer area box filled with black. The property name mapper instances for the text element, text box and pointer area box, as well as the pointer instances are declared and assigned within the *BoxPointer* instance that also assigns its properties object as shared.
After these considerations, implementing the class code for list elements in the *BoxPointer* class is easy. The same goes for the visual editor, where the main concern is that requests are always performed on the shared properties instance. Import and export implementations are deferred to section 6.6 on extending import and export layers.
The only thing that remains to be done except for packaging and configuration using the *ComponentConfigurer* is implementing a *BoxPointerHandler*. As with the image handler, this is not difficult if carefully done. See section 6.4 for more details on implementing this and other handlers.
Additionally, the primitive offers extended animation subeffects for the *move* animator. Apart from moving the full object, it is also possible to move the main object body without modifying the pointer positions and moving individual pointer target positions. The former operation is needed when illustrating operations such as inserting elements into a list or removing them, while the latter operation is used for "linking" elements. Instead of performing a straightforward *translate*, the effect is restricted to the associated target primitive or primitives. Listing A.3 on page 183 contains the code for the handler that illustrates this situation.

6.3 Extending Animation Effects

Similarly to graphical primitives, animation effect extensions can be divided in two groups: those that provide a fully new animator, and others that provide only modified or extended subeffects.

The first type requires the implementation of a new animation effect, while the latter can be accomplished by providing handler extensions, as discussed in section 6.4. Here, we instead focus on the former aspect of implementing a completely new animation effect.
As an example, we regard the *Zoom* animation effect. Primitives can be scaled using the *Scale* animator. However, this operation only applies to the currently selected primitives. The *Zoom* effect operates on the display and is therefore independent of the visible primitives. It enables the visualizer to guide and focus the user's focus by setting the magnification and also defining the upper left corner of the currently visible animation content. A similar operation is available in *JSamba* [196].
Zoom effects require the target magnification and the upper left corner of the target display. These parameters can be queried in a simple GUI editor. The *Zoom* effect benefits from the fact that the animation window already employs a scroll pane for displaying the animation content. The current magnification factor is also modifiable by the user in a slide ruler using a pair of *get/set* methods.
Implementing an animation effect typically proceeds along the following steps:

1. determine the effect properties and whether they should be represented by attributes,
2. implement the animation effect initialization method *init*,
3. implement the *execute* method for performing the operations needed when the execution has finished,
4. implement code for generating the intermediate execution stage display in the *action* and *getProperty* methods,
5. implement a GUI editor for querying and adjusting the properties,
6. implement import and export facilities (see section 6.6 for more details),
7. embed the new animation effect using the *ComponentConfigurer* component.

For the *Zoom* animator, we decide to represent the upper left corner of the display by a *PTPoint* primitive that is generated in the same way as other primitives. The advantage of using such a point instead of absolute coordinates is that it is easier to specify the desired location, both visually and using scripting notation with relative placement. The timing specification is inherited from the super class *animal.animator.TimedAnimator*. Finally, the target magnification is represented by an integer value specifying the percentage of the current display magnification, with 100 representing the current magnification (100%).
The animator initialization invokes the inherited initialization method. As the magnification can be adjusted by the user, the current magnification is stored to serve as a basis for determining the target magnification. The *execute* method used for showing the final state of the animation effect execution determines the target magnification by multiplying the original magnification with the zoom factor. This value is then passed to the animation window component as the current magnification. After that, the inherited *execute* method is invoked to perform additional operations such as removing the animator from the list of scheduled animation effects.
The *action* method requests the desired zoom factor for the current point in time from the *getProperty* method. This factor is then assigned to the animation window component. The *getProperty* method performs a straightforward calculation for determining the target magnification. Assuming

that the zoom factor at the start of the animator is $zoom_{start}$ and that $zoom_{target}$ is the target zoom factor, the current zoom factor for any point in time during the animator execution is determined as

$$zoom_{now} = zoom_{start} + \frac{(zoom_{target} - zoom_{start}) * time_{now}}{time_{total}}$$

where $time_{now}$ and $time_{total}$ represent the current execution stage and the total duration of the animator, respectively.
The implementation of a GUI editor for querying the target upper left corner of the display is very straightforward. Note that the required entries for picking the target point and setting the timing are already inherited from the editor for timed animation effects. Similarly, it is easy to implement import and export facilities for *Zoom* effects. This topic will be discussed in section 6.6. After the extension has been successfully compiled and copied to an appropriate location in the CLASSPATH, the user or visualizer can activate the animation effect within the *ComponentConfigurer* component.

6.4 Extending Handler Capabilities

The primitive handlers are used for determining the specific animation effects supported by a given primitive. They also map a given effect to appropriate primitive invocations, as discussed in section 4.6. Extending the handlers offered by ANIMAL can be accomplished in two different ways: implementing a new handler, or adding an extension handler.
We still have to define the handlers for the *PTImage* and *PTBoxPointer* primitives introduced in sections 6.2.1 and 6.2.2, respectively. Remember that a primitive handler as specified by the framework contains the two methods *getMethods* and *propertyChange*. The former is responsible for constructing and returning a *java.util.Vector* instance of appropriate animation effect names for the primitive. The *propertyChange* method examines the *java.beans.PropertyChangeEvent* passed as a parameter and has to invoke appropriate methods on the primitive for generating the visual display that fits the desired effect.

6.4.1 Implementing a New Handler

The *ImageHandler* class that provides the handler for image primitives is very easy to implement. We focus on the main ideas of the implementation here. The complete code is included in appendix A.1 as Listings A.1 - A.2.
The *getMethods(Object)* method returns the names of the supported methods. We regard only two different parameter types in this extension: *java.awt.Point* representing a *Move* effects and *java.lang.Boolean* for *Show / TimedShow* effects. For move effects, the sole entry of the returned vector is *translate*, the standard move method. *TimedShow* animators offer both *show* and *hide*. The primitive passed in as a parameter can be ignored within this method. Other handlers may have to examine the primitive to determine the appropriate animation effects, as we will see with the *BoxPointerHandler*.
The handler's *propertyChange* analyzes the information encoded in the *java.beans.PropertyChangeEvent* parameter and determines appropriate actions. Note that the support for *show* and *hide* effects does not have to be encoded explicitly, as this is performed internally within the ANIMAL framework by adding or removing the primitives from the vector of objects scheduled for drawing.

The same process has to be performed for the *BoxPointerHandler* class. Listings A.3 - A.5 on page 185 show the processing performed for *Move* effects. Apart from simply moving the whole object with the *translate* method, we also have to manage the pointers (called "tips" within ANIMAL). The *setTip* method sets the position of the first pointer without moving the rest of the list element. The *translateWithFixedTip* method performs the opposite operation: it moves the whole object except for the target position of the first pointer. Both operations are sufficient for singly-linked lists. However, list elements with more than a single pointer need more flexible support. This will be examined in the next section. Additionally, the color properties shown in Figure 6.1 on page 149 can be adjusted by a *ColorChanger* animator.
The implementation of the *propertyChange* method is also relatively easy. The *show / hide* effects again are deferred to the super class, and the straightforward *translate* effect is also a matter of about five lines of code. The diverse operations for changing the color of the individual subcomponents can use the code of the respective primitives after a slight adaptation.

6.4.2 Example Extension: *Move* Subtype Support

In some cases, the developer may not wish to implement a full-fledged new animation effect, but rather add new subeffects to an existing animator. For example, *polylines* can be moved both as a whole object and on an individual node basis. This is even more important for *list elements*, where the visualizer must be able to move each pointer individually. Additionally, it should be possible to move the whole list element except for selected pointers. If these requirements are not met, the behavior of the list element is too restricted to provide full list element support.
The needed features for selecting individual nodes of a list element are already provided for polylines and can easily be reused for list elements. However, the code for offering the additional methods for moving or fixing selected pointers as well as for performing the actual operation has to be implemented.
For this end, developers can take either of two different solution approaches. First, they can modify the implementation of the underlying primitive handler by adding appropriate lines of code to the handler. This has adverse effects on reuse and may also affect system stability. Alternatively, they can implement a handler extension which acts for all purposes like the real handler, but is embedded dynamically based on the current configuration and can therefore also be removed easily if problems occur.
Based on the implementation of the standard list element handler starting in Listing A.3 on page 183f, Listings A.6 - A.9 on pages 186-189 contain the code for an extended list element handler. The *getMethods* method inherited from the super class contains additional methods for moving the whole list element except for individual pointers (*translateWithFixedTip #, translateWithFixedTips...*) as well as for moving selected pointers (*setTip #, setTips...*). The method names that end with ellipsis (...) cause a special *NodeSelector* window to pop up for selecting the precise nodes. The ellipsis in the method name is then replaced by the chosen node numbers. Otherwise, the number following the hash mark character # is the number of the selected pointer. Example method names could therefore be *setTips 1 3* for setting the first and third pointer, or *translateWithFixedTips 2 4* for moving the full list element without changing the position of the second and fourth pointer.
The property change handling is similar to the code used within the standard list element handler. First, the graphical primitive passed in is validated as an instance of the list element class *PTBoxPointer* and cast to the appropriate type. Then a cascaded conditional check for the chosen method

name is performed. If the method name contains a hash mark #, the integer following it represents the node number to work on. The move distance is calculated from the values encoded in the *PropertyChangeEvent*. The appropriate primitive method for setting the pointer or moving the object without the pointer can then be invoked.
Supporting the animation of multiple pointers at the same time is slightly more challenging. Here, a *java.util.StringTokenizer* is used for extracting the numbers passed by the method name. Note that the editing process replaces method names such as *setTips...* with a concrete name like *setTips 1 3* after prompting the visualizer for the nodes. A boolean array sized according to the number of pointers is used for representing the nodes to modify. The individual array elements are initialized to *false* and thus "no modification". The concrete elements to modify are determined by the StringTokenizer. After the array is initialized, the appropriate methods are invoked, based on the type of animation method. Note that the method does *not* end with *else super.propertyChange(ptgo, e)*, as this would lead to a Stack overflow due to tailless recursion.

6.5 Adding Language Support

Adding a new language support for the GUI components is easy. The user or visualizer simply downloads a language support pack from the ANIMAL home page [169] and uses the *ComponentConfigurer* component to embed it in ANIMAL. The translation texts are stored in ASCII files named *AnimalResources.ll_CT*, where *ll* stands for the language code and CT for the country code in upper case. Thus, the US-English texts are stored in *AnimalResources.en_US*, German resides in *AnimalResources.de_DE* and Spanish would be stored in *AnimalResources.es_SP*.

Listing 6.1: Example Definitions from the US-English Language Support File

```
About=About Animal
AnimAuthor=Animation Author:
AnimInfo=About this Animation
AnimSize=Animation Size:
AnimTitle=Animation Title:
about.iconName=Help.gif
about.label=About Animal
about.mnemonic=a
about.targetCall=showAboutDialog
about.toolTipText=Shows information about Animal
animBGColor=animation background color
animBGColorOption=Animation Background:
animInfo.iconName=Help.gif
animInfo.label=About this visualization
animInfo.mnemonic=v
animInfo.targetCall=showAnimInfoDialog
animInfo.toolTipText=Show info about the current animation
```

The format of the resource files is a simple sequence of one *key=value* pair on each line. Listing 6.1 shows an example selection of the US-English texts for ANIMAL. If the chosen language is not supported, it is still easy to provide support. The developer simply has to copy the English resource file and replace the values - only the values, *not* the keys! - with the appropriate translation. After

adding the language file via the *ComponentConfigurer*, the new language is instantly available within the *Language* menu of the main ANIMAL window.
The entries of the file can be classified in four categories according to their target area of use: simple labels, standard GUI elements such as radio buttons, menu items and button labels. The simple labels, such as *AnimAuthor* in Listing 6.1, provide only the straightforward translation. Standard GUI element entries usually also include a tool tip text that provides a short description. Menu items and buttons may also contain an icon and a method to invoke upon activation. The icon and method are specified by name, as can be seen with the entries *about.iconName* and *about.targetCall* in Listing 6.1, respectively.

6.6 Extending Import and Export Facilities

The possible extensions for import and export facilities can be grouped in two aspects: providing a completely new import or export format, or adding to an existing format. Completely new import or export formats have to be developed from scratch using ANIMAL's support classes and embedded using the *ComponentConfigurer*. Additions to an existing format only have to be placed in an appropriate directory where the Java runtime environment can locate them. They are used automatically if the primitive they handle is encountered.

6.6.1 Example Extension: Adding JSamba Import Facilities

JSamba [197] employs a scripting language for specifying animation content. Providing an import filter for JSamba and other scripting-based animation formats requires the following operations:

1. implement components that handle the parsing of the scripting code,
2. map the parsed primitives and animation effects to ANIMAL primitives and effects "as closely as possible",
3. insert the generated elements into the animation.

ANIMAL comes with an extensive support component for parsing ASCII-based scripting languages that place at most one command on a line of input. The developer can use this class to generate a parser with comparatively little effort, or use the *sourcerer* component that builds a parser from an extended Backus-Naur-Form specification. The output of this parser is a properties object that stores the parsed values, and can be used for building ANIMAL components.
Mapping the primitives and animation effects to ANIMAL is typically easy to achieve. In some cases, compromises may be necessary. For example, ANIMAL does not support rounded edges in polygons. While this is a possible area for future extensions, animations that employ rounded edges can currently only be rendered approximately. Once the current command has been mapped into an ANIMAL representation, the objects can be inserted with a single straightforward method invocation.
Implementing import facilities for new primitives or animators in an existing format is even easier. The developer can adapt the implementation of the new filter to those already present for the given format. The same support classes for parsing ASCII-based input can be used. For example, parsing support for *Image* primitives or *Zoom* effects is very straightforward to implement after the developer has defined the precise structure of the file format. The new classes are then placed in the

appropriate directory in the CLASSPATH. For example, an *ImageImporter* class for core ANIMAL would be placed in the directory *animal.exchange.animalascii* if the ASCII-based ANIMAL format is used.

6.6.2 Example Extension: Image and Video Export

As an interesting application of the flexible export facilities in ANIMAL, we provide both image and video exporting. To keep ANIMAL's code small, both facilities rely on external APIs for performing the file stream handling. Image generation is highly flexible: the user or visualizer can select an arbitrary set of steps for export and decide between a static snapshot of the step start or a "dynamic" export which saves all intermediate frames as individual images. The latter can for example be used for generating animated GIFs using other freely available tools. Additionally, the output images can be scaled to the canvas display magnification or any user-specified magnification. The actual exporting is accomplished by drawing the requested animation steps at the chosen magnification and passing the resulting image to the export library. We have chosen *JIMI* [202] for this purpose, as it is both easy to handle and flexible. Due to the straightforward implementation, the complete source code for image exporting is only 6615 Bytes long.
Video export currently uses Apple's *Quicktime for Java* API [10]. The export follows the same basic approach, except that the animation display images are now drawn on a *QTCanvas* object instead of the ANIMAL animation window. The total size of the Quicktime generation filter is roughly 13 kB. However, due to the use of native libraries within the Quicktime for Java library, video export is currently limited to MacOS™ and Microsoft Windows™ .

6.7 Interactivity Support

Within the course of a cooperation with Tom Naps at the University of Wisconsin Oshkosh, ANIMAL was adapted to fit the *Visualizer* interface of Naps' JHAVÉ system. Part of this adaptation covers the support for interactive prediction using multiple choice quizzes and dynamic HTML-based documentation, as described in [2].

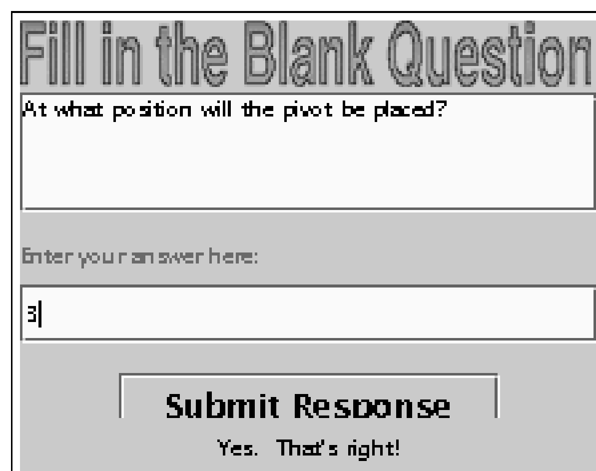

Figure 6.2: Interactive Prediction: Fill in the Blanks Question

The added components are gathered in an *InfoFrame* as shown in Figure 6.2. Here, the user is prompted to predict the next position of the pivot element in a *Quicksort* animation. Adaptive documentation [134] is also supported and can be shown in the *InfoFrame*. For layout reasons, this is not included in Figure 6.2. All interactive elements are available in ANIMALSCRIPT, including dynamically adapting documentation that can change the document URL to update the contents. More details on this can be found in [131], although this does not yet cover the incorporation of ANIMAL. A paper that describes the incorporation of interactivity into ANIMAL is currently in print [181].

6.8 Summary

In this chapter, we have outlined some extension possibilities for ANIMAL. The dynamic loading of independent components widely employed throughout ANIMAL allows easy plugging in and unplugging of extensions. Developing extensions is usually relatively easy, largely independent of the content of the extension. In general, implementation difficulties stem more from the implemented content than from the environment as fixed by the framework.
Typical extensions areas include new graphical primitives such as *images* and *list elements*, animation effects such as *zoom* and special animation effect methods, such as moving only selected nodes of a given primitive. Graphical primitive handlers acting as agents between the associated primitives and effects thereon can also be extended by an external class. Thus, no code modification is necessary.
ANIMAL's GUI can be translated on the fly by selecting the target language from the main AV window. Adding a new language is another configuration operation performed within the *ComponentConfigurer*. Providing a new language requires only the translation of an ASCII-based resource file and configuring the component.
Extending the set of import and export filters provided by ANIMAL is performed by implementing the filters using the prepared support classes for ASCII parsing and object storage. The new objects are then added to ANIMAL using the *ComponentConfigurer*. As example filters, we have briefly touched on importing *JSamba* [197] and exporting animations in image or Quicktime video format.
As we have seen, the large goals of making ANIMAL both extensible and configurable along the lines of the proposed framework were achieved. In the next chapter, we examine how ANIMAL measures up against the requirements placed in chapter 3.

Chapter 7

Evaluation

7.1 Introduction

In this chapter, we evaluate the features offered by the ANIMAL-FARM framework introduced in chapter 4 and its prototype ANIMAL. To do so, we revisit the requirements defined in chapter 3 and discuss the extent of their realization in our system.
For evaluation purposes, we offer two different views of the requirements. First, we examine the state of support for individual requirements independent of the user role placing the requirement. This mapping is then performed within a table that summarizes the findings and maps them against the different requirements.
The requirements can be divided into the following 13 different areas, each of which is evaluated in a separate section:

- required environment including special hardware or operating system demands,
- extensibility and configurability,
- development state and performance,
- applicability,
- content generation,
- content presentation,
- user interface considerations,
- display controls,
- user interaction,
- educational support,
- algorithm understanding support,
- file exchange,
- and programmer requirements.

7.2 Evaluation of ANIMAL's Support for the Requirements

In this section and its subsections, we evaluate the degree to which ANIMAL measures up against the requirements of chapter 3. The section is structured into individual subsections that address one segment of related requirements each. We also provide a table overview that maps the current implementation state of ANIMAL against the different user roles.

7.2.1 System Requirements Evaluation

System requirements define the target platform needed for running a piece of software - in our case, an AV system. The foremost system requirements as discussed in section 3.2.1 are operation system independence (**GR1**) and modest hard disk and RAM requirements (**GR2**). Additionally, the system should not require special additional hardware or software libraries (**GR3**), and be able to run without an Internet connection (**GR4**).
Today, the main criterion for platform independence is usually no longer the actual hardware but rather the operating system. On the one hand, several operating systems are available for different hardware platforms - for example, Linux -, and on the other hand, there is no longer a unique mapping from hardware to the operating system even for "exotic" hardware. Therefore, the main question to ask is whether the given system is independent of the operating system it runs under. Most AV systems which are freely available today are implemented in Java. Provided that a sufficiently recent Java runtime environment is available for the target system, any Java-based AV system is basically platform independent. However, the platform independence can only be guaranteed if no special hardware or additional, possibly platform-dependent, software components are required.
ANIMAL is written in standard Java and thus fulfills **GR1**. The base system also does not require any special hardware of external libraries, in accordance with **GR3**. Optional extensions, such as *image* and *Quicktime video* export, require special libraries, which may introduce platform dependence due to native code. This is actually the case with the Quicktime library, which is currently limited to Windows and MacOS. If this library were needed to run ANIMAL, **GR3** would be violated; however, its absence only prevents accessing an optional extension without restricting the basic functionality.
The standard ANIMAL distribution is sufficiently small to fit on a single 1.44 MB floppy, leaving enough space for an archive of all currently available ANIMAL animations. The RAM requirements are dictated by Java and are typically 32 MB or more on most modern computers. Thus, **GR2** that demands modest hard disk and RAM requirements is fulfilled.
In contrast to some AV systems such as *Jeliot* [72], ANIMAL works on a computer without Internet access, as demanded in **GR4**. If an Internet connection is present, the user may log in to the ANIMAL repository and download animations and extension packages. As ANIMAL is free, it can be acquired and used by both teachers and learners. Its lean size of less than one megabyte also makes it highly usable for both faculty and students (**VR6**).

7.2.2 Extensibility Requirements

Apart from the system requirements regarded in section 7.2.1, extensibility (**VR2**) places the most far-reaching restrictions on a system's architectural design. In chapter 3, we have isolated the following requirements that address extensibility: easy addition of components (**DR1**), the presence of

documentation (**DR2**) that illustrates how extensions can be implemented without detailed system knowledge (**DR3**), and support for configuring the system's components (**VR3**).
We address all of these requirements in the ANIMAL-FARM AV framework presented in chapter 4. An extensive documentation of how extensions can be generated is included in this thesis and also provided on the Internet [168]. The most challenging aspect regarding extensibility is making the provision of extensions not only *possible* but also *easy* - especially since we have set ourselves the goal of making source code modifications unnecessary.
As we have seen in chapter 4, this ambitious goal is achieved by the AV framework presented in this thesis. How did we manage to do it? Let us briefly summarize the main techniques, to further ensure that the goal has been met.
Within the framework design, we have consequently adapted the approach of representing object state in a dynamic data structure to extensible system design. Usually, this approach is found in libraries of adaptable GUI components, such as Java Swing. We have extended this approach as follows. The framework contains only a very small number of "fixed" central classes that provide the basic frame for the system. Most of these classes are abstract. Concrete instances are embedded in the system by dynamic loading, based on a configuration that can be passed in at start-up, or is fixed to disk in the form of a configuration file. To provide maximum flexibility, the configuration file is always looked for in the "current" directory, that is the directory where the application was started, before the *CLASSPATH* entries are searched. Therefore, there can in principle be as many different configurations as there are directories in the given file system.
The dynamically loaded classes are checked for conformance to the given target interface or class. If this check is passed and the attempt to instantiate the class using Java's reflection support also leads to no error, the new object is entered to a hash table. The entries of the hash table are *Prototypes* [61, p. 127ff] for future instances of the given class. The class name serves as the lookup key. Using hash tables for storing prototypes has the advantage that adding and removing components is very easy. To add a component, the system simply follows the same approach as described at the start of this paragraph: load the appropriate class definition, check its interface, and insert an instance of the class in the hash table, if all tests were passed. Removing a component is achieved by simply removing its key from the hash table.
The framework goes one step further by representing object state by properties instead of fixed attributes. The rationale behind this is threefold: first, properties allow the insertion and therefore "declaration" of new keys at run-time. Second, default values for all properties keys can be installed, with the default value being returning when the currently requested key has no associated value. Finally, mapping shared state-defining attributes is easier to accomplish using properties, as access restrictions need not be circumvented or rendered ineffective by making everything "public" or having default access.
From the extensibility point of view, the first two points are the most interesting. They effectively mean that if developers of a new extension require attributes not declared in existing classes, they can "declare" them by inserting an appropriate value in the chosen element. Note that this requires no source code modification - not even the presence of the actual source code is needed for successful compilation! Thus, our chosen approach offers new, innovative and interesting venues into adaptive software construction.
So far, we have illustrated that developing extensions without touching existing code is possible in our framework and the systems built on it. However, this does not mean that the development is necessarily easy. In order to also achieve this goal, the framework introduces another new concept: a *handler* class that acts as a negotiating agent between two other classes. Within the AV area, the

target classes are the graphical primitives on the one hand, and the animation effects on the other hand. The handler completely decouples these two classes, despite their seeming dependence on each other. It does so by acting as a *Singleton* [61, p. 1287ff] that handles the specific operations for one type of primitive.

An animation effect essentially needs to be able to perform two different types of operation on primitives. First, it has to query the status of the primitive to determine the concrete animation subeffects applicable to the primitive. Second, it has to invoke appropriate methods in the primitive instance that cause the primitive to adapt to the desired outcome of the animation effect.

The handler cuts this direct connection between the animation effect and the primitive. The animation effect contacts the primitive only via the handler. Thus, the handler generates and returns the list of applicable animation subeffects, and also determines and invokes the set of primitive methods needed for representing the effect. Both operations may require querying the state of the underlying primitive. For example, if a *Move* animation effect can also move single polyline nodes, the handler has to query the given polyline instance to determine how many nodes it has in order to generate the list of all applicable animation subeffects. However, this type of access is uncritical from the object-orientation point of view. The handler is a Singleton specially adapted for the primitive, and thus may have to change if the primitive changes. Preferably, developers should implement *extension handlers* for such operations. Following our basic approach of maximum flexibility, these are also added and maintained dynamically in a hash table, allowing for easy addition and removal.

Finally, we have to justify the claim that adding and removing components is easy, independent of how difficult or easy their implementation might have been. For this end, the framework provides the *ComponentConfigurer* component described in section 5.2. This component offers an easy way for adding and removing extensions while the system is running. It thus fulfills the requirements for easy component addition (**DR1**) and configurability (**VR3**). To sum up, we can therefore state that the requirements **VR3**, **DR1**, **DR2** and **DR3** are fulfilled very well by our framework.

7.2.3 Development State and Performance

Another issue of paramount importance for the target audience is the development state of the system. An "alpha" or "beta" version of a system is often unacceptable for many target usage areas due to likely changes in later versions and known but yet unresolved bugs. In general, the system should be well-maintained and stable (**GR5**) and, ideally, have a fixed update policy for planned versions (**GR6**). From the user's and visualizer's point of view, the total cost of ownership should also be small (**GR7**). Cheap but well-implemented software with sufficient performance (**GR8**) also has benefits for the developer, who is more likely to be interested in developing extensions on a good basis. Finally, all four roles are likely to appreciate a simple and straightforward download, installation and easy maintenance of the software (**GR9**).

ANIMAL's official release is tested for stability over several weeks before it is made publicly available. As with any other large piece of software, we cannot guarantee the absence of bugs. However, we have ascertained the stability and document known bugs on the tool's web page including updates when the bug has been resolved. Thus, ANIMAL can be taken to fulfill **GR5** and **GR6**.

Currently, ANIMAL is given away for free, and thus has an almost zero TCO value, as requested by **GR7**. Certain costs for the time needed for getting acquainted with ANIMAL can be assessed. However, these costs could not be avoided by any system, no matter how "intuitive" the interface might claim to be. ANIMAL is also not especially resource-hungry, running on all current computers with ease. The computer on which large parts of ANIMAL's newer components have been

developed is a 300 MHz Intel Pentium II notebook running SuSE Linux (release 6.1 to 7.3). ANIMAL runs smoothly on this computer; most components do not take a noticeable amount of time. The main exceptions to this rule are the elaborate parsing methods required for resolving relative placement or internationalization within ANIMALSCRIPT animation input. In all other areas, the system performance is more than sufficient for the performance requirement (**GR8**).
Finally, the newest release of ANIMAL has simplified the process of downloading and installing ANIMAL even further. After navigating to the system home page [168], the user has to fill out a small form and can then download a Java archive file that contains the full distribution. This *jar* can be invoked either manually or - under Microsoft Windows™ - by double clicking it with the mouse. The latter approach assumes that the file type has been appropriately associated with the Java runtime environment; otherwise, the *CLASSPATH* variable used by Java has to be updated. Maintaining the installed system is also easy, as the maintainer has to do nothing at all. The configuration files that steer the start-up component determination and initialization are searched first in the current directory, then in the *CLASSPATH* and finally in the *jar* file of the distribution. If a user should ruin his local installation, he or she can simply change to a different directory and restart ANIMAL "from scratch". Thus, **GR9** is also satisfied by ANIMAL.

7.2.4 Applicability

After the more technical aspects discussed in the previous sections, we will now focus briefly on the breadth of applicability offered by ANIMAL. As stated in **GR10**, the application area should not be restricted to certain topic areas, and the system goals should be well-documented (**GR11**). Again, ANIMAL measures up well against these requirements.
First, ANIMAL is not focused on a single topic area, such as graphs or sorting algorithms. Instead, both its primitives and animation effects are generic to allow arbitrary content to be displayed. The free placement of components and the support of the most relevant graphical primitives also allows ANIMAL to break out of the bounds of its original algorithm animation content and act as a small-scale alternative to full-fledged presentation tools such as Microsoft PowerPoint™ and StarOffice Impress™. For example, the animations generated by students within ANIMAL's graphical front-end include applications from Marketing, Economics, Physics and Mendel's Laws from Genealogy - topics that might as well have been presented in professional presentation tools. The application area is therefore flexible enough to meet the associated requirement of free applicability (**GR10**).
The goals of the system – providing easy-to-use, portable and flexible support for AV while incorporating modern software engineering techniques – are well-documented within this thesis. Thus, the requirement **GR11** that demands well-documented system goals is also met.

7.2.5 Animation Generation

Chapter 3 defines seven requirements regarding the generation of animation content. Firstly, the user shall be able to specify or generate the input data used by the algorithm (**UR9**). ANIMAL does not directly support this requirement, as it is completely context-free. It thus has no easy way of determining what input types and values might be required. In principle, the order of the algorithm parameters, names and types can be described in a *XML*-based notation. However, this approach reaches its limit if only subaspects of an intricate method invocation sequence are to be illustrated. For example, illustrating the Ford-Fulkerson algorithm for maximum flow in a network as described for example in [185] requires the initialization of a matrix and, depending on the realization, a set

of other method invocations. This is difficult to describe abstractly without causing undue difficulty to the person generating the specification. Instead of this approach, ANIMAL delegates the parsing and incorporation of input parameter data to the (possibly external) subcomponents that generate the animation content. For example, the JHAVÉ system [131] offers special input provision facilities for the algorithms it handles. The ANIMAL Sorting Algorithms Demo applet [172] shows an ANIMAL-specific animation generation applet that generates ANIMALSCRIPT animations of several popular sorting algorithms according to the element and color settings chosen by the user. The applet illustrates one approach how ANIMAL can show animations generated from user input by shifting the context knowledge needed for querying the parameter from the AV system to context-specific generation wrappers.

A second generation requirement is that several generation approaches (**VR5**) shall be offered to better adjust to the visualizer's skills and preferences. As described in previous chapters, ANIMAL currently offers three generation approaches: visually using the mouse with drag and drop, by scripting using ANIMALSCRIPT, and by API invocations. Additionally, by adding input filter applications as done for sorting algorithms [172], topic-specific animation families may be generated, and animations may also be extracted from filtered code interpretation. Thus, **VR5** is very well addressed by ANIMAL. The requirement of automatic ad-hoc generation (**VR11**) can also be met by the prototypical implementation. Note that automatic generation nearly always implies context knowledge of either the specific topic area or the underlying interpreted programming language.

The visual front-end for generating and editing animation content is easy to pick up, as shown in chapter 5.6. We have also tried to make the scripting and API front-end as easy to learn as possible by choosing a straightforward notation. One part of making a system easy to learn (**VR15**) lies in object placement. For visual generation, including a grid with user-adjustable width is enough for placing objects. Programmed animation generation approaches such as scripting and API invocations benefit greatly from relative object placement (**VR14**). In the absence of relative placement, the visualizer has to calculate the screen positions of all primitives manually. This is both time-consuming and likely to lead to incorrect placement. Relative placement, on the other hand, allows placing one object at a certain distance from a reference object, independent of where this reference object resides. ANIMALSCRIPT and the generator API address the requirement **VR14** by offering flexible relative object placement based on the reference object's bounding box, previously defined locations, the last position used or individual nodes of a polyline or polygon.

One aspect that a purely graphical front-end cannot easily achieve is quick generation of animation content, as demanded by **VR12**. ANIMAL therefore incorporates the scripting language ANIMALSCRIPT and the animation generation API for quick generation of animation content. Note that "quick" here refers to the time needed by an experienced visualizer for generating animation content. The time needed to familiarize oneself with the animation system has to be taken into account separately. ANIMAL keeps this initial learning time moderately short by providing easy-to-use components.

ANIMALSCRIPT also offers an optional component that allows the embedding of other ANIMALSCRIPT files to an existing animation. Thus, a library of reusable animation components may be built (**VR13**). A typical application of this component is to extract the introduction and summary sections of animations for a given sorting algorithm and dynamically linking them from the generated concrete sorting applications.

7.2.6 Content Presentation

We isolated four user requirements regarding the content presentation capabilities of an AV system in chapter 3: embedding of textual explanations (**UR1**), pseudo code (**UR2**), abstract views (**UR3**), and linking to (possibly external) documentation (**UR4**).
Embedding a textual explanation of the display is easy to do in systems based on visual generation. ANIMAL's editing front-end thus offers full support for arbitrary textual components. ANIMALSCRIPT and the generator API also support placing explanatory elements, textual or otherwise. The ability to use relative placement again is useful for adding elements next to the entries they are supposed to explain. **UR1** is therefore fulfilled by ANIMAL in all generation and display front-ends.
Pseudo code addition can be a powerful assistance in focusing on the contents of a given algorithm. For example, it can be used for abstracting or simplifying parts of the code. ANIMAL's graphical front-end for animation generation and editing views pseudocode as standard text and therefore supports embedding, indentation and highlighting.
ANIMALSCRIPT and the generator API offer special support for code and pseudo code. The code is specified as members of a *code group*, with each line containing one or more text elements with a visualizer-specified indentation level. The line or element to highlight can be chosen with a single command. Three different types of color highlighting are supported: *unhighlight*, *highlight* and *context highlight*. The latter can be used if the context of the current command is to be highlighted, typically the enclosing block or method invocation. ANIMAL therefore satisfies the requirement **UR2** for pseudocode embedding.
The simplest type of abstract algorithm views is pseudocode, which we have already determined to be supported by ANIMAL. Other, more elaborate abstract views include metaphors or "stories" [50]. While no special support for abstract views is incorporated in ANIMAL, the visualizer can assemble such elements using the graphical primitives ANIMAL provides within the GUI front-end, ANIMALSCRIPT and the generator API. We therefore argue that requirement **UR3** is at least partially supported by ANIMAL.
Sometimes, the layout prevents the inclusion of explanatory notes into the main animation body. In other cases, the visualizer may want to separate the description from the actual content. One good way of doing this is by placing the documentation in a separate window. The documentation may be in HTML format and thus allow the user to browse through it at will. Refined documentations may be adaptive to the current state of the display, and for example include concrete values lifted from the current animation into the documentation, as described in [2]. Based on the cooperation with Tom Naps, author of the JHAVÉ environment [131], ANIMAL also incorporates links to external documentation in the *InfoFrame* component, and thus fulfills **UR4**.

7.2.7 User Interface

We have already shown that ANIMAL is highly flexible and easy to customize in the previous sections and chapters, as demanded by requirement **UR21**. ANIMAL's graphical front-end employs the standard user interface considerations regarding clearness, orderliness, consistency, and forcing interaction only where necessary, as requested by **UR17**. Note that ANIMAL does not enforce sensible color choices, as this would restrict the creativity of the visualizer and possibly also the range of applicability. By making the last color chosen the default color for future operations, helpful color choices are made easier, in keeping with **UR20**.
ANIMAL also offers both smooth and static transitions. The visualizer may specify the base offset

and duration for each effect. Note that a duration of 0 equals an instantaneous transformation. To take different hardware performance and user perceptions into account, the user can also adjust the animation speed between the theoretical value of 0% and 1000% (10 times the original speed). Primitive transitions with a duration are performed smoothly, provided that the hardware is sufficiently fast to perform them on time. This is noticeably more often the case if virtual frame-based time is used than when using real time on a millisecond basis. Even reverse execution of effects is performed smoothly. Requirement **UR19** is therefore fulfilled.
Finally, both animations and the user front-end employed by ANIMAL are fully translatable on an ad-hoc basis. Currently, content translation is limited to animations generated in ANIMALSCRIPT or using the generator API. The target language for the animation is chosen on loading the animation to allow for placement resolution. The GUI front-end can be translated at any time by selecting the appropriate entry from the *Language* menu of ANIMAL's main window. Note that both translation types depend on the presence of appropriate language resources. Thus, many ANIMALSCRIPT animations may not be translatable due to a lack of language resource files. As this could easily be remedied by providing appropriate files, requirements **UR18** and **VR4** are fully satisfied.

7.2.8 Content Display and Controls

The controls for animation content display have to be highly flexible to allow the highest degree of comfort for users and visualizers. Users especially need to be able to pause and rewind the animation to be able to achieve meaningful understanding [2, 134]. The minimum requirements therefore are support for *play, pause, stop* and *rewind*. ANIMAL goes beyond this by adding a static *step forward / backward* facility.
Typically, ANIMAL animation steps are linked by an user interaction event rather than a fixed time delay. Thus, they offer a natural pause between segments of content, contrary to some other systems such as *JAWAA* [161, 152] or *JSamba* [196, 197]. The user can employ these breaks for gathering thoughts, rethinking what was shown, and assimilating the content, as stated in requirement **UR6**.
The *slide show* mode, on the other hand, links all originally unlinked animation steps into one large procession that can be started and paused with one mouse click each. The slide show mode is also employed in other systems such as *JAWAA* [161, 152] and *JSamba* [196, 197]. However, to our knowledge, ANIMAL is the only current general AV system that offers a play, step and slide show mode for *both* directions, as well as a jump to the start or end of the animation.
Additionally, the speed of the animation can be adjusted between 0% and 1000% of the original speed, as discussed in section 7.2.7. Finally, the display may be scaled between 0% and 500% of its original size. Thus, ANIMAL exceeds the requirements of video player controls (**UR5**) and adjustable speed (**UR30**). The speed controls also allow graceful degradation (**UR31**) if the current machine is slower or faster than anticipated by the animation's author.
At the user's discretion, individual animation steps may be skipped, due to the built-in structured view of the algorithm (**UR12**) and the slide rule for adjusting the current animation step. Additionally, the visualizer may decide to skip less interesting aspects of the underlying algorithm when developing the animation. As ANIMAL is not based on code interpretation, arbitrarily long "jumps" in the display are possible, as demanded by **VR8**.
ANIMAL does not provide direct support for displaying a "history" of previous states. However, given enough screen space, the previous state can be shown next to the current state, as employed in other tools [100, 101, 148]. Additionally, the flexible access controls allow going back to the previous step for looking up the last state, as well as replaying the current step with smooth motions

to easily detect changes. Requirement **UR32** is therefore at least partially satisfied.
ANIMAL currently does not support fixed visualizer conventions, such as the size of elements, similar to style sheets in Web context. At the present stage of development, only one set of properties is stored that reflects the last chosen settings for elements of the given type. However, we plan to provide an extension that offers style sheet-like capabilities in a later release. Requirement **VR7** is therefore mostly unsupported at the current stage of development. Multiple views (**VR9**) are also currently unsupported, but planned for a later extension.
Finally, we are working on embedding ANIMAL into several courses at various universities, including the University of Siegen, Germany, where ANIMAL was originally developed, the Darmstadt University of Technology, Germany, and the University of Wisconsin, Oshkosh. We hope to ultimately be able to duplicate the close relation to the lecture materials for a given course that was important for the success of the *BALSA* system. Currently, the associated requirement (**VR10**) is not yet fulfilled, but we have made significant progress in that direction. For this end, we count on our current positive experiences in faculty propagation, as demanded by **UR14**.

7.2.9 User Interaction

One segment of the user interaction (**UR7**) is using the controls that steer the animation. This activity, including the selection of labeled animation steps that act as hyperlinks, has already been discussed in section 7.2.8. In general, the simplest form of user interaction is requesting mouse clicks or other actions for advancing to the next animation step. As described in section 7.2.7, this is done in ANIMAL at the visualizer's discretion. A more advanced degree of interaction lies in the area of *interactive predictions*. Here, ANIMAL in combination with the JHAVÉ environment offers flexible interactive prediction including a "for real" mode that deactivates the animation controls until an answer has been submitted. Both **UR7** and **UR10** are therefore addressed by ANIMAL.
We have also outlined before how the user may specify input data and perform a role shift to visualizer, as described in requirements **UR8** and **UR9**. Note that the choice of appropriate data depends on both context and algorithm and can therefore not be performed automatically by a generic system. Of course, any type of *interesting* or *pathological* input data for any given algorithm can be used within ANIMAL; however, the data and the resulting animation effects have to be specified by the visualizer. Requirement **UR11** is therefore supported to the degree possible for a generic AV system.

7.2.10 Educational Support

Many AV systems are not able to embed any type of meaningful complexity analysis into their content. If a complexity analysis is embedded, it is usually restricted either to an abstract proof with no connection to the animation content, or limited to a fixed input set. Based on ANIMALSCRIPT's modular animation components, ANIMAL offers a middle-way solution. The basic introductory comments as well as the efficiency and complexity analysis of a given algorithm can be externalized to a separate file and shared by all animation instances of the given algorithm. ANIMALSCRIPT supports value assignments to String-based variables, which can be replaced by their value at the time of reading. From a practical point of view, the visualizer may set a given variable to the number of comparisons, assignments, swaps or any other value. The variable can then be used in the externalized animation module that discusses the efficiency of the algorithm in general, thus inserting the actual value of the given invocation. This can be used as a foundation for embedding

efficiency analysis even in a generic, context-unaware AV system (**UR13**).

ANIMAL currently does not support specific visual layout notations, for example "fact cards" as described in [125] (**UR15**), or the adjustment of the level of detail shown (**UR16**). Note that both requirements are very difficult to achieve for a generic system that is by definition unaware of the displayed primitives' semantics.

7.2.11 Algorithm Understanding Support

Several issues must be addressed to reach optimal support for algorithm understanding. First, the speed of the display must be adjustable to the user's preferences (**UR30**). The presentation of related issues should be consistent over separate animations (**UR33**); for example, the array underlying most sorting algorithms should always be displayed in the same way. As we have seen before, ANIMAL offers flexible controls for adjusting the display speed. The precise presentation of the animation content is not enforced by ANIMAL to avoid restricting visualizers unnecessarily. However, the primitives offered by ANIMAL and especially ANIMALSCRIPT are well-suited for consistent presentation. The requirement **UR30** is therefore fully met and **UR33** is at least partially fulfilled.

Two related issues that ANIMAL can handle to a certain extent are the clear mapping of code to animation components (**UR34**) and focusing of user attention (**UR25**). ANIMAL supports diverse approaches for explaining the mapping of the underlying code to the animation content: textual explanations (**UR1**), pseudo code including indentation support and highlighting (**UR2**), using external documentation (**UR4**) and interactive predictions (**UR10**). Similarly, color and especially highlighting techniques can be used for focusing the user's attention (**UR25**). However, the visualizer is not forced to employ any of these techniques. As ANIMAL is completely unaware of the context in which it is used, it also cannot provide automatic support for these features. The requirements **UR25** and **UR34** are therefore fulfilled only partially by ANIMAL. However, we can argue that a generic system cannot fulfill them to a greater degree than ANIMAL does. In a similar vein, ANIMAL offers a good selection of features for focusing on the understanding of algorithms and data structures (**VR1**), without enforcing adherence.

In the same vein, the requirements for a didactical structure of the animation (**UR26**) and the usage of small data sets for introducing new topics (**UR27**) are supported by ANIMAL, but cannot be enforced. The same goes for loop shortening (**UR28**), which is easy to do both in the visual generation front-end and using ANIMALSCRIPT or the API. Restricting the animation content to allowing only meaningful user events (**UR35**) and including at most one *interesting event* per animation step (**UR29**) can also not be enforced by a generic AV system such as ANIMAL.

At first glance, it might appear that ANIMAL is ill suited for supporting understanding of the algorithmic content. However, this is actually not the case. ANIMAL offers rich features for supporting the understanding of algorithms; however, it is up to the visualizer to use or ignore them. This is due to the fact that as a context-unaware system, no support can be provided for specific algorithms. In fact, as ANIMAL allows for greater design freedom in assembling the animation content, we can argue that ANIMAL allows better algorithm understanding than topic-specific tools if the visualizer works well.

7.2.12 File Exchange

File exchange between different applications is an import feature, but rather uncommon in AV systems. In general, the visualizer should be able to store the animation content at any time (**UR22**), and users should also be able to load and store animations at their discretion. ANIMAL offers flexible export and import facilities for this purpose, as required by **UR23** and **UR24**, respectively. Apart from the internal animation format in compressed and plain text format, ANIMAL also offers exporting to *XML*, various image formats and *Quicktime* videos. The latter two options require special external libraries. Note that Quicktime support is currently limited to Microsoft Windows™ and MacOS™.
Another related issue for file exchange is format compatibility. On the one hand, the current system release must be able to parse the content of previous releases (**GR12**). This is achieved in many systems, for example office products such as *Sun StarOffice*. Of even more interest is the facility to parse code produced by *later* releases of the software than is currently being used. ANIMAL addresses both issues by maintaining a version number for each primitive and effect. Depending on the version number, the appropriate set of attributes is parsed. In general, additions to the format are always added at the end of the line containing the element. Thus, requirements **GR12** and **GR13** are addressed by ANIMAL.

7.2.13 Programmer Requirements

Currently, ANIMAL does not address the programmer requirements for code editing in an IDE (**PR1**) and standard library support (**PR2**). ANIMAL can serve as a small-scale programming environment testbed (**PR3**) where the programmers implement graphical primitives or effects. In this function, it can also be used to introduce the dynamic loading and reconfiguration options.

7.3 Summary

In this chapter, we have evaluated the degree to which ANIMAL fulfills the requirements we defined in chapter 3. The discussion was split into different areas depending on the target focus of the individual requirements.
As we have seen, ANIMAL currently fulfills 62 of the 69 requirements. Seven requirements are only partially fulfilled, including support for abstract views (**UR3**), input data specification (**UR9**), automatic ad-hoc generation (**VR11**) and a close link to lecture materials (**VR10**). Five other requirements are supported but cannot be enforced by a generic system such as ANIMAL: didactical animation structure (**UR26**), small data sets for introductions (**UR27**), loop shortening (**UR28**), restricting user events (**UR35**) and placing at most one "interesting event" in each step (**UR29**).
ANIMAL currently does not address the following requirements: visual layout support (**UR15**), adjustment of detail level (**UR16**), visualizer animation conventions (**VR7**) and multiple views (**VR9**). The programmer requirements for an IDE (**PR1**), standard libraries support (**PR2**) and acting as a programming testbed (**PR3**) are also not fulfilled.
While ANIMAL is therefore not yet "perfect", it offers a large selection of highly relevant and innovative features for AV systems. The main focus area of extensibility and configurability is not found in most other comparable systems.

Chapter 8

Conclusions

Algorithm visualization and its subfield of algorithm animation have become more popular with teachers and instructors all over the world over the last few years. This is evidenced both by the number of publications on the topic and the number of algorithm animations available on the World Wide Web. In 1998, Price *et al.* estimated the number of available animation systems at more than 150 [156]. Since then, the number of systems and individual applets has risen further.
In chapter 2, we have provided a short history of algorithm animation and introduced the four user roles *user, visualizer, developer* and *programmer*. The *user* is the end-user of the AV system, focused on watching and possibly interacting with the animation specified by the *visualizer*. The implementor of the AV system is the *developer*, while the *programmer* wrote the animated algorithm - perhaps without knowing about it being targeted for animation. Individuals may also assume multiple roles. For example, the *visualizer* will typically also assume the *user* role to check the quality of the generated animation.
We have examined a large selection of algorithm visualization applets and systems in section 2.5. Based on our findings in this evaluation and literature research, we have formulated requirements that different user roles place on algorithm visualization (AV) systems. The individual requirements were discussed with their justification and related references in chapter 3. Most current AV systems can only address a very limited number of these requirements. AV systems, as any other piece of software, have to address rising expectations regarding functionality over time. It is unlikely that there will ever be a "perfect" AV system that satisfies all conceivable wishes of the target audience. Thus, we have to settle for a flexible AV system that allows the incorporation of changes and additions ("extensions"). Embedding new functionality must also be as easy as possible.
We have presented a framework for a dynamically extensible and adaptable AV system in chapter 4. The framework uses dynamic loading and reflection to achieve dynamic extensibility. For this end, we have consequently adapted the well-known dynamic data structure *hash table* for dynamic administration of system components. The chosen approach shows good performance and a very high degree of flexibility. The *handler* concept introduced in the same chapter enables a clean decoupling of graphical objects and animation effects. It further enhances data hiding and encapsulation, and can be applied to a large set of advanced topic areas.
The framework uses properties for representing object state, rather than the standard attributes. In this way, we open the door wide for dynamic extensibility and adaptivity. The insertion of new "attributes" as key and value pairs in a given object's properties is simple for any developer familiar with Java. The specialized support classes for propertied objects also reduce the disadvantage of losing static type-checking inherent in employing a dynamic data structure for state representation.

To ease the work of system developers and provide a boon for the users, the framework also supports internationalization of GUI elements. In fact, the support package added for this end makes generating a translatable GUI element easier than generating it in the standard way using the Swing API! Translating all GUI elements is achieved with a single method invocation, which may for example result from a menu selection.

On the AV side, the presented framework offers fully reversible animation playing. As late as 2001, an efficient rewind was considered "one of the most important 'open questions' in AV" by a member of the First International Program Visualization Workshop program committee [2]. We submit that our solution to this problem was finished less than a year after the mentioned workshop, illustrating the power and expressiveness of the concepts underlying our work.

We have also introduced the ANIMAL AV system, the first prototypical implementation of the framework. ANIMAL is dynamically extensible and configurable and allows animation generation and editing without context awareness. Based on the primitive types *point, polyline / polygon, text* and *arc* and the animation effects *(timed) show, move, rotate* and *change color*, nearly all types of two-dimensional displays can be achieved. In many cases, two or more primitives have to be combined to form a new entity.

ANIMAL uses *handlers* to mediate between the primitives and the effect, allowing for a very clean separation of concerns. Primitives focus only on their state representation and graphical rendition. Animation effects focus only on determining their current state depending on the current execution time - without any awareness of the underlying animated primitives or even the specific animation method chosen.

As one of the example extensions presented in chapter 6, we have built *list element* support based on a text element, two polygons, and a set of polylines with a tip at the end. Other example extensions presented include a *zoom* effect, new move effect subtypes, additional import and export facilities, and adding a new language for the graphical front-end.

As shown in our evaluation in chapter 7, ANIMAL fulfills all but seven of the 69 requirements introduced. Additionally, seven of the supported requirements are only partially supported, and five further requirements cannot be checked and enforced by a generic AV system unaware of the topic content. In total, ANIMAL addresses a larger number of requirements than any other current AV system that we are aware of.

The main research contributions of this thesis lie in diverse areas. We have presented a more extensive set of requirements based on our own research and reference literature than published before in this field. The requirements take the different interaction roles into account and thus provide four specific and one general perspective to requirements.

We have also defined an extensible AV framework that contains a large set of hotspots for future extensions. The proposed framework offers "real" reverse playing, as opposed to replaying a recorded history. The very strict separation of concerns of the components allows for easy customization and dynamic extension. Especially the incorporation of the negotiating *handler* components with dynamic extensibility allows for very flexible and adaptable usage. At the same time, developing extensions is very easy due to the tight focus that allows the developer to focus on a single aspect of the whole system in each component.

The components of the system are gathered at start-up or while the system is running, based on a configuration file and runtime configuration support. The configuration file is retrieved from the directory where the system is started, allowing for a different system to be used in each directory of a file system. If the directory does not contain a configuration file, the *CLASSPATH* are searched, followed by the distribution *jar* file.

We have also examined techniques for building GUI front-end components that can be translated by a single mouse click. Based on this research, we have introduced internationalization issues to algorithm animation, using relative object placement to account for different dimensions of content in varying languages.
The embedded extensible scripting language ANIMALSCRIPT allows for modular animation design and also supports automatic generation of animation content based on input data specified by the user or visualizer. In conjunction with the JHAVÉ AV system by Naps *et al.* [131], ANIMALSCRIPT also supports interesting features such as interactive predictions, as presented in a paper awaiting publication in mid-2002 [181].
The dynamic display capabilities of the ANIMAL-FARM framework include modifying the display speed and magnification, entering the target animation step, dragging a slider for fast-forwarding between 0% and 100% of the animation content, and a full-fledged video player control bar. The main feature of the video player capabilities is the unlimited capability for reverse playing, whether by jumping to the associated step or playing the step in reverse with full dynamics.
Individual animation steps can be labeled. The labels are gathered in a list and outline the structure of the animation. They also act as hyperlinks to the associated steps. Again, it does not matter if a selected label links to previous states of the current animation, as ANIMAL can easily rewind the animation to the selected state.
The ANIMAL-FARM framework and the implementation prototype ANIMAL are more extensive than most comparable systems. However, there are still several areas of future research to follow. In general, the research topics can be grouped in four areas: software engineering, AV, evaluation, and educational aspects.
On the software engineering side, the main weakness of the framework at the moment is its lack of a formal evaluation. While our experience in using the system shows that it is highly expressive and adaptable, it would be preferable to actually determine the degree to which the system can be adapted to different needs. Similarly, a careful assessment of the framework might locate limitations that are not obvious at first glance, as is the case with most of today's software systems.
Another interesting application of the framework comes in the form of the *J-Updater* package [193]. This package shall support the downloading of new additions to the system both outside and during run-time. Due to the extensive support for dynamic component addition and removal already embedded in the framework, the dynamic update should not prove to be too difficult. However, the work on this is not yet finished.
One application that partly belongs to software engineering and partly to AV is interactive AV generation and controlling by the user. Here, the user shall be able to enter parameters for the execution of a given algorithm and then see an animation based on the provided values. The challenge here is the specification and implementation of a front-end for querying parameter values for an *arbitrary* algorithm. Implementing a component that prompts for values for a specific algorithm or class of algorithms, such as sorting algorithms, is relatively easy. The task becomes much more challenging if there shall be only one component that is capable of handling *all* types of algorithms.
We speculate that a verbose language definition for specifying the parameters and possible conditions placed on them is necessary. For example, how can we address the effective input of a matrix, as needed for most graph algorithms? What about special properties and interdependencies? A simple property would be that negative values are forbidden for invocations of the *faculty* function. Implementing the same condition for a matrix used for *Dijkstra's Shortest Paths* algorithm is more challenging. In some cases, special combinations of parameter values may be inappropriate, for example for sorting algorithms that require an array and the length of the array passed as parameters.

In this case, the length of the array could of course be determined from the input values - but how can this condition be specified? Of course, there are even more complex examples, and it is as yet completely unclear how far one could pursue this topic area.

There are also several unresolved AV research areas. Of course, foremost of them are the requirements not yet supported by either the framework or the system, such as the incorporation of multiple animation windows, style sheets, and the adjustment of detail level. Additionally, more advanced types of interaction could be supported. This starts with interactive predictions that actually provide a didactically motivating answer, such as outlining reasons why a given answer might have been incorrect, or links to further resources. More advanced interaction types could query the user for interacting with the current animation state, for example by selecting one displayed primitive. Depending on which object is selected by the user, the animation could branch in different directions, or simply remain in the current step until the "correct" element is selected.

Another interesting application is the addition of a code interpretation system as an import filter. For example, Jeliot 2000 [113] could be used as a front end for analyzing Java code. Instead of providing its own graphical rendition of the underlying algorithm, an ANIMAL animation could be provided. As we have not yet managed to enter a cooperation with interested parties possessing code interpretation-based AV systems, we could not pursue this area. However, the concept seems plausible enough that it should be possible to perform this type of cooperation. In fact, the main problem will likely be adapting the code interpretation-based AV system to actually generated ANIMAL animation code as output, instead of a graphical rendition of the current algorithm state.

Two further related areas of future work concern an applet front-end for ANIMAL. Due to the large number of animations already available on [168], this applet should also incorporate the ability to load a compressed archive of all registered animations once. The user could then select the current animation to view from the list of archive entries and interact with the animation. Once a given animation is over, the user shall also be able to select another animation without having to restart the applet.

There are several areas in the framework that would profit from an extensive evaluation. This concerns the usage of ANIMAL as an AV system compared to other AV systems. Other aspects are as the learning effect to be gained from interactive predictions with didactical feedback as motived above. A working group led by the author and Tom Naps at the ITiCSE 2002 conference will evaluate techniques for improving the educational impact of AV. We speculate that other interesting areas of evaluation will come from this working group.

Finally, there is still no extensive AV repository geared for educational purposes. The "standard" collection of algorithm animations [33] lists the visualizations only by site and algorithm, and also does not incorporate newer entries. The repository prototype we have set up at [168] is still far from complete, but already offers more information about a single animation than other repositories. This includes a description of the animation content including screen shots and a classification of the interaction type, as well as bibliographic data. A team of eight AV experts including the author of this thesis are currently working on building an even more extensive AV repository to be available in summer 2002. The URL will probably be `http://www.algoanim.net`; the concrete URL will also be given on [168].

Appendix A

References

Algorithm Animation System Requirements Page

GR1: Operating system independence . . . 44
GR2: Modest hard disk and RAM requirements . . . 45
GR3: No special hardware or additional component requirements . . . 45
GR4: Independence of an Internet connection . . . 45
GR5: System maintenance and stability . . . 46
GR6: Update policy . . . 46
GR7: Total Cost of Ownership . . . 46
GR8: Sufficient system performance . . . 46
GR9: Easy downloading, installation and maintenance . . . 47
GR10: Flexible application area . . . 47
GR11: Well-documented system goals . . . 47
GR12: Upward file format compatibility . . . 47
GR13: Downward file format compatibility . . . 47
UR1: Embedded textual explanation of view . . . 48
UR2: Embedded pseudocode with highlighting . . . 49
UR3: Metaphors and Applications . . . 49
UR4: Linking to documentation . . . 49
UR5: Full-fledged video player controls . . . 50
UR6: Embedded breaks between steps . . . 50
UR7: User interactivity support . . . 50
UR8: Support role shift from user to visualizer . . . 51
UR9: Input data generation or specification . . . 51
UR10: Incorporate predictions and “quizzes” . . . 51
UR11: “Interesting” and “pathological” data incorporation . . . 52
UR12: Labeling of main steps . . . 52
UR13: Embedded analysis, complexity and comparison . . . 52
UR14: Faculty propagation . . . 53
UR15: Use of a visual layout notation . . . 53
UR16: Adjustment of detail level . . . 53
UR17: General user interface considerations . . . 54
UR18: Internationalization in animations and user front-end . . . 54
UR19: Smooth transitions . . . 54

UR20: Color incorporation with "useful" settings 54
UR21: Easy customization 55
UR22: Animation saving 55
UR23: Export facilities 55
UR24: Import facilities 56
UR25: Focusing of attention 56
UR26: Didactical structure of animation 56
UR27: Small data sets for introduction 56
UR28: Loop shortening 56
UR29: One "interesting event" per step 57
UR30: Adjustable speed control 57
UR31: "Graceful degradation" 57
UR32: Incorporation of a "history" 57
UR33: Consistent presentation over diverse animations 57
UR34: Clear mapping of code to animation 58
UR35: Restriction to meaningful user events 58
VR1: Focus on understanding the data structures and algorithmic steps 58
VR2: Extensibility 59
VR3: Configurability 59
VR4: Internationalization in generation front-end 59
VR5: Incorporation of several generation approaches 59
VR6: Educational use 60
VR7: Establishing own conventions 60
VR8: Step skipping 60
VR9: Optional multiple views 61
VR10: Close link between system and lecture materials 61
VR11: Automatic ad-hoc generation 61
VR12: Quick generation 61
VR13: Reusable animation components or modules 62
VR14: Relative object placement 62
VR15: Ease of learning 62
DR1: Easy addition of components 63
DR2: Extensibility documentation 63
DR3: Extension without detailed system knowledge 63
PR1: Code editing in an IDE 64
PR2: Standard libraries support 64
PR3: Programming environment testbed 64

List of Tables

2.1 Typical Properties of Generation Approaches 40

3.1 Requirements Overview 66

4.1 Internationalization Support for GUI Elements in *translator.TranslatableGUIElement* 81

6.1 *PropertyNameMapper* Example Usage 151

List of Figures

2.1 Base Form of Algorithm Animation Using an "Appropriate" Transformation 5
2.2 Schematic View of User Roles 7
2.3 Role Distribution in Topic-Specific Systems 9
2.4 Role Distribution in GUI-Based Systems 10
2.5 Role Distribution in API-based Systems 12
2.6 Role Distribution in Scripting-based Systems 14
2.7 Role Distribution in Declarative Visualization 15
2.8 Role Distribution in Code Interpretation Systems 16
2.9 Insertion Sort Applet by Hiromasa Sekisita [190] 18
2.10 Screen Shot of Gosling and Smith's Applet for Sorting Algorithms [70] 19
2.11 *Shellsort* Animation taken from [186] 20
2.12 *Towers of Hanoi* Animation taken from [60] 21
2.13 *Traveling Salesman Problem* Applet by Filipova [58] 22
2.14 Bubble Sort Screen Shot taken from [73] 22
2.15 Quicksort Animation Screen Shot from Eck's *xSortLab* applet [52] 23
2.16 Radix Sort Animation Screen Shot from [9] 24
2.17 Bubble Sort Screen Shot taken from [32] 25
2.18 Screen Shot taken from Papagelis' *Kruskal* animation [146] 26
2.19 Quicksort Screen Shot from Lang's applet [109] 26
2.20 Partitioning in Quicksort as Illustrated in Lang's applet [109] 27
2.21 Delaunay Triangulation Applet taken from [112] 28
2.22 Screen Shot of Neto's Merge Sort Applet [135] 28
2.23 Pâté Screen Shot taken from [162] 31

4.1 Modeling Attributes Using *Properties* with Customized Conversion Wrappers 70
4.2 Class Diagram of the Extended Property Support Class *XProperties* 71
4.3 Default Properties Interaction 73
4.4 UML Class Diagram for *Propertied* and *Editable* Objects 74
4.5 Import Layer Structure in the Framework 77
4.6 Export Layer Structure in the Framework 78
4.7 Class Structure of the *Translator* Class 79
4.8 *ComponentConfigurer* Window for Adding Extensions on the Fly 82
4.9 Abstract Schematic Handler Structure 84
4.10 Abstract Overview of the ANIMAL-FARM Framework Components 91
4.11 Architecture of Graphical Objects in the Framework 93
4.12 Architecture of the Framework Animation Effect Classes 97

4.13 Handlers and Handler Extensions in the ANIMAL-FARM Framework 100
4.14 Cooperation between *Animator, Primitive* and *GraphicObjectHandler* Instances . . 101
4.15 Incorporating Dynamically Added Handler Extensions 102
4.16 Example Animation Structure 104
4.17 Structure of Link Objects for Modeling the Transitions between Animation Steps . 104
4.18 Animation Representation in the ANIMAL-FARM Framework 106
4.19 Example Control Tool Bar 108
4.20 Timed Animator Execution 110
4.21 Treating Reverse Execution in the ANIMAL-FARM Framework 111
4.22 Import Layer Structure 112
4.23 Export Layer Structure 113

5.1 *PTGraphicObject* as an Implementation of *GraphicObject* 119
5.2 Inheritance Structure of the Graphical Primitives in ANIMAL 121
5.3 Class Diagram for *Point* Primitives in ANIMAL 122
5.4 Interpretation of Homogenous Coordinates 122
5.5 Class Diagram of *Polyline / Polygon* Primitives in ANIMAL 123
5.6 UML Inheritance Diagram for the *PTText* Class 125
5.7 UML Inheritance Diagram for the *PTArc* Class 126
5.8 Architecture of the Animator Base Class *Animator* 127
5.9 Inheritance Diagram for ANIMAL Animation Effects 129
5.10 Inheritance Diagram for *Show* Animators 129
5.11 Example *Show* Effect 130
5.12 Inheritance Diagram for *TimedShow* Animators 130
5.13 Inheritance Diagram for *ColorChanger* Animators 131
5.14 Color Interpolation in the *ColorChanger* Animator 131
5.15 Example Color Interpolation in the *ColorChanger* Animator 131
5.16 Inheritance Diagram for *Move* Animators 132
5.17 Timing of *Move* Animators 132
5.18 Example *Move* Animator 133
5.19 Inheritance Diagram for *Rotate* Animators 133
5.20 Timing Resolution for *Rotate* Animators 134
5.21 Example of *Rotate* Animators 134
5.22 ANIMAL's Main AV Frame 135
5.23 ANIMAL's Animation Window 136
5.24 ANIMAL's Control Tool Bar 137
5.25 ANIMAL's Implementation of the Animation Structure View 137
5.26 ANIMAL's *Drawing Window* GUI 138
5.27 *Arc* Primitive Editor Window 139
5.28 ANIMAL's *Animation Overview* GUI 140
5.29 ANIMAL's *Move Options* GUI Editor 140
5.30 ANIMAL's Export Mode Chooser Window 141

6.1 Schematic View of List Element Primitives and Specific Components 149
6.2 Interactive Prediction: Fill in the Blanks Question 157

A.1 Extension Listings

Listing A.1: ImageHandler Implementation, Part 1

```
package animal.graphics;

import java.awt.Point; // for "Move" animator

import java.beans.PropertyChangeEvent;

import java.util.Vector; // for "getMethods(Object o)"

import animal.misc.MSMath; // used for point difference

/**
 * Provides the operations that can be performed on images.
 *
 * @author Guido Roessling <roessling@acm.org>
 * @version 1.0 2001-11-15
 */
public class ImageHandler extends GraphicObjectHandler
{
  /**
   * Generates a Vector of effect types for the primitive.
   * The underlying animation effect is characterized by
   * the second parameter, which may also encode relevant
   * information.
   * We may also have to examine the properties of the
   * primitive, which is therefore passed in as a parameter.
   *
   * @param ptgo the graphical primitive that is used in
   * the animation effect. The parameter is currently not
   * used for images, but may be needed in the future.
   * @param obj the object that characterizes the type
   * of animation effect for which the vector of effects
   * has to be generated.
   */
  public Vector getMethods(PTGraphicObject ptgo, Object obj)
  {
    Vector result = new Vector(); // generate result container

    if (obj instanceof Point) // animation types for Move
      result.addElement("translate"); // move whole image

    if (obj instanceof Boolean)
    { // animation types for Show and TimedShow
      result.addElement("show");    // show image
      result.addElement("hide");    // hide image
    }
```

Listing A.2: ImageHandler Implementation, Part 2

```
    // add extension methods provided in other classes
    addExtensionMethodsFor(ptgo, obj, result);

    // return the vector of method names
    return result;
  }


  /**
   * Change a primitive property to perform the desired effect.
   * To do so, examine the PropertyChangeEvent passed in for
   * the animation effect name and the old and new value. Then
   * perform a set of appropriate method invocations on the
   * primitive passed as the first parameter.
   *
   * @param ptgo the graphical primitive to be animated
   * @param e the PropertyChangeEvent that encodes the
   * animation subeffect name, old and new value.
   */
  public void propertyChange(PTGraphicObject ptgo,
                             PropertyChangeEvent e)
  {
    // only works if the passed object is a PTImage!
    if (ptgo != null && ptgo instanceof PTImage) {
      PTImage image = (PTImage)ptgo;

      String what = e.getPropertyName(); // retrieve name

      if (what.equalsIgnoreCase("translate")) { // move
        // moving is always relative! Determine difference
        // between the last and current position and use this.
        Point old = (Point) e.getOldValue();
        Point now = (Point) e.getNewValue();
        Point diff = MSMath.diff(now, old);

        // move by (new.x - old.x, new.y - old.y)
        image.translate(diff.x, diff.y);
      }
      else // defer to super class; also handles show / hide!
        super.propertyChange(ptgo, e);
    }
  }
}
```

Listing A.3: BoxPointerHandler Implementation, Part 1

```
package animal.graphics;

import java.awt.Color; // for "ColorChanger" animator
import java.awt.Point; // for "Move" animator

import java.beans.PropertyChangeEvent;

import java.util.Vector; // for "getMethods(Object o)"

import animal.misc.MSMath;

/**
 * Provides operations that can be performed on list elements.
 *
 * @author Guido Roessling <roessling@acm.org>
 * @version 1.0 2001-11-15
 */
public class BoxPointerHandler extends GraphicObjectHandler
{
  /**
   * Generates a Vector of effect types for the primitive.
   * The underlying animation effect is characterized by
   * the second parameter, which may also encode relevant
   * information.
   * We may also have to examine the properties of the
   * primitive, which is therefore passed in as a parameter.
   *
   * @param ptgo the graphical primitive that used in
   * the animation effect. This must be examined to offer
   * appropriate methods for translating individual pointers
   * ("tips").
   * @param obj the object that characterizes the type
   * of animation effect for which the vector of effects
   * has to be generated.
   */
  public Vector getMethods(PTGraphicObject ptgo, Object obj)
  {
    Vector result = new Vector(); // generate output

    // primitive must be list element for this class!
    if (ptgo == null || !(ptgo instanceof PTBoxPointer))
      return new Vector(); // return empty method set

    // declare and cast local list element
    PTBoxPointer boxPointer = (PTBoxPointer)ptgo;
```

Listing A.4: BoxPointerHandler Implementation, Part 2

```
    if (obj instanceof Point) { // Move animator subtypes
      result.addElement("translate"); // move whole element

      // add methods depending on number of pointers
      if (boxPointer.getPointers() != null) {
        // determine number of pointers
        int nrPointers = boxPointer.getPointers().length;
        if (nrPointers > 0) {
          result.addElement("setTip"); // set the tip
          // also, move the whole element except for tip
          result.addElement("translateWithFixedTip");
        }
      }
    }
    if (obj instanceof Color) { // ColorChanger animator
      // enumerate all settable color properties
      result.addElement("text box frame & pointer color");
      result.addElement("fillColor");
      result.addElement("pointer box frame color");
      result.addElement("pointer background color");
      result.addElement("color");
      result.addElement("textcolor");
    }
    if (obj instanceof Boolean) { // Show or TimedShow
      result.addElement("show"); // show whole element
      result.addElement("hide"); // hide whole element
    }

    // add extension methods from other classes
    addExtensionMethodsFor(ptgo, obj, result);

    return result; // return all determined methods
  }


  /**
   * Transform the requested property change in method calls
   *
   * @param ptgo the graphical primitive to modify
   * @param e the PropertyChangeEvent that encodes
   * the information which property has to change how
   */
  public void propertyChange(PTGraphicObject ptgo,
                             PropertyChangeEvent e)
  {
    // only works if the passed object is a PTBoxPointer!
    if (ptgo != null ptgo instanceof PTBoxPointer) {
      PTBoxPointer boxPointer = (PTBoxPointer)ptgo; // cast
      String what = e.getPropertyName(); // retrieve property
```

Listing A.5: BoxPointerHandler Implementation, Part 3

```
      if (newValue != null && newValue instanceof Point) {
        // must be Move animator!
        Point old = (Point) e.getOldValue(); // old position
        Point now = (Point) e.getNewValue(); // new pos
        Point diff = MSMath.diff(now, old);  // difference

        if (what.equalsIgnoreCase("setTip")) { // (first) tip
          if (boxPointer.getPointers() != null)
            boxPointer.setTip(0,
                              MSMath.sum(boxPointer.getTip(0),
                                         diff)); // first tip
        }
        else if (what.equalsIgnoreCase("translate"))
          boxPointer.translate(diff.x, diff.y); // whole elem
        else if (what.equalsIgnoreCase("translateWithFixedTip"))
        boxPointer.translateWithFixedTips(diff.x, diff.y);
      }
      else if (newVal != null && newVal instanceof Color) {
        Color color = (Color)e.getNewValue();
        if (what.equalsIgnoreCase("text box frame & pointer color"))
          boxPointer.setColor(color);
        else if (what.equalsIgnoreCase("fillColor"))
          boxPointer.setBoxFillColor(color);
        else if (what.equalsIgnoreCase("pointer box frame color"))
          boxPointer.setPointerAreaColor(color);
        else if (what.equalsIgnoreCase("pointer background color")
                 || what.equalsIgnoreCase("color"))
          boxPointer.setPointerAreaFillColor(newVal);
        else if (what.equalsIgnoreCase("textcolor"))
          boxPointer.setTextColor(color);
      }
      else // handle by superclass, incl. Show / TimedShow!
        super.propertyChange(ptgo, e);
    }
  }
}
```

Listing A.6: Extension Handler for Moving Multiple List Element Pointers, Part 1

```
package animal.graphics;

import java.awt.Point; // for "Move" animator

import java.beans.PropertyChangeEvent;

import java.util.StringTokenizer; // to separate multiple nodes
import java.util.Vector; // for "getMethods(Object o)"

import animal.misc.MSMath; // used for point difference

/**
 * Perform animators on list elements with multiple pointers
 *
 * @author Guido Roessling <roessling@acm.org>
 * @version 1.0 2001-11-27
 */
public class BoxPointerMoveMultiplePointers
  extends GraphicObjectHandlerExtension // extension!
{
  /**
   * Create a new extension and assign the appropriate type
   */
  public BoxPointerMoveMultiplePointers()
  {
    type = PTBoxPointer.TYPE_LABEL; // handles list elements
  }

  /**
   * Generates a Vector of effect types for the primitive.
   * The underlying animation effect is characterized by
   * the second parameter, which may also encode relevant
   * information.
   * We may also have to examine the properties of the
   * primitive, which is therefore passed in as a parameter.
   *
   * @param ptgo the graphical primitive that used in
   * the animation effect. This must be examined to offer
   * appropriate methods for translating individual pointers
   * ("tips").
   * @param obj the object that characterizes the type
   * of animation effect for which the vector of effects
   * has to be generated.
   */
  public Vector getMethods(PTGraphicObject ptgo, Object obj)
  {
    // if primitive null or no list element, return "nothing"
    if (ptgo == null || !(ptgo instanceof PTBoxPointer))
      return result;
```

Listing A.7: Extension Handler for Moving Multiple List Element Pointers, Part 2

```
    Vector result = new Vector(); // result vector

    // cast primitive to appropriate type
    PTBoxPointer boxPointer = (PTBoxPointer)ptgo;

    if (obj instanceof Point) { // Move animator subtypes
      if (boxPointer.getPointers() != null) { // any pointers?
        // retrieve number of pointers, must be more than 1
        int nrPointers = boxPointer.getPointers().length;
        if (nrPointers > 1) {
          // move whole element except for tip t (for all t)
          for (int t = 0; t < nrPointers; t++)
            result.addElement("translateWithFixedTip #" + (t+1));

          // set only tip t, leave rest as it is
          for (int a = 0; a < nrPointers; a++)
            result.addElement("setTip #"+(a + 1));

          // let visualizer select which tips to set
          result.addElement("setTips...");

          // let visualizer select which tips to leave untouched
          result.addElement("translateWithFixedTips...");
        }
      }
    }
    return result; // return result vector
  }


  /**
   * Transform the requested property change in method calls
   *
   * @param ptgo the graphical primitive to modify
   * @param e the PropertyChangeEvent that encodes
   * the information which property has to change how
   */
  public void propertyChange(PTGraphicObject ptgo,
                             PropertyChangeEvent e)
  {
    // only works if the passed object is a PTBoxPointer!
    if (ptgo != null && ptgo instanceof PTBoxPointer) {
      PTBoxPointer boxPointer = (PTBoxPointer)ptgo;
```

Listing A.8: Extension Handler for Moving Multiple List Element Pointers, Part 3

```
// retrieve target method name
String what = e.getPropertyName();

// check if move effect (else not our problem here!)
if (e.getNewValue() instanceof Point) {
  Point old = (Point) e.getOldValue(); // old position
  Point now = (Point) e.getNewValue(); // new position
  Point diff = MSMath.diff(now, old); // difference
  if (what.startsWith("translateWithFixedTip #")) {
    // determine single fixed node number
    int num = Integer.parseInt(what.substring(23));

    // build appropriate map
    boolean[] map = new boolean[
                        boxPointer.getPointerCount()];
    // mark node for move
    map[num-1] = true;

    // move list element according to chosen map
    boxPointer.translateWithFixedTips(map, diff.x,
                                       diff.y);
  }
  else if (what.startsWith("setTip #")) {
    // determine single target node to move
    int num = Integer.parseInt(what.substring(8));

    // move the selected node
    boxPointer.setTip(num - 1,
                      MSMath.sum(boxPointer.getTip(num - 1),
                                 diff));
  }
  else if (what.startsWith("setTips ")
           || what.startsWith("translateWithFixedTips ")) {
    // act on multiple nodes at the same time
    // move nodes, or move element with fixed nodes?
    boolean setTipsMode = what.startsWith("setTips");

    // determine selected nodes
    StringTokenizer stringTok = new StringTokenizer(
                            what.substring((setTipsMode) ?
                                           7 : 22));
    int nodeCount = boxPointer.getPointerCount();

    // build map for nodes
    boolean[] map = new boolean[nodeCount];
    int currentNode = 0;
```

Listing A.9: Extension Handler for Moving Multiple List Element Pointers, Part 4

```
          // iterate over parameter to extract nodes
          while (stringTok.hasMoreTokens()) {
            // parse current node number
            currentNode = Integer.parseInt(stringTok.nextToken());
            if (currentNode>0 && currentNode <= nodeCount)
              map[currentNode-1] = true; // select for operation
          }

          if (boxPointer.getPointers() != null) {
            if (setTipsMode) { // set only tips, leave object
              int a;
              for (a=0; a < boxPointer.getPointers().length; a++)
                if (map[a]) // if marked, set it!
                  boxPointer.setTip(a,
                                          MSMath.sum(boxPointer.getTip(a),
                                                     diff));
            }
            else // move object except for mapped nodes
              boxPointer.translateWithFixedTips(map, diff.x,
                                                       diff.y);
          }
        }
      }
    }
  }
```

Listing A.10: PropertyNameMapper Implementation, Part 1

```
package animal.misc;

import java.io.PrintStream; // for status printing

import java.util.Properties; // for internal lookup

/**
 * Perform property lookup in shared properties
 *
 * @author Guido Roessling <roessling@acm.org>
 * @version 1.0 2001-11-30
 */
public class PropertyNameMapper
{
  /**
   * "forward" mapping of key to value
   */
  private Properties mapping = new Properties();

  /**
   * "reverse" mapping of value to key
   */
  private Properties reverseMapping = new Properties();

  /**
   * Insert a new mapping of key to mappedKey
   *
   * @param key the original key
   * @param mappedKey the actual key used for lookup
   */
  public void insertMapping(String key, String mappedKey)
  {
    mapping.put(key, mappedKey);
    reverseMapping.put(mappedKey, key);
  }

  /**
   * Retrieve an element at the key passed in "forward" lookup
   *
   * @param key the access key
   * @return the property name at the key, else the key itself
   */
  public String lookupMapping(String key)
  {
    if (mapping.containsKey(key))
      return mapping.getProperty(key);
    else
      return key;
  }
```

Listing A.11: PropertyNameMapper Implementation, Part 2

```
  /**
   * Retrieve an element at the key passed in "reverse" lookup
   *
   * @param key the access key
   * @return the property name at the key, else the key itself
   */
  public String lookupReverseMapping(String key)
  {
    if (reverseMapping.containsKey(key))
      return reverseMapping.getProperty(key);
    else
      return key;
  }

  /**
   * Remove the selected entry from both mappings
   *
   * @param key the key to remove
   */
  public void removeMapping(String key)
  {
    if (mapping.containsKey(key)) {
      String value = lookupMapping(key);
      reverseMapping.remove(value);
      mapping.remove(key);
    }
  }

  /**
   * Print the status of the current property mapping
   *
   * @param out the PrintStream to which to print
   * @see java.io.PrintStream
   */
  public void printMapping(PrintStream out)
  {
    out.println("straight mapping: ");
    mapping.list(out);
    out.println("reverse mapping: ");
    reverseMapping.list(out);
  }
}
```

Bibliography

[1] Daniel Mark Abrahams-Gessel. Delaunay Triangulations From Hulls. WWW: http://www.cs.dartmouth.edu/~gessel/Java/CGApp.html, 1996.

[2] Jay Martin Anderson and Thomas L. Naps. A Context for the Assessment of Algorithm Visualization System as Pedagogical Tools. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 121–130, July 2001.

[3] Woi L. Ang. Bin Sort. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/binsort.html, 1998.

[4] Woi L. Ang. Building Hash Tables. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/hash_tables.html, 1998.

[5] Woi L. Ang. Data Structures and Algorithms: Heap. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/heaps.html, 1998.

[6] Woi L. Ang. Huffman Encoding & Decoding Animation. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/huffman.html, 1998.

[7] Woi L. Ang. Matrix Chain Multiplication. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/mat_chain.html, 1998.

[8] Woi L. Ang. Quick Sort. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/qsort.html, 1998.

[9] Woi L. Ang. Radix Sort. WWW: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/radixsort.html, 1998.

[10] Apple Computers, Inc. Quicktime for Java API. WWW: http://developer.apple.com/quicktime/qtjava/index.html, 2001.

[11] Ronald Baecker. *Sorting Out Sorting*: A Case Study of Software Visualization for Teaching Computer Science. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, pages 369–381. MIT Press, 1998.

[12] Ronald Baecker and Aaron Marcus. Printing and Publishing C Programs. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 4, pages 45–62. MIT Press, 1998.

[13] Ronald Baecker and Blaine Price. The Early History of Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 2, pages 29–34. MIT Press, 1998.

[14] Ronald Baecker and Dave Sherman. Sorting out Sorting. 30 minute color film, Dynamic Graphics Project, University of Toronto (excerpted and "reprinted" in SIGGRAPH Video Review 7, 1983), 1981. Distributed by Morgan Kaufman Publishers.

[15] Ryan S. Baker, Michael Boilen, Michael T. Goodrich, Roberto Tamassia, and B. Aaron Stibel. Testers and Visualizers for Teaching Data Structures. 30th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99), New Orleans, Louisiana*, pages 261–265, March 1999.

[16] Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, Jim Winstead, Lars Torben Wilson, Rasmus Lerdorf, Zeev Suraski, and Andrei Zmievski. *PHP Manual*, 2000. WWW: `http://snaps.php.net/manual/`.

[17] John Bazik, Roberto Tamassia, Steven P. Reiss, and Andries van Dam. Software Visualization in Teaching at Brown University. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 25, pages 382–398. MIT Press, 1998.

[18] Mordechai Ben-Ari. Program Visualization in Theory and Practice. *Informatik / Informatique, Special Issue on Visualization of Software*, pages 8–11, April 2001.

[19] Benjamin S. Bloom and David R. Krathwohl. *Taxonomy of Educational Objectives; the Classification of Educational Goals, Handbook I: Cognitive Domain*. Addison-Wesley, 1956.

[20] Christopher M. Boroni, Frances W. Goosey, Michael T. Grinder, and Rockford J. Ross. A Paradigm Shift! The Internet, the Web, Browsers, Java, and the Future of Computer Science Education. 29th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98), Atlanta, Georgia*, pages 145–152, May 1998.

[21] Christopher M. Boroni, Frances W. Goosey, Michael T. Grinder, and Rockford J. Ross. Engaging Students with Active Learning Resources: Hypertextbooks for the Web. 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 65–69, February 2001.

[22] Alyce Brady. Introduction to Programming in C++ Extra Notes. WWW: `http://max.cs.kzoo.edu/CS110W01/extraSlides.shtml`, February 2001. Contains Powerpoint animations for Selection Sort, Insertion Sort and Merge Sort.

[23] Beatrix Braune and Reinhard Wilhelm. Focusing in Algorithm Explanation. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):1–7, January 2000.

[24] Stina Bridgeman, Michael T. Goodrich, Stephen G. Kobourov, and Roberto Tamassia. PILOT: An Interactive Tool for Learning and Grading. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 139–143, March 2000.

[25] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. *Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan*, pages 4–9, October 1991.

[26] Marc H. Brown. A Taxonomy of Algorithm Animation Displays. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 3, pages 35–42. MIT Press, 1998.

[27] Marc H. Brown and John Hershberger. Fundamental Techniques for Algorithm Animation Displays. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 7, pages 81–102. MIT Press, 1998.

[28] Marc H. Brown and John Hershberger. Program Auralization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 10, pages 137–144. MIT Press, 1998.

[29] Marc H. Brown and Marc A. Najork. Algorithm Animation Using Interactive 3D Graphics. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 9, pages 119–136. MIT Press, 1998.

[30] Marc H. Brown and Robert Sedgewick. A System for Algorithm Animation Structures. *ACM SIGGRAPH '84 Proceedings, Minneapolis, Minnesota*, pages 177–186, July 1984.

[31] Marc H. Brown and Robert Sedgewick. Interesting Events. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 12, pages 155–172. MIT Press, 1998.

[32] Peter Brummund. Colossians 3:17. WWW: `http://www.cs.hope.edu/~alganim/animator/Animator.html`, 1997.

[33] Peter Brummund. The Complete Collection of Algorithm Animations, 1998. WWW: `http://cs.hope.edu/~alganim/ccaa/`.

[34] Michael D. Byrne, Richard Catrambone, and John Stasko. Do Algorithm Animations Aid Learning? Technical report, Georgia Tech, August 1996. Available as Technical Report GIT-GVU-96-18 from `http://www.cc.gatech.edu/gvu/softviz/empir/empir.html`.

[35] Philippe M. Cassereau. Wavelet-based Image Coding. In Andrew B. Watson, editor, *Digital Images and Human Vision*, chapter 2, pages 13–21. MIT Press, 1993.

[36] Giuseppe Cattaneo, Umberto Ferraro, Giuseppe F. Italiano, and Vittorio Scarano. Concurrent Algorithms and Data Types Animation over the Internet. *Proceedings of the* 15th *IFIP World Computer Congress, Vienna, Austria*, 1998. Online at `http://wonderland.dia.unisa.it/catai/`.

[37] Paul Chew. Voronoi/Delaunay Applet. WWW: `http://www.cs.cornell.edu/Info/People/chew/Delaunay.html`, 1997.

[38] Arturo I. Concepcion, Nathan Leach, and Allan Knight. Algorithma 99: An Experiment in Reusability & Component Based Software Engineering. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 162–166, March 2000.

[39] Pierluigi Crescenzi, Camil Demetrescu, Irene Finocchi, and Rossella Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, April 2000.

[40] Pierluigi Crescenzi, Nils Faltin, Rudolf Fleischer, Chris Hundhausen, Stefan Näher, Guido Roessling, John Stasko, and Erkki Sutinen. The Algorithm Animation Repository. *Proceedings of the Second International Program Visualization Workshop, Århus, Denmark*, page (submitted for review), June 2002.

[41] Karel Culik II and Jarkko Kari. Digital Images and Formal Languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3 - Beyond Words, chapter 10. Springer, 1997. ftp://ftp.cs.sc.edu/pub/culik/handbook.tar.gz.

[42] Wanda Dann, Stephen Cooper, and Randy Pausch. Making the Connection: Programming With Animated Small World. 5th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland*, pages 41–44, July 2000.

[43] Adriana Cássia Rosse de Almeida. String Matching Algorithm Animation Applet. WWW: http://www.dcc.ufmg.br/~cassia/smaa/english/, (no date given).

[44] Camil Demetrescu and Irene Finocchi. Smooth Animation of Algorithms in a Declarative Framework. *Proceedings of the 1999 IEEE Symposium on Visual Languages (VL99)*, pages 280–287, 1999.

[45] Herbert L. Dershem and Peter Brummund. Tools for Web-based Sorting Animations. 29th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98), Atlanta, Georgia*, pages 222–226, May 1998.

[46] Herbert L. Dershem and James Vanderhyde. Java Class Visualization for Teaching Object-Oriented Concepts. 29th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98), Atlanta, Georgia*, pages 53–57, May 1998.

[47] Stephan Diehl, Andreas Kerren, and Reinhard Wilhelm. Ganimal. WWW: http://rw4.cs.uni-sb.de/~ganimal/, 2000.

[48] John Domingue. Viz. WWW: http://kmi.open.ac.uk/sv/viz/viz.html, 1997.

[49] John Domingue and Paul Mulholland. Staging Software Visualizations on the Web. *IEEE Symposium on Visual Languages (VL '97), Capri, Italy*, pages 368–375, 1997.

[50] Sarah A. Douglas, Christopher D. Hundhausen, and Donna McKeown. Toward Empirically-Based Software Visualization Languages. *1995 IEEE Symposium on Visual Languages*, pages 342–349, 1995.

[51] Sarah A. Douglas, Donna McKeown, and Christopher D. Hundhausen. Exploring Human Visualization of Computer Algorithms. *Proceedings of Graphics Interface '96, Toronto, Canada*, pages 9–16, 1996.

[52] David Eck. The xSortLab Applet. WWW: `http://math.hws.edu/TMCM/java/xSortLab/index.html`, 1997.

[53] Jukka Eskola and Jorma Tarhio. Animation of Flowcharts with Excel. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 59–68, July 2001.

[54] Nils Faltin. Designing Courseware on Algorithms for Active Learning with Virtual Board Games. 4th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE '99), Cracow, Poland*, pages 135–138, June 1999.

[55] Nils Faltin. Aktives Lernen von Algorithmen mit interaktiven Visualisierungen. *Tagungsband "Informatiktage 2000" der Gesellschaft für Informatik*, pages 121–137, 2000.

[56] Nils Faltin and Tobias Gross. Das Lernprogramm Heapsort-SALA 2.0. WWW: `http://OLLI.informatik.uni-oldenburg.de/heapsort_SALA`, 2000.

[57] Christine Faulkner. *The Essence of Human-Computer Interaction*. Prentice Hall, 1998.

[58] Tanya V. Filipova. Travelling Salesman Problem Animation. WWW: `http://nuweb.jinr.dubna.su/~filipova/tsp.html`, 1996.

[59] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics*. Addison-Wesley, 2 edition, 1995.

[60] Taisuke Fukuno. Hanoi Applet. WWW: `http://sariel.miyako.co.jp/~uni/Hanoi.html`, 1998.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[62] Michael R. Garey and David S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

[63] Carlisle E. George. EROSI - Visualizing Recursion and Discovering New Errors. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 305–309, March 2000.

[64] David Ginat. Misleading Intuition in Algorithmic Problem Solving. 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 21–25, February 2001.

[65] Peter A. Gloor. AACE - Algorithm Animation for Computer Science Education. *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 25–31, 1992.

[66] Peter A. Gloor. Animated Algorithms. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 27, pages 409–416. MIT Press, 1998.

[67] Peter A. Gloor. User Interface Issues For Algorithm Animation. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 11, pages 145–152. MIT Press, 1998.

[68] Frances Goosey and Chris Boroni. WebLab Program Animator. WWW: `http://www.cs.montana.edu/webworks/webworks-home/projects/program_animator/program_animator.html`, 1997.

[69] James Gosling, Jason Harrison, and Jim Boritz. Sorting Algorithms Demo. WWW: `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`, 2001.

[70] James Gosling and Kevin A. Smith. The Sorting Algorithm Demo (1.1). WWW: `http://java.sun.com/applets/jdk/1.1/demo/SortDemo/index.html`, 1996.

[71] Eric Gramond and Susan H. Rodger. Using JFLAP to Interact with Theorems in Automata Theory. 30th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99), New Orleans, Louisiana*, pages 336–340, March 1999.

[72] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of User Algorithms on the Web. *IEEE Symposium on Visual Languages*, pages 360–367, 1997.

[73] Alejo Hausner. Bubble Sort. WWW: `http://www.cs.princeton.edu/~ah/alg_anim/version1/BubbleSort.html`, 1996.

[74] Alejo Hausner. Merge sort. WWW: `http://www.cs.princeton.edu/~ah/alg_anim/version1/MergeSort.html`, 1996.

[75] Alejo Hausner. Quicksort. WWW: `http://www.cs.princeton.edu/~ah/alg_anim/version1/QuickSort.html`, 1996.

[76] Michael T. Heath, Allen D. Malony, and Diane T. Rover. Visualization for Parallel Performance Evaluation and Optimization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 23, pages 347–365. MIT Press, 1998.

[77] Thomas G. Hill. Assessing the Instructional Value of Student Predictions in Tree Animations. *Doctoral Consortium at the* 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, page vii, 2001. WWW: `http://duke.csc.villanova.edu/docConsortium/DC01/participants/hill.html`.

[78] Steven Holzner. *Inside XML*. New Riders Publishing, 2000.

[79] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2000.

[80] Cay S. Horstmann and Gary Cornell. *Core JAVA 2 - Advanced Features*. Prentice Hall, 2000.

[81] Christopher D. Hundhausen. The Search for an Empiricial and Theoretical Foundation for Algorithm Visualization. Available online at `http://lilt.ics.hawaii.edu/~hundhaus/writings/`, 1993.

[82] Christopher D. Hundhausen. Toward the Development of Highly Interactive Software Visualization Systems: A User-Centered Approach. *International Workshop on Software Visualization, SIGCHI '94, Boston, MA*, 1994.

[83] Christopher D. Hundhausen and Sarah Douglas. SALSA and ALVIS: A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations. *IEEE Symposium on Visual Languages, Los Alamitos, California*, pages 67–68, 2000.

[84] Christopher D. Hundhausen and Sarah Douglas. Using Visualizations to Learn Algorithms: Should Students Construct Their Own, or View an Expert's? *IEEE Symposium on Visual Languages, Los Alamitos, California*, pages 21–28, 2000.

[85] Christopher David Hundhausen. *Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Communication in an Undergraduate Algorithms Course*. PhD thesis, University of Oregon, 1999. Unpublished Doctoral Dissertation, available as technical report CIS-TR-99-07 (June 1999) in Department of Computer and Information Science, University of Oregon, Eugene.

[86] Ted Hung and Susan H. Rodger. Increasing Visualization and Interaction in the Automata Theory Course. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 6–10, March 2000.

[87] Christian Icking, Rolf Klein, Peter Köllner, and Lihong Ma. VoroGlide 2.0. WWW: `http://wwwpi6.fernuni-hagen.de/Geometrie-Labor/VoroGlide/`, 1996.

[88] James H. Cross II, T. Dean Hendrix, Karl S. Mathias, and Larry A. Barowski. Software Visualization and Measurement in Software Engineering Education: An Experience Report. 29th *ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico*, pages 12b1 5–10, November 1999.

[89] Achim Janser. *Entwurf, Implementierung und Evaluierung des interaktiven Lehr- und Lernsystems ViACoBi für die Visualisierung von Algorithmen der Computegraphik und Bildverarbeitung*. PhD thesis, Gerhard-Mercator-Universität Duisburg, 1998. Abstract online at `http://www.informatik.uni-duisburg.de/Info2/Janser/Janser.html`.

[90] Duane Jarc, Michael B. Feldman, and Rachelle S. Heller. Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 377–381, March 2000.

[91] Dean Jerding and John T. Stasko. Visualization of Object-Oriented Programs. WWW: `http://www.cc.gatech.edu/gvu/softviz/ooviz/ooviz.html`, 1998.

[92] Ricardo Jiménez-Peris and Marta Patiño-Martínez. Visualizing Recursion and Dynamic Memory. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 111–120, July 2001.

[93] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and J. Pacios-Martínez. VisMod: A Beginner-Friendly Programming Environment. *Proceedings of the 1999 ACM Symposium on Applied Computing, San Antonio, Texas*, pages 115–120, 1999.

[94] Biliana Kaneva and Dominique Thiébaut. Sorting Algorithms. WWW: `http://cs.smith.edu/~thiebaut/java/sort/demo.html`, 1997.

[95] Alex Karweit and Robyn Geer. JELLRAP 1.0. WWW: `http://www.cs.duke.edu/~rodger/tools/jellrap/index.html`, 1997.

[96] Frank Katritzke. *Refinements of Data Compression Using Weighted Finite Automata*. PhD thesis, University of Siegen, Germany, 2001.

[97] Charles F. Keleman and Eugene R. Turk. Visual Active Scaffolding. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 179–192, July 2001.

[98] Sami Khuri. Designing Effective Algorithm Visualizations. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 1–12, February 2001.

[99] Sami Khuri. A User-Centered Approach for Designing Algorithm Visualizations. *Informatik / Informatique, Special Issue on Visualization of Software*, pages 12–16, April 2001.

[100] Sami Khuri and Hsiu-Chin Hsu. Visualizing the CPU Scheduler and Page Replacement Algorithms. 30th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99), New Orleans, Louisiana*, pages 227–231, March 1999.

[101] Sami Khuri and Hsiu-Chin Hsu. Interactive Packages for Learning Image Compression Algorithms. 5th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland*, pages 73–76, 2000.

[102] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 European Conference on Object-Oriented Programming, Jyväskylä, Finland*, pages 200–242. Springer-Verlag Berlin Heidelberg, June 1997.

[103] Andrew Kitchen. Sorting Algorithms. WWW: `http://www.cs.rit.edu/~atk/Java/Sorting/sorting.html`, (no date given).

[104] Ken Knowlton. L6: Bell Telephone Laboratories Low-Level Linked List Language. 16 minute black-and-white film, Murray Hill, N. J., 1966.

[105] Ken Knowlton. L6: Part II. An Example of L6 Programming. 30 minute black-and-white film, Murray Hill, N. J., 1966.

[106] Donald E. Knuth. *The TEXBook*. Addison-Wesley, May 1986.

[107] Boris Koldehofen, Marina Papatriantafilou, and Philippas Tsigas. Distributed Algorithms Viisualisation for Educational Purposes. 4th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE '99), Cracow, Poland*, pages 103–106, June 1999.

[108] Paul LaFollette, James Korsh, and Raghvinder Sangwan. A Visual Interface for Effortless Animation of C/C++ Programs. *Journal of Visual Languages and Computing*, 11(1):27–48, February 2000.

[109] Hans-Werner Lang. Algorithmen. WWW: `http://www.iti.fh-flens-burg.de/lang/algorithmen/algo.htm`, 2001.

[110] Matti Lattu, Veijo Meisalo, and Jorma Tarhio. On Using a Visualization Tool as a Demonstration Aid. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 141–162, July 2001.

[111] Jari M. Lavonen, Veijo P. Meisalo, and Matti P. Lattu. Visual Programming: Basic Structures Made Easy. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 163–178, July 2001.

[112] Geoff Leach. Delauney Triangulation. WWW: `http://goanna.cs.rmit.edu.au/~gl/research/comp_geom/delaunay/delaunay.html`, 1996.

[113] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. An Extended Experiment with Jeliot 2000. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 131–140, July 2001.

[114] Henry Lieberman and Christopher Fry. ZStep 95: A Reversible, Animated Source Code Stepper. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 19, pages 277–292. MIT Press, 1998.

[115] Linda Luo, Mervyn Ng, and Woi L. Ang. Red Black Tree. WWW: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/red_black.html`, 1998.

[116] J. D. McWhirter. AlgorithmExplorer: A Student-Centered Algorithm Animation System. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 174–181, Washington, September 1996. IEEE Computer Society Press.

[117] Kurt Mehlhorn and Stefan Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 2000.

[118] D. Merlini, S. Petruzzi, R. Sprugnoli, and M. C. Verri. A System for Algorithms' Animation. *Proceedings of IEEE Multimedia Systems 1999, Florence, Italy*, pages 1033–1034, 1999.

[119] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[120] Ramin Miraftabi. Intelligent Agents in Program Visualizations: A Case Study with Seal. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 53–58, July 2001.

[121] Sandeep Mitra. Java Sorting Animation Page - Bubble Sort. WWW: `http://www.cs.brockport.edu/cs/javasort.html`, 1999.

[122] Sandeep Mitra. Java Sorting Animation Page - Racing Sorts Page! WWW: `http://www.cs.brockport.edu/cs/java/apps/sorters/race_sorters/sortchoiceinp.html`, 1999.

[123] John Morris. Data Structures and Algorithms. WWW: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/index.html`, 1998.

[124] John Morris and Woi L. Ang. Optimal Binary Search Tree. WWW: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/opt_bin.html`, 1998.

[125] Paul Mulholland and Stuart Watt. Learning by Building: A Visual Modelling Language for Psychology Students. *Journal of Visual Languages and Computing*, 11(5):481–504, October 2000.

[126] David R. Musser. Advanced programming - rensselaer csci-6090. WWW: `http://www.cs.rpi.edu/~musser/ap/index_16.html`, December 2000. Contains a binary tree searching, insertion and deletion animation in Powerpoint.

[127] MySQL Group. *MySQL Reference Manual*, 2001. WWW: `www.mysql.com`.

[128] Fernando Naharro-Berrocal, Cristóbal Pajera-Flores, and J. Ángel Velázquez-Iturbide. Foundations for the Automatic Construction of Animations and their Application to Functional Programs. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 29–40, July 2001.

[129] Fernando Naharro-Berrocal, Cristóbal Pareja-Flores, J. Ángel Velázquez-Iturbide, and Margarita Martínez-Santamarta. Automatic Web Publishing of Algorithm Animations. *Informatik / Informatique, Special Issue on Visualization of Software*, pages 41–45, April 2001.

[130] Fernando Naharro-Berrocal, Cristóbal Pareja-Flores, and Jose Ángel Velázquez-Iturbide. Automatic Generation of Algorithm Animations in a Programming Environment. 30th *ASEE/IEEE Frontiers in Education Conference, Kansas City, Missouri*, pages S2C 6–12, October 2000.

[131] Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.

[132] Thomas L. Naps. Algorithm visualizations for CMSC34 - Algos and Data Structures. WWW: `http://gaigs.cmsc.lawrence.edu/cmsc34/AVClient.html`, 1998.

[133] Thomas L. Naps. A Java Visualizer Class: Incorporating Algorithm Visualisations into Students' Programs. 3rd *Annual ACM SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education, Dublin, Ireland (ITiCSE '98)*, pages 181–184, September 1998.

[134] Thomas L. Naps. Incorporating Algorithm Visualization into Educational Theory: A Challenge for the Future. *Informatik / Informatique, Special Issue on Visualization of Software*, pages 17–21, April 2001.

[135] David Neto. MergeSort demo with comparison bounds. WWW: `http://www.cs.toronto.edu/~neto/teaching/238/16/mergesort.html`, 1996.

[136] Mervyn Ng. Minimum Spanning Tree Animation. WWW: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/mst.html`, 1998.

[137] Mervyn Ng and Woi L. Ang. Dijkstra's Algorithm. WWW: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/dijkstra.html`, 1998.

[138] Dung Nguyen and Stephen B. Wong. Design Patterns for Sorting. *32nd ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 263–267, 2001.

[139] James Noble. Basic Relationship Patterns. *Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97), Munich, Germany*, 1997.

[140] Rainer Oechsle and Dieter Jahn. *Vi*sualisierung von *a*usgewählten Lehrinhalten *d*er *I*nformatik - Beispielanimationen aus dem Projekt Vivaldi. *Tagungsband GI-Workshop Softwarevisualisierung (SV 2000)*, page ??, 2000. Published as Technical Report A01/00, FR 6.2 Informatik, Universität des Saarlandes.

[141] Gabriele Ortmann. The Sorting Algorithm Demo (1.1). WWW: `http://archiv.informatik.fh-augsburg.de/informatik/projekte/meile/emiel/sort/bin/sort.html`, 2001.

[142] Mathew Palakal. Graphical Linear Queue Demo. WWW: `http://www.cs.iupui.edu/cgi-bin/cgiwrap/pacer/build_applet_page.cgi?topic=datastructs&subtopic=queue&spec=LinearQueue`, 1997.

[143] Mathew Palakal. Graphical Stack Demo. WWW: `http://www.cs.iupui.edu/cgi-bin/cgiwrap/pacer/build_applet_page.cgi?topic=datastructs&subtopic=stack&spec=StackDemo`, 1997.

[144] Mathew J. Palakal, Frederick W. Meyers, and Carla L. Boyd. An Interactive Learning Environment for Breadth-First CS Curriculum. *29th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98), Atlanta, Georgia*, pages 1–6, May 1998.

[145] Athanasios Papagelis. Backtracking. WWW: `http://students.ceid.upatras.gr/~papagel/project/kef5_8.htm`, 1998.

[146] Athanasios Papagelis. Minimum Genetic Tree Animation: Kruskal Algorithm. WWW: `http://students.ceid.upatras.gr/~papagel/project/kruskal.htm`, 1998.

[147] Athanasios Papagelis. Minimum Genetic Tree Animation: Prim Algorithm. WWW: `http://students.ceid.upatras.gr/~papagel/project/prim.htm`, 1998.

[148] Marta Patiño-Martínez and Ricardo Jiménez-Peris. Visual Debugging of Functional Programs. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 101–110, July 2001.

[149] Wim De Pauw, Doug Kimelman, and John Vlissides. Visualizing Object-Oriented Software Execution. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 22, pages 239–346. MIT Press, 1998.

[150] Connie Peng. Convex Hull Demo. WWW: `http://www.cs.umd.edu/~peng/cmsc754/animated/convexhull_anim.html`, (no date given).

[151] Frank Peters. Visualisierung von Algorithmen - Systematik und Bestandsaufnahme. Master's thesis, Carl von Ossietzky-Universität Oldenburg, 2001. Available online at `http://www-cg-hci.informatik.uni-oldenburg.de/~da/peters/Kalvin`.

[152] W. Pierson and S. H. Rodger. Web-based Animation of Data Structures Using JAWAA. 29th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98), Atlanta, Georgia*, pages 267–271, 1998.

[153] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser, editors. *The C++ Standard Template Library*. Prentice Hall, 2000.

[154] Sandeep Poonen. Animation of Sort Algorithms. WWW: `http://blackcat.brynmawr.edu/~spoonen/JavaProject/sorter.html`, 1998.

[155] B. A. Price, R. M. Baecker, and I. S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–264, 1993.

[156] Blaine Price, Ronald Baecker, and Ian Small. An Introduction to Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 1, pages 3–27. MIT Press, 1998.

[157] Robert Rasala. Automatic Array Algorithm Animation in C++. 30th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99), New Orleans, Louisiana*, pages 257–260, March 1999.

[158] Ken Rebman. PowerPoint info. WWW: `http://piglet.uccs.edu/~haefner/math511/tools/presentations/presentations.htm`, June 1999. Contains a Powerpoint animation of Pascal's Triangle.

[159] Steven Robbins. The JOTSA Animation Environment. *Proceedings of the* 31st *Annual Hawaii International Conference on Systems Sciences*, pages 655–664, 1998.

[160] Susan Rodger. Integrating Animations into Courses. 1st *Annual ACM SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education (ITiCSE '96), Barcelona, Spain*, pages 72–74, June 1996.

[161] Susan Rodger. JAWAA and Other Resources, 1997. Available at `http://www.cs.duke.edu/~rodger/tools/tools.html`.

[162] Susan H. Rodger. Pâté. WWW: `http://www.cs.duke.edu/~rodger/tools/pateweb/`, 2000.

[163] Gruia-Catalin Roman. Declarative Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 13, pages 173–186. MIT Press, 1998.

[164] Gruia-Catalin Roman and K. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22(10):25–36, October 1989.

[165] Gruia-Catalin Roman, K. Cox, C. Wilcox, and J. Plun. Pavane: A System for Declarative Visualization of Concurrent Computation. *Journal of Visual Languages and Computing*, 3(1):161–193, January 1992.

[166] Friedrich Roschmann. The Towers from Hanoi in JavaScript! WWW: `http://privat.schlund.de/R/RoschmannFriedrich/`, 1996.

[167] Rockford J. Ross. Hypertextbooks for the Web. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 221–233, July 2001.

[168] Guido Rößling. Algorithm Animation Repository. WWW: `http://www.animal.ahrgr.de/en/AnimList.html`, 2001.

[169] Guido Rößling. ANIMAL Home Page. WWW: `http://www.animal.ahrgr.de`, 2001.

[170] Guido Rößling. Key Decisions in Adopting Algorithm Animations for Teaching. *Open IFIP-GI Conference on Social, Ethical and Cognitive Issues of Informatics and Information and Communication Technologies (SEC III), Dortmund, Germany*, page (accepted and in print), July 2002.

[171] Guido Rößling. ANIMAL-FARM: An Extensible Framework for Algorithm Animation. *Second International Program Visualization Workshop, Århus, Denmark*, page (submitted and under review), June 2002.

[172] Guido Rößling and Jens Falk. ANIMAL sorting algorithms demo applet. WWW: `http://www.animal.ahrgr.de/sortDemoApplet.html`, December 2001.

[173] Guido Rößling and Bernd Freisleben. Approaches for Generating Animations for Lectures. *Proceedings of the* 11th *Society for Information Technology and Teacher Education Conference*, pages 809–814, February 2000.

[174] Guido Rößling and Bernd Freisleben. Experiences In Using Animations in Introductory Computer Science Lectures. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 134–138, March 2000.

[175] Guido Rößling and Bernd Freisleben. Flexible Generation of Animations Using ANIMAL. *International Conference on Mathematics / Science Education & Technology*, page 432, February 2000.

[176] Guido Rößling and Bernd Freisleben. Generación de visualizaciones de software usando ANIMALSCRIPT. *Novática*, 1(150):38–42, April 2001.

[177] Guido Rößling and Bernd Freisleben. Program Visualization Using ANIMALSCRIPT. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 41–52, July 2001.

[178] Guido Rößling and Bernd Freisleben. Software Visualization Generation Using ANIMALSCRIPT. *Informatik / Informatique, Special Issue on Visualization of Software*, 8(2):35–40, April 2001.

[179] Guido Rößling and Bernd Freisleben. ANIMALSCRIPT: An Extensible Scripting Language for Algorithm Animation. 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 70–74, February 2001.

[180] Guido Rößling and Bernd Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):(in print), April 2002.

[181] Guido Rößling and Thomas L. Naps. A Testbed for Pedagogical Requirements in Algorithm Visualizations. *7th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), Århus, Denmark*, page (accepted and in print), June 2002.

[182] Guido Rößling and Thomas L. Naps. Towards Intelligent Tutoring in Algorithm Visualization. *Second International Program Visualization Workshop, Århus, Denmark*, page (submitted and under review), June 2002.

[183] Guido Rößling, Markus Schüler, and Bernd Freisleben. ANIMAL: A New Interactive Modeler for Animations in Lectures. *31st ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, page 437, March 2000.

[184] Guido Rößling, Markus Schüler, and Bernd Freisleben. The ANIMAL Algorithm Animation Tool. *5th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland*, pages 37–40, July 2000.

[185] Robert Sedgewick. *Algorithms in Java, Parts 1–4*. Addison-Wesley, Reading, MA, USA, 1998.

[186] Robert D. Sedgewick. Shellsort Animations. WWW: `http://www.cs.princeton.edu/~rs/shell/animate.html`, 1996.

[187] Hiromasa Sekisita. Breadth-First Search Animation. WWW: `http://al.ei.tuat.ac.jp/~sekisita/graph2-e.html`.

[188] Hiromasa Sekisita. Depth-First Search Animation. WWW: `http://al.ei.tuat.ac.jp/~sekisita/graph-e.html`.

[189] Hiromasa Sekisita. Example of an Algorithm Animation: Bubble Sort. WWW: `http://al.ei.tuat.ac.jp/~sekisita/bsort-e.html`.

[190] Hiromasa Sekisita. Example of an Algorithm Animation: Insertion Sort. WWW: `http://al.ei.tuat.ac.jp/~sekisita/isort-e.html`.

[191] Hiromasa Sekisita. Example of an Algorithm Animation: Selection Sort. WWW: `http://al.ei.tuat.ac.jp/~sekisita/ssort-e.html`.

[192] Hiromasa Sekisita. My Algorithm Animation System. WWW: `http://al.ei.tuat.ac.jp/~sekisita/aas/index.html`.

[193] Matthew Smith. J-Updater – A Generic Java-based Software Updater. Technical report, Department of Electrical Engineering and Computer Science, University of Siegen, Germany, October 2001.

[194] John Stasko. Using Student-built Algorithm Animations as Learning Aids. *28th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, California*, pages 25–29, February 1997.

[195] John Stasko. Building Software Visualizations through Direct Manipulation and Demonstration. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 14, pages 187–203. MIT Press, 1998.

[196] John Stasko. Samba Algorithm Animation System, 1998. Available at `http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html`.

[197] John Stasko. Smooth Continuous Animation for Portraying Algorithms and Processes. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 8, pages 103–118. MIT Press, 1998.

[198] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[199] John T. Stasko. PVaniM 2.0 Visualization Package. WWW: `http://www.cc.gatech.edu/gvu/softviz/parviz/pvanimOL/pvanimOL.html`, (no date given).

[200] Stauff. Bewegte Mathematik. WWW: `http://www.muenster.de/~stauff/bmru.htm`, (no date given).

[201] Linda Stern, Harald Søndergaard, and Lee Naish. A Strategy for Managing Content Complexity in Algorithm Animation. 4th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'99), Cracow, Poland*, pages 127–130, September 1999.

[202] Sun Microsystems, Inc. Jimi Software Development Kit. WWW: `java.sun.com/products/jimi`, February 2000.

[203] Chien Wei Tan. Quick Sort. WWW: `http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/Java/q_sort/tqs_new.html`, 1998.

[204] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2 edition, 2001.

[205] Minoru Terada. Animating C Programs in Paper-Slide-Show. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 79–88, July 2001.

[206] Joseph A. Turner and Joseph L. Zachary. Javiva: A Tool for Visualizing and Validating Student-Written Java Programs. 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 45–49, February 2001.

[207] Jose Ángel Velázquez-Iturbide and Antonio Presa-Vázquez. Customization of Visualizations in a Functional Programming Environment. 29th *ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico*, pages 12b3 22–28, November 1999.

[208] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[209] Stuart N. K. Watt. Froglet: A Source-Level Stepper for Lisp. Technical Report KMI-TR-10, Human Cognition Research Laboratory, The Open University, UK, 1994.

[210] William Yurcik and Larry Brumbaugh. A Web-Based Little Man Computer Simulator. 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 204–208, 2001.

[211] Steven J. Zeil. ALGAE v2.0 - Sorting Demo. WWW: `http://www.cs.odu.edu/~zeil/AlgAE/WeissJavaSort/sortdemo.html`, 2000.

[212] Andreas Zeller. Animating Data Structures in DDD. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 69–78, July 2001.

[213] Romuald J. Żyłła. Tower of Hanoi in JS. WWW: `http://chemeng.p.lodz.pl/zylla/games/hf.html`, 1999.