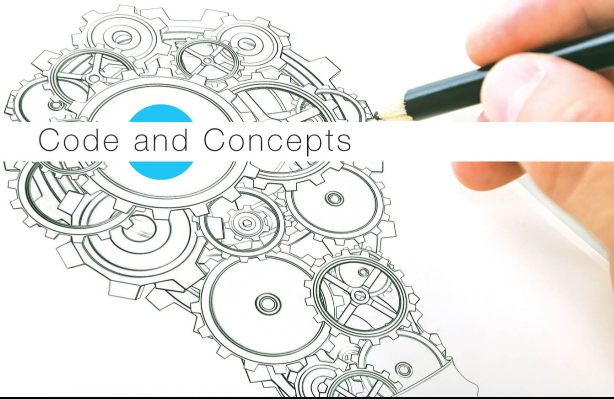
HOLGER
GAST

HOW TO USE OBJECTS

Code and Concepts



How to Use Objects

This page intentionally left blank

How to Use Objects

Code and Concepts

Holger Gast

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Cataloging-in-Publication Data

Names: Gast, Holger, 1975- author.
Title: How to use objects : code and concepts / Holger Gast.
Description: New York : Addison-Wesley Professional, 2015. | Includes
  bibliographical references and index.
Identifiers: LCCN 2015038126 | ISBN 9780321995544 (hardcover : alk. paper)
Subjects: LCSH: Object-oriented programming (Computer science)
Classification: LCC QA76.64 .G39 2015 | DDC 005.1/17—dc23
LC record available at http://lccn.loc.gov/2015038126

ISBN-13: 978-0-321-99554-4
ISBN-10: 0-321-99554-6
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

To Dorothea, Jonathan, and Elisabeth
—HG

This page intentionally left blank

Contents

Preface

In roughly 15 years of teaching software engineering subjects at the University of Tübingen, from introductory programming courses through software engineering to software architecture, with a sideline on formal software verification, I have learned one thing: It is incredibly hard for those with basic—and even advanced—programming skills to become professional developers.

A professional developer is expected to deliver workable solutions in a predictable and dependable fashion, meeting deadlines and budgets, fulfilling customer expectations, and all the while writing code that is easy to maintain, even after the original project has long been declared finished.

To achieve all of this, the professional developer has to know both concepts and code. The concepts of software engineering, software design, and software architecture give high-level direction toward the goal and provide guidelines toward achieving it. Above all, they provide recurring solution patterns that are known to work and that other professionals will recognize. The concrete coding techniques must complement this knowledge to create good software. The guidelines come with many pitfalls and easy misconceptions, and the patterns must be put into a concrete shape that follows implicit conventions to be recognized. This is the second thing I have learned: It is incredibly hard to translate good concepts to good code.

I have written this book to present professional strategies and patterns side by side with professional code, in the hope of providing precisely the links and insights that it takes to become a professional developer. Rather than using classroom-sized toy examples and leaving the remainder to the reader's imagination, I select and analyze snippets from the code base of the Eclipse IDE. In many cases, it is the context of the nontrivial application that explains why one code structure is good, while a very similar structure fails.

This page intentionally left blank

Acknowledgments

In finishing the book, I am deeply grateful to many people. To my academic advisor, Professor Herbert Klaeren, who taught me how to teach, encouraged me to pick practically relevant topics for my lectures, and improved the original manuscript by reading through every chapter as it came into existence. To my editor, Christopher Guzikowski, for trusting me to write this book and for being generous with his advice and guidance in the writing process. To the reviewers, who have dedicated their time to help me polish the manuscript into a book. To my wife, Dorothea, who taught me how to write, encouraged me to write, and suffered the consequences gladly. And finally, to my students, who entrusted me with their feedback on and criticism of my lectures, and who were always eager to discuss their design proposals and solutions freely. The core idea of this book, to present code and concepts side by side, would not have been possible without these constant and supportive stimuli.

This page intentionally left blank

About the Author

Holger Gast graduated with a degree in computer science from the University of Tübingen, Germany, in 2000, and received a Ph.D. with a dissertation on type systems for programming languages in 2005 (Tübingen). As a post doctoral fellow, he worked on formal correctness proofs for software and finished his Habilitation for Computer Science in 2012 (Tübingen).

Since 2000, he has being teaching in the area of software engineering at different levels of the computer science curriculum, starting from introductory programming courses to lectures on software design and architecture. His other interests include scientific databases for the humanities and the model-driven construction of data-driven web applications.

This page intentionally left blank

Introduction

What makes a professional developer? The short answer is obvious: A professional developer produces good-quality code, and reliably so. It is considerably less obvious how the professional developer achieves this. It is not sufficient to know all the technical details about a language and its frameworks, because this does not help in strategic decisions and does nothing for the communication within a team. It is also not sufficient to know the buzz words of design and architecture, because they give no hints as to the concrete implementation. It is not sufficient to read through catalogs of design patterns, because they focus on particular challenges and are easily misunderstood and misused if seen out of context. Instead, the professional developer has to have a firm grasp of all of these areas, and many more. He or she must see the connections and must be able to switch between the different perspectives at a moment's notice. The code they produce, in the end, is just a reflection of a large amount of background considerations on many different details, all of which are interconnected in often subtle ways.

This book aims to cover some of the difficult terrain found along the path to professionalism that lies ahead of a developer who has just finished an introductory course on programming, a university curriculum on computer science, or a first job assignment. It presents the major topics that have proved relevant in around 30 years since the mainstream adoption of object-oriented development. Beyond that, it highlights their crucial points based on my 15 years of experience in teaching software development at all levels of a university curriculum and working through many and various software projects.

The Central Theme: Code and Concepts

The main theme of this book is that object-oriented development, and software development in general, always requires a combination of concepts and code. Without code, there will obviously be no software. Without concepts, the code will have an arbitrary, unpredictable structure. Concepts enable us to talk about the code and to keep it understandable and maintainable. They support us in making design and implementation decisions. In short, they explain why the code looks the way it does.

The field of object-oriented development offers a particularly fair amount of time-proven concepts. Here are just a few examples. At the smallest scale, the idea of replacing "method calls" with "messages" helps to keep objects independent. The approach of designing objects to take on "responsibilities" in a larger network of objects explains how even small objects can collaborate to create substantial applications. It then turns out that networks of objects often follow "patterns" such that standard problems can be solved consistently and reliably. The idea of describing method calls by "contracts" gives a consistent guide for obtaining correct code. “Frameworks” and “inversion of control” have become essential for building large applications effectively.

Concepts are useful and even necessary for writing good-quality object-oriented code, but it takes a fair amount of diligence, insight, and experience to translate them into code faithfully. Teaching experience tells us that the concepts are easily misunderstood and that subtle deviations can sometimes have disastrous consequences. In fact, the same lesson applies to many tutorials and introductory expositions. For instance, the famous MODEL-VIEW-CONTROLLER pattern is often given with a “minimal” example implementation. We have seen several cases where the model holds a reference to the concrete view class, and a single instance, too. These blunders break the entire pattern and destroy its benefits. The fact that the code works is just not good enough for professional developers.

Because code and concepts are both essential and must be linked in detail, this book always takes you all the way. For each topic, we introduce the central concepts and explain the general lay of the land with a few illustrations. But then we go on immediately to show how the concepts are rendered in concrete code. We do not stop at giving minimal examples but also explore the more intricate points. In the example of the MODEL-VIEW-CONTROLLER pattern, it is easy to get right for small examples. But as soon as models get more complex, the professional developer makes sure that only those parts that have changed are repainted. Similarly, attaching an event-listener to a button in the user interface is simple enough, but the professional must avoid freezing the display by executing long-running operations. This, in turn, requires concurrent execution.

Of course, there might still be the danger of oversights in “minimal” examples. Wherever feasible, we therefore present code taken from the Eclipse platform and highlight those elements that exhibit the concept at hand. This choice has a further advantage: It shows the concepts in action and in context. Very often, the true value of an approach, and sometimes even its justification, shows up only in really large applications. For instance, it is essential to keep software extensible. Convincing examples of extensibility can, however, be found only in modular systems such as Eclipse. Finally, if you want to dig a bit deeper into a particularly interesting point, you can jump right into the referenced sources.

In connection with the resulting code, there is one final story that is usually not told: the story of how the code actually gets written. Professional developers can become amazingly productive if they do not insist on typing their code, but know all the tricks that will make their IDE generate the code for them. For instance, knowing about the concept of “refactoring” is all right and useful. But professionals must also master the refactoring tools in Eclipse, up to the point where they recognize that three particular tools in sequence will bring about the desired code change. On the theme of code and concepts, we will therefore also highlight the Eclipse tools that apply to each concept.

The Structure of the Book

The book is organized in four parts. They approach the topic of object-oriented development by moving roughly from the “small” aspects of individual language elements to the “large” aspects of design and architecture. However, they also provide complementary answers to the same question: What does a professionally designed “object” really look like?

Part I: Language Usage Professional code always starts with professional language usage: A professional applies the language elements according to their intentions, rather than misusing them for seemingly nifty tweaks and hacks. The term "usage" is actually meant as in "usage dictionary" for natural languages; that is, if code obeys the idioms, the phrases, and the hidden connotations of the language constructs, it becomes more readable, understandable, and maintainable.

Part II: Contracts Professional code must above all be reliable. It must work in all situations that it is constructed for and it must be clear what these situations really are. The idea of design-by-contract gives a solid foundation for the necessary reasoning. It carries all the way from high-level descriptions of methods down to the details of formal software verification. As a complementary approach, the behavior of objects must be established by comprehensive testing.

Part III: Events Software of any size is usually event-driven: The application functionality is triggered by some framework that establishes the overall structure and fundamental mechanisms. At the core, the interpretation of methods changes, compared to Part II: A method does not implement a service that fulfills a specific request by the caller, but a reaction that seems most suitable to the callee. We follow this idea in the particular area of user interfaces and also emphasize the architectural considerations around the central model-view separation in that area. Because almost all applications need to do multiple things at once, we also include a brief introduction to multithreading.

Part IV: Responsibility-Driven Design One goal of object-oriented development is to keep the individual objects small and manageable. To achieve a task of any size, many objects must collaborate. The metaphor of assigning "responsibilities" to individual objects within such larger networks has proved particularly useful and is now pervasive in software engineering. After an introductory chapter on designing objects and their collaborations, we explore the ramifications of this approach in taking strategic and architectural decisions.

Together, the four parts of this book are designed to give a comprehensive view of object-oriented development: They explain the role of individual objects in the overall application structure, their reactions to incoming events, their faithful fulfillment of particular service requests, and their role in the larger context of the entire application.

How to Read the Book

The topic of object-oriented software development, as described previously, has many facets and details. What is more, the individual points are tightly interwoven to form a complex whole. Early presentations of object-oriented programming tended to point out that it takes an average developer more than a year in actual projects to obtain a sufficient overview of what this approach to programming truly entails. Clearly, this is rather unsatisfactory.

The book makes an effort to simplify reading as much as possible. The overall goal is to allow you to use the book as a reference manual. You can consult it to answer concrete questions without having to read it cover-to-cover. At the same time, the book is a proper conventional textbook: You can also follow longer and more detailed discussions through to the end. The central ingredients to this goal are the following reading aids.

Layered Presentation The presentation within each chapter, section, and subsection proceeds from the general points to the details, from crucial insights to additional remarks. As a result, you can stop reading once you feel you have a sufficient grasp on a topic and come back later for more.

Core Sections Each chapter starts with a self-contained section that explains the chapter's core concepts. The intention is that later chapters can be understood after reading the core sections of the earlier ones. By reading the core sections of all chapters, you get a "book within a book"—that is, a high-level survey of object-oriented software development. The core sections themselves are kept to a minimum and should be read through in one go.

Snappy Summaries Every point the text explains and elaborates on is headed by a one-sentence summary, set off visually in a gray box. These snappy summaries give a quick overview of a topic and provide landing points for jumping into an ongoing discussion.

Self-Contained Essays All top-level sections, and many subsections, are written to be self-contained treatments of particular topics. After reading the core section of a chapter, you can usually jump to the points that are currently most interesting.

Goal-Oriented Presentation The book's outline reflects particular goals in development: How to write good methods? How to use inheritance and interfaces? How to structure an application? How to use multithreading? How to work with graphical user interfaces? How to obtain flexible software? Everything else is subsumed under those goals. In particular, design patterns are presented in the context to which they contribute most. They are kept very brief, to convey the essential point quickly, but the margin always contains a reference to the original description for completeness.

Extensive Cross-Referencing Jumping into the midst of a discussion means you miss reading about some basics. However, chances are you have a pretty good idea about those anyway. To help out, all discussions link back to their prerequisites in the margin. So if you stump upon an unknown concept, you know where to look it up. It is usually a good idea to read the core section of the referenced chapter as well. In the other direction, many of the introductory topics have forward pointers to more details that will give additional insights. In particular, the core sections point to further information about individual aspects.

The cross-referencing in the margin uses the following symbols:

📖	Reference to literature with further information or seminal definitions, ordered by relevance
↰	Reference to previous explanations, usually prerequisites
»	Reference to later material that gives further aspects and details

Furthermore, many paragraphs are set apart from the normal presentation by the following symbols:

Crucial details often overlooked by novices. When missed, they break the greater goals of the topic.

An insight or connection with a concept found elsewhere. These insights establish the network of concepts that make up the area of object-oriented development.

An insight about a previous topic that acquires a new and helpful meaning in light of the current discussion.

An additional remark about some detail that you may or may not stumble over. For instance, a particular detail of a code snippet may need further explanation if you look very closely.

A decision-making point. Software development often involves decisions. Where the normal presentation would gloss over viable alternatives, we make them explicit.

A nifty application of particular tools, usually to boost productivity or to take a shortcut (without cutting corners).

A (small) overview effect [259] can be created by looking at a language other than Java or by moving away from object-oriented programming altogether. Very often, the specifics of objects in Java are best appreciated in comparison.

Hints for Teaching with the Book

The book emerged from a series of lectures given by the author in the computer science curriculum at the University of Tübingen between 2005 and 2014. These lectures ranged from introductory courses on programming in Java through object-oriented programming and software engineering to software architecture. For this book, I have chosen those topics that are most likely to help students in their future careers as software developers. At the same time, I have made a point of treating the topics with the depth that is expected of university courses. Particularly intricate aspects are, however, postponed to the later sections of each chapter and can be omitted if desired.

If you are looking at the book as a textbook for a course, it may be interesting to know that the snappy summaries actually evolved from my transparencies and whiteboard notes. The style of the lectures followed the presentation in the book: After explaining the conceptual points, I reiterated them on concrete example code. The code shown in the book is either taken from the Eclipse platform or available in the online supplement.

The presentation of design patterns in this book, as explained earlier, is geared toward easy reading, a focus on the patterns' main points, and close links to the context to which the patterns apply. An alternative presentation is, of course, a traditional one as given in [100,59,263], with a formalized structure of name, intent, motivation, structure, down to consequences and related patterns. I have chosen the comparatively informal approach here because I have found that it helped my students in explaining the purpose and the applications of patterns in oral exams and design exercises. In larger courses with written exams, I have often chosen a more formalized presentation to allow students to better predict the exam and to prepare more effectively. For these cases, each pattern in the book points to its original publication in the margin.

The layered presentation enables you pick any set of topics you feel are most appropriate for your particular situation. It may also help to know which sections have been used together in which courses.

CompSci2 This introductory programming course is mostly concerned with the syntax and behavior of Java and the basics of object-oriented programming (Section 1.3, Section 1.4, Section 1.6, Section 1.5). I have included event-based programming of user interfaces (Section 7.1) because it tends to be very motivating. Throughout, I have used the view of objects as collaborating entities taking on specific responsibilities (Section 11.1). This overarching explanation enabled the students to write small visual games at the end of the course.

Software Engineering The lecture gives a broad overview of practical software engineering so as to prepare the students for an extended project in the subsequent semester. I have therefore focused on the principles of object-oriented design (Section 11.1, Section 11.2.1, Section 11.5.1). To give the students a head start, I have covered those technical aspects that would come up in the projects—in particular, graphical user interfaces (Section 7.1), including the principle of model-view separation (Section 9.1, Section 9.2.1), the challenges of frameworks (Section 7.3), and the usability issue of long-running jobs (Section 7.10). I have also covered the fundamental design principles leading to maintainable code (Section 11.5), focusing on the Single Responsibility Principle (Section 11.2.1) for individual objects and the Liskov Substitution Principle for hierarchies (Section 3.1.1). Throughout, I have discussed prominent patterns—in particular, OBSERVER (Section 2.1), COMPOSITE (Section 2.3.1), ADAPTER (Section 2.4.1), PROXY (Section 2.4.3), LAYERS (Section 12.2.2), and PIPES-AND-FILTERS (Section 12.3.4).

Object-Oriented Programming This bachelor-level course builds on CompSci2 and conveys advanced programming skills. We have treated object-oriented design (Section 11.1, Section 11.2.1, Section 11.3.2, Section 11.3.3) and implementation (Section 1.2.1, Sections 1.3–1.8) in some depth. Because of their practical relevance, we have covered user interfaces, including custom-painted widgets and the MODEL-VIEW-CONTROLLER pattern (Section 7.1, Section 7.2, Section 7.5, Section 7.8, Section 9.2). Finite State Machines served as a conceptual basis for event-based programming (Chapter 10). As a firm foundation, I have included a thorough treatment of contracts and invariants, including the practically relevant concept of model fields (Section 4.1). I have found that practical examples serve well to convey these rather abstract topics (Section 4.2) and that interested students are happy to follow me into the realm of formal verification (Section 4.7.2).

Software Architecture 1 This lecture treats fundamental structuring principles for software products. Because of the varying backgrounds of students, I started with a brief survey of object-oriented design and development (Section 11.1, Section 11.3.2, Section 10.1). This was followed by the basic architectural patterns, following [59] and [218]: LAYERS, PIPES-AND-FILTERS, MODEL-VIEW-CONTROLLER, and INTERCEPTOR (Section 12.2.2, Section 9.2, Section 12.3.4, Section 12.3.2). Because of their practical relevance, I included UNDO/REDO (Section 9.5) and the overall structure of applications with graphical interfaces (Section 9.4). The course ended with an outlook on design for flexible and in particular extensible and reusable software (Section 12.2, Section 12.3, Section 12.4).

Software Architecture 2 This lecture covers concurrent programming and distributed systems. For space reasons, only the first area is included in the book (Section 7.10, Chapter 8).

This page intentionally left blank

Chapter 9

Structuring Applications with Graphical Interfaces

Chapter 7 introduced the technical and conceptual basis for building user interfaces using the SWT framework that comes with Eclipse. At the core, development comprises two aspects: setting up a widget tree with layout information to create the visual appearance, and attaching event-listeners to the individual widgets to implement the application's reaction to user input. Although this seems simple enough, this basis alone is too weak for building larger applications: Since the application's functionality tends to be scattered throughout event-listeners, one will almost certainly end up with a code base that cannot be maintained, extended, and ported to different platforms—in other words, software that must be thrown away and redeveloped from scratch.

This chapter investigates the architectural building block that keeps applications with user interfaces maintainable and portable: In the code, one always separates the application's business logic strictly from its graphical interface. Section 9.1 introduces this approach, called model-view separation, and traces it through different examples within the Eclipse platform. Next, Section 9.2 discusses its conceptual and technical basis, the classical MODEL-VIEW-CONTROLLER pattern. Section 9.3 introduces the JFace framework, which complements the basic SWT widgets by connecting them to the application's data structures. Section 9.4 uses a running example *MiniXcel*, a minimal spreadsheet implementation, to give a self-contained overview and to explore several implementation details of model-view separation that must be mastered to create truly professional applications. Finally, Section 9.5 adds the aspect of making edits undoable, which is indispensable for achieving usability.

Throughout the presentation, we will pay particular attention to the fact that model-view separation is deceptively simple: While the concept itself is rather straightforward, its rendering in concrete code involves many pitfalls. We will discuss particularly those aspects that have often been treated incorrectly in the work of novices to the field.

Before we start to delve into these depths of software design and implementation, there is one general piece of advice to set them into perspective:

Always gear the application toward the end users' requirements.

The reason for placing this point so prominently is that it is neglected so
📖258 often. As developers, we often get swept away by our enthusiasm for the
technically possible and the elegance of our own solutions. However, soft-
ware development is not a modern form of *l'art pour l'art*, but a means of
solving other people's pressing problems. These people, called "users," do
📖229 not care about the software's internals; they care about their own workflows.
So before you even start to think about the software's view and model and
📖28 the elegance of their separation, talk to the end users: What are their expec-
tations of the software's concrete behavior? How do they wish to interact
with the software? Which particular tasks must the software support? The
conscientious professional software engineer starts application development
by learning about the users' work—in other words, by learning about the
software's application domain. Everything said subquently must be subject
to this overall guideline.

9.1 The Core: Model-View Separation

Every application has a purpose for which it is built and which provides its unique value to its users. Correspondingly, the application contains code that implements the *business logic* to fulfill that purpose. Apart from that, most applications need a graphical user interface, simply because they have nontechnical users who do not appreciate command-line tools too much.

Apart from all of the strategic considerations related to software quality
»9.2.2 and maintenance, to be discussed later, it is useful to keep the code imple-
menting the business logic and the user interface separate simply because they have different characteristics (Fig. 9.1). Users buy, for instance, CAD software because its business logic can do CAD and nifty computations, but they accept it into their working routine because they like the way they can interact with it. The business logic of a CAD system must be extremely reliable to prevent bridges from collapsing, and it must be stable enough through different software releases, for instance, to read the same files correctly throughout projects running for several years. The interface, in contrast, must be visually appealing and must adapt to the changing working habits of its users so that they can, for instance, exploit new input methods such as 3D interaction devices. To achieve stability, the business
logic must adhere to rigorous contracts and must be tested comprehensively,
↰7.11 ↰5.3.5 while the interface is event-based and cannot be tested easily, especially if
it is liable to frequent changes. Finally, the business logic must deal with internal data structures and basic services such as file I/O, which are easily ported to different platforms. The API of graphical interfaces, in contrast, varies dramatically between platforms, and user interface code is usually not portable at all—for instance, from SWT to Swing. Keeping business logic and user interface separate is therefore first of all a matter of separation of concerns.

Business Logic	**User Interface**
Why users buy the application	Why users accept the application
The trusted, reliable, valuable core	Visually appealing front-end
Stable over a long time (e.g., file formats)	Volatile to adapt to changing user expectations [229]
Dominated by software-intrinsic concerns	Dominated by usability
Governed by contracts and service providers (Section 4.1)	Governed by events and reactions (Section 7.1, Section 7.11)
Demanding style and non-redundancy (Section 4.5)	Defensive programming (Section 4.6)
Comprehensive unit testing (Section 5.1)	Interface testing (Section 5.3.5)
Largely independent of operating system	Depends closely on window system (Section 7.1)

Figure 9.1 Characteristics of Business Logic and the User Interface

Keep the user interface and the business logic in different modules.

Accepting the goal of this separation, we have to investigate how it can be accomplished in the concrete software. Fig. 9.2 gives an overview, whose aspects we will explore in the remainder of this section. As a first step, one places the user interface and the business logic into separate modules, as indicated by the dashed horizontal dividing line in the figure. Referring to their roles in the MODEL-VIEW-CONTROLLER pattern, the business logic and the user interface are also called the *model* and the *view*, respectively, which explains the term *model-view separation* as a summary of the principle. »9.2

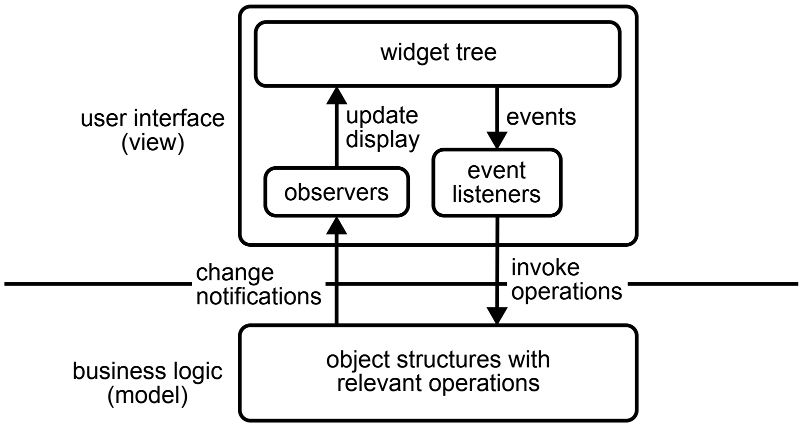


Figure 9.2 Overview of Model-View Separation

»A.1 In Eclipse, modules are implemented as plugins. Throughout the Eclipse code base, plugins with suffix `.ui` access the functionality provided by the corresponding plugins without that suffix. For instance, `org.eclipse.jdt.ui` accesses the Java Development Tools, whose logic comes in plugin `org.eclipse.jdt.core`, as well as `org.eclipse.jdt.launching`, `org.eclipse.debug.core`, and others.

Introducing separate plugins will at first appear as a somewhat large overhead for small applications. However, the sophisticated support for plugin development in Eclipse removes any technical complexity and exhibits the benefits of the split: The functionality can be linked into different applications to enable reuse; unit tests run much faster on plugins that do not require the user interface to come up; the OSGi class loader ensures that the logic code cannot inadvertently access interface classes; the logic module remains small and focused on its task; and several more. And, finally, successful small applications have a tendency to grow quickly into successful large applications; the split into different plugins ensures that they will also grow gracefully. »A.1.2 »A.1

The model contains the application's core functionality.

From the users' perspective, an application is all about the user interface, since they are not and should not be aware of any other part. The interface creates simplifications and abstractions that keep all the technical complexity under the hood. [229] When writing a letter with a word processor, for example, one certainly does not want to think about linear optimization problems for line and page breaking. [142]

The software engineer, in contrast, focuses on the business logic, or the model, in Fig. 9.2. That component contains the data structures and algorithms that solve the problems that the application is built for. Its objects constitute the machinery that the whole project relies on. »11.1 Its answers to the technical, conceptual, and maybe scientific challenges make up the team's and the company's competitive advantage. The user interface from this perspective is merely a thin, albeit commercially all-important, wrapper that enables nontechnical users to take full advantage of the functionality.

We have chosen the term "core functionality" rather than just "functionality" in this summary because the user interface does provide its own nontrivial behavior. Visual highlights and effects, reactions to drag-and-drop gestures, »9.4.4 and wizards to guide the user—they all require careful engineering in themselves. Yet, they do not belong to the "core," because they would need to be rebuilt from scratch on a new platform.

Never mention user interface classes in the logic.

The goal of the proposed division is to keep the business logic independent of the user interface, because this will establish precisely the separation of concerns indicated in Fig. 9.2. This can, however, be accomplished only if the code implementing the business logic never mentions user interface classes, such as widgets, images, or other resources: A single reference to a specific user interface library destroys portability and testability. At the level of modules, this means that the user interface module will reference the logic module, but not the reverse.

Connect the user interface to the logic using OBSERVER.

The question is then how logic objects can ever communicate with interface objects at all. The key insight here is that the OBSERVER pattern enables precisely this communication: The subject in the pattern accesses its observers ↰2.1.2 only through an interface that is defined from the perspective of the subject and is independent of the concrete observers.

In the case of model-view separation, the observer interface is contained in the business logic module, and that module sends change messages to observers in the interface module (see Fig. 9.2). These observers will translate the generic change notifications into concrete updates of the widgets.

Let us look at the example of Eclipse's management of background jobs, which also exhibits several interesting facets beyond the bare fundamentals. ↰2.1.1 We have already seen that the platform's `JobManager` allows observers to register for change notifications:

org.eclipse.core.internal.jobs.JobManager

```
public void addJobChangeListener(IJobChangeListener listener)
public void removeJobChangeListener(IJobChangeListener listener)
```

The interface `IJobChangeListener` is contained in the same package as the job manager itself, in `org.eclipse.core.runtime.jobs`. Neither that interface nor the `IJobChangeEvent` is connected in any way to possible user interfaces.

org.eclipse.core.runtime.jobs.IJobChangeListener

```
public interface IJobChangeListener {
    public void scheduled(IJobChangeEvent event);
    public void aboutToRun(IJobChangeEvent event);
    public void running(IJobChangeEvent event);
    public void done(IJobChangeEvent event);
     ...
}
```

↰2.1.2 The discussion of the OBSERVER pattern has pointed out that the definition of the observer interface must be independent of specific intended observers. It should focus instead on the possible changes occurring in the subject. This guideline becomes even more important in the case of model-view separation, because here the express intention is to keep the view exchangeable. Unfortunately, it is often tempting to reduce the complexity of the user interface code by sending along detailed notifications that meet »9.4.3 the interface's needs precisely, especially to obtain efficient incremental screen updates. In the long run, the simplicity of the current implementation will have to be paid for during subsequent changes and extensions of the user interface.

The standard user interface for jobs is the *Progress* view, implemented in class `ProgressView` and several helpers. They reside in the user interface package `org.eclipse.ui.internal.progress`. The central class is the ↰1.3.8 (singleton) `ProgressManager`, which registers to observe the (singleton) `JobManager`.

org.eclipse.ui.internal.progress.ProgressManager.JobMonitor

```
ProgressManager() {
     ...
    Job.getJobManager().addJobChangeListener(this.changeListener);
}
```

org.eclipse.ui.internal.progress.ProgressManager

```
private void shutdown() {
     ...
    Job.getJobManager().removeJobChangeListener(
        this.changeListener);
}
```

Construct view-related information at the view level.

The example of the *Progress* view also illustrates a typical aspect that accounts for a lot of the complexity involved in presenting the business logic adequately to the user: the need to create intermediate view-related data structures.

The model of jobs is essentially a flat list, where each job provides progress reports through progress monitors. Usability, however, is improved by arranging the display into a tree of running jobs, job groups, tasks, and subtasks that integrates all available information. The `ProgressManager` in the user interface therefore constructs a tree of `JobTreeElement` objects. Since the information is useful only for a specific intended user interface and might change when the users' preferences change, the maintenance of the tree is handled entirely in the view, not in the model. ↰7.10.2

⇄? This is actually a design decision. From a different perspective, the model itself might be structured. For instance, the JVM's bare `Thread`s naturally form a tree.

The `ProgressManager`'s internal logic then integrates two sources of information into a single consistent tree: the running and finished jobs, obtained through the observer registered in the preceding example, and the progress reports sent by the running jobs, to be discussed next.

Let the model access the view only through interfaces.

The observer pattern is only one instance of a more general principle, if we perceive the view and the model as different *layers* of the overall application. In this context, a lower layer accesses a higher layer only through interfaces defined in the lower layer, so as to allow higher layers to be exchanged later on. Furthermore, the calls to higher layers usually take the form of event notifications (see Fig. 9.2). In a typical example, the operating system's networking component does not assume anything about applications waiting for data, but it will notify them about newly arrived data by passing that data into the buffers belonging to the application's sockets. »12.2.2 📖59

Both aspects—the access through interfaces and the notifications—can also be seen in the handling of progress reports. The model-level `Job`s receive an object to be called back for the reports, but this object is given as an interface `IProgressMonitor`:

org.eclipse.core.runtime.jobs.Job

```
protected abstract IStatus run(IProgressMonitor monitor);
```

The user interface can then create a suitable object to receive the callbacks. In Eclipse, this is also done in the `ProgressManager` class, where `progressFor()` creates a view-level `JobMonitor`.

org.eclipse.ui.internal.progress.ProgressManager

```
public IProgressMonitor createMonitor(Job job,
                                      IProgressMonitor group,
                                      int ticks) {
    JobMonitor monitor = progressFor(job);
     ...  handle grouping of jobs
    return monitor;
}
```

The guideline of accessing the user interface only through interfaces can also be seen as a positive rendering of the earlier strict rule that no class from the user interface must ever occur in the model code. If the model code must collaborate with a view object, it must do so through model-level interfaces implemented by view objects.

Event-listeners mainly invoke operations defined in the model.

We have now discussed in detail the notifications sent from the model layer to the view layer, depicted on the left-hand side of Fig. 9.2. This focus is justified by the fact that the decoupling between model and view originates from the proper use of interfaces at this point. »12.1

The right-hand side of Fig. 9.2 shows the complementary collaboration between view and model. By technical necessity, the user input is always delivered to the application code in the form of events. The question then arises as to how the expected behavior of the overall application should be divided between the event-listeners in the view and the code in the model component. ↰7.1

The main insight is that the event-listeners are a particularly bad place for valuable code. The code cannot be tested easily, which makes it hard to get it stable in the first place, let alone keep it stable under necessary changes. Also, the code will probably be lost entirely when the users demand a different interface or the application is ported to a different platform (Fig. 9.1). ↰5.3.5 ↰5.4.8

It is therefore a good idea to place as little code and logic as possible into the event-listeners, and to move as much as possible into the model instead. There, it can be made reliable through contracts and testing; there, it can be reused on different operation systems; there, it can be maintained independently of the vagaries of user interface development. ↰4.1 ↰5.1

In the end, the ideal event-listener invokes only a few methods on the model. The only logic that necessarily remains in the event-listeners relates to the interface-level functionality such as the handling of drag-and-drop of data and of visual feedback on the current editing gestures. »9.4.4

In practice, one often starts adding functionality to meet concrete user demands, and one usually starts at the interface. The user says, "I need a button right here to do this particular thing," and the developer starts developing right with the event-listener. Such event-listeners tend to become long and complex, and it is useful to refactor them in retrospect. First, try to factor code fragments that are independent of the user interface into separate methods within the listener, then move those methods into the model. There, they will also be available to other team members for reuse. ↰1.2.2 ↰1.4.5

Design the model first.

It is tempting to start a new project with the user interface: You make rapid progress due to the WindowBuilder, you get early encouragement from prospective users, and you can show off to your team leader. All of this is important, since nifty data structures without a usable interface are not worth much—in the end, the users have to accept the application and use it confidently. For this reason, it can also be strategically sensible to start with the interface and even a mock-up of the interface, to check whether anybody will buy the finished product. ↰7.2

Because starting with the user interface is such an obvious choice, we wish to advocate the complementary approach: to start with the model. Here are a few reasons for postponing work on the user interface for a little while. 📖59

- You stand a better chance that the model will be portable and reusable. As with the test-first principle, the missing concrete collaborators in the user interface reduce the danger of defining the model, and in particular the observer interfaces (Fig. 9.2), specifically for those collaborators. ↰5.2 ↰2.1.2
- Test-first is applicable to the model, and it will have its usual benefits. ↰5.2
- The model will naturally contain all required functionality, so that the danger of placing too much functionality into the listeners is avoided from the start.
- There is no danger that a mock-up user interface presumes an API for the model that cannot be supported efficiently.
- The mission-critical challenges, such as in algorithmics, will be encountered and can be explored before an expensive investment in the user interface has taken place. If it turns out that the application will take a longer time than expected or cannot be built at all, the company has lost less money. Also, there is still time to hire experts to overcome the problems before the release.
- The user interface can focus on usability. Once the functionality is available, the user interface team just has to provide the most effective access paths to that functionality; it does not have to delve into the business logic aspects.

»9.2.2 Together, these aspects maximize the benefits of model-view separation.

Envision the interface while creating the model.

Conversely, a strict focus on the model is likely to have drawbacks for the final product. From an engineering point of view, the API of the model may not suit the demands of the interface, so that workarounds have to be found:

- The event-listeners contain extensive logic to access the existing API. This means that this logic will be lost when the interface has to change.
- ↰2.4.1 The model contains adapters to provide the expected API.
- The model has to be refactored.

From a usability perspective, the fixed model API may induce developers to take the easy way out of these overheads and to provide a user interface that merely mirrors the internals. A typical example comprises CRUD (CReate Update Delete) interfaces to databases, which are easy to obtain, but which are known to provide insufficient support for the user's workflows. 📖220,145,266 📖114

Model-view separation incurs an extra complexity that will pay off.

We have seen much motivation and many benefits of model-view separation, and we will discuss the details. »9.2.2 At the end of this overview, however, let us consider not the benefits, but the costs of model-view separation.

- Splitting the code into separate modules always involves the design of interfaces between the modules, and the communication about them can take a lot of time and presents the potential for mistakes that must be remedied later at high cost. When a data structure is kept right in the user interface, one can hack in a new requirement at the last minute. In contrast, if the data is encapsulated in a different module, one may have to negotiate with the developers who are responsible first.
- The collaboration from model to view always takes place by generic change notifications (Fig. 9.2), rather than specific method calls that update parts of the screen. In the model, one has to provide the general OBSERVER pattern for many objects, even if there is in the end only a single concrete observer in the user interface. Furthermore, the logic to translate the changes into screen updates itself can be substantial and complex, especially if it is necessary to repaint the smallest possible screen area to keep the application responsive. ↰2.1 ↰2.1.4 »9.4.3

Model-view separation is therefore an effort that must be taken at the start of a project. »9.4 The walk-through example of MiniXcel will give you a mental checklist of the single steps, which allows you to assess the overall effort up front. We hope that the checklist is then simple enough to convince you of using model-view separation in all but the most trivial throwaway applications. Even in projects of a few thousand lines, the investment in the extra structure will pay off quickly, since the software becomes more testable, maintainable, and changeable. And if the application happens to live longer than expected, as is usually the case for useful software, it is ready for that next step as well.

9.2 The Model-View-Controller Pattern

The Model-View-Controller pattern (MVC) has proven a tremendous success in many different areas of user interfaces, starting from the original SmallTalk toolkit, through all major players such as Qt, GTK, SWT, 📖146
Swing, and MFC, right to web application frameworks such as Ruby on Rails and ASP.MVC. Naturally, the different areas have produced different variants that suit their specific needs. Nevertheless, the fundamental concept remains the same. We will study here the classical version, which will 📖59
also clarify the workings of the variants. We will use a minimal example to illustrate the conceptual details of the pattern clearly without swamping the discussion with unnecessary technical complications. A more extended example will be given in the MiniXcel application. Also, we start out with »9.4
the classical separation of view and controller, even if most practical implementations unify these roles. Understanding the separate responsibilities of »9.2.8
view and controller separately first will later help to create clearer structures.

9.2.1 The Basic Pattern

The structure of the MVC pattern is shown in Fig. 9.3. In essence, the pattern reflects the model-view separation: The business logic is kept separate ↰9.1
from the user interface code, and the logic collaborates with the interface only through generic change notifications in the Observer. The pattern ↰2.1
adds a finer subdivision in the interface layer: The *view* is responsible for rendering the application data on the screen, while the *controller* contains the logic for reacting to user input.

The benefit of this additional split is mainly a stricter separation of concerns. We have seen in the discussion of the Mediator that the event- ↰7.7
listeners attached to widgets can quickly become complex in themselves. Moving this code into a self-contained object will keep the code of the view more focused on the visual presentation itself. Although many practical implementations reunite the two roles in the Document-View variant, »9.2.8
is useful to consider them separately first, since this will lead to a clearer structure within the view component of this later development.

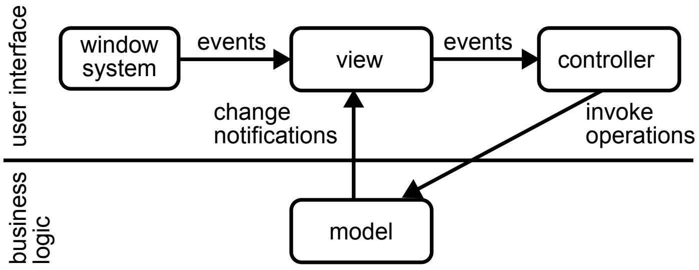


Figure 9.3 The Basic Model-View-Controller Pattern

In summary, the three roles of the pattern then perform these tasks:

- ↰9.1 ↰7.11 The model maintains the application's data structures and algorithms, which constitute its business logic. The model is the valuable and stable core of the product; it is built to last through revisions and ports to different window systems. It builds on precise contracts and is thoroughly unit-tested.
- ↰7.1 The view renders the current state of the application data onto the screen. It accesses the model to retrieve the data, and registers as an observer to be notified about any changes and to keep the display up-to-date. By technical necessity, it also receives all user input as events and passes those events on to the controller.
- ↰1.8.1 The controller interprets the user input events as triggers to perform operations and modifications on the model. It contains the logic for handling the events. In this role, it is a typical decision maker: It decides what needs to be done, but delegates the actual execution to others. In the basic pattern, this means calling the model's methods.

The pattern describes all three roles as if they were filled by single objects. However, this is hardly ever the case: The application logic is usually implemented in a complex component with many helper objects that collaborate intensively, and even the view may need helpers to fulfill its task.

↰7.5 To see the pattern in action, we implement a tiny widget that enables a single integer value to be incremented and decremented by clicking on different areas (Fig. 9.4). Rather than building a compound widget, we implement this from scratch to show all of the details.

The model maintains the application data and supports observers.

↰9.1 Following the earlier advice, we start with the model. Its "functionality" is to maintain a single integer value. To serve as a model in the pattern, the object also implements the OBSERVER pattern. The crucial point to be noted is that the model is in no way adapted to the intended presentation on

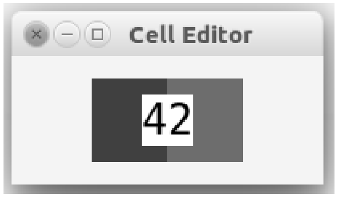


Figure 9.4 Minimal MVC Example

the screen. In particular, the observers are merely notified that the content has changed (line 21); there is no indication that this notification will trigger a screen update later on.

celledit.mvc.IntCell

```
1  public class IntCell {
2    private int content;
3    private EventListenerList listeners = new EventListenerList();
4    public void addCellListener(CellListener l) {
5         ...
6    }
7    public void removeCellListener(CellListener l) {
8         ...
9    }
10   public int get() {
11     return content;
12   }
13   public void set(int cnt) {
14     int old = content;
15     this.content = cnt;
16     fireCellChanged(old);
17   }
18   protected void fireCellChanged(int old) {
19     for (CellListener l : listeners.getListeners(
20                                          CellListener.class))
21       l.cellChanged(this, old, content);
22   }
23 }
```

The view displays the data on the screen.

The view in the pattern must paint on the screen, so it derives from `Canvas`. ↰7.8
It keeps references to the current model and controller, as well as the (larger) ↰7.4.1
font used for painting and the computed preferred size. ↰7.1

celledit.mvc.View

```
public class View extends Canvas {
    private IntCell model;
    private Controller controller;
    private Font fnt;
    private Point sizeCache;
     ...
}
```

↰7.8 The main task of the view is to render the application data on the screen. The following excerpt from the painting method gives the crucial point: Line 3 gets the current value from the model and transforms it into a string to be drawn on the screen in line 7. The remaining code serves to center the string in the widget (`bounds` is the area available for painting).

celledit.mvc.View

```
1 private void paintControl(PaintEvent e) {
2      ...  paint red and green fields
3     String text = Integer.toString(model.get());
4     Point sz = g.textExtent(text);
5     int x = bounds.width / 2 - sz.x / 2;
6     int y = bounds.height / 2 - sz.y / 2;
7     g.drawString(text, x, y);
8 }
9
```

The view keeps the display up-to-date by observing the model.

To keep the display up-to-date, the view must observe the model. Whenever the model changes, the view observes the new model.

celledit.mvc.View

```
public void setModel(IntCell c) {
    if (this.model != null)
        this.model.removeCellListener(modelListener);
    this.model = c;
    if (this.model != null)
        this.model.addCellListener(modelListener);
}
```

↰7.4.1 ⚠ Do not forget to detach the view from the model when the view is disposed. This can be achieved reliably by setting the model to `null` in a `DisposeListener`.

↰7.8 The `modelListener` merely requests a complete repainting of the widget. In many scenarios, this is too inefficient for production use, so that »9.4.3 incremental repainting must be implemented. To demonstrate the pattern, the simple choice is sufficient.

celledit.mvc.View

```
private CellListener modelListener = new CellListener() {
    public void cellChanged(IntCell cell, int oldVal, int newVal) {
        redraw();
    }
};
```

The view forwards user input to the controller.

Finally, the view must forward the events to the controller. This is usually achieved by registering the controller as an event-listener. For the current example, we delegate the actual registration to the controller itself to demonstrate an exchange of the controller later on. »9.2.7

celledit.mvc.View.setController

```
public void setController(Controller c) {
    if (controller != null)
        controller.detach(this);
    controller = c;
    if (controller != null)
        controller.attach(this);
}
```

Having finished with the model and the view, we have set up the main axis of Fig. 9.3: The display on the screen will always reflect the current data, independent of how that data will be manipulated. We will now add this last aspect by implementing the controller.

The controller receives all relevant user input.

The controller must receive all user input relevant to the expected reactions. Since the input is technically sent to the view, the controller registers itself as a listener on the view. In the current example, it becomes a mouse-listener to receive the clicks that will trigger the increment and decrement operations. (The `super` call merely remembers the view in a field `view`.)

celledit.mvc.MouseController.attach

```
public void attach(View view) {
    super.attach(view);
    this.view.addMouseListener(this);
    this.view.addMouseTrackListener(this);
    this.view.addMouseMoveListener(this);
}
```

The controller interprets the events as operations on the model.

The summary of tasks given earlier states that the purpose of the controller is to translate raw input events into operations on the model. The implementation can be seen in the callback methods for mouse clicks. The controller accesses the model to be operated on (lines 2–3) and checks which area the click actually occurred in (lines 4 and 7). Based on this information, it decides whether the model value should be incremented or decremented (lines 6 and 8). As a detail, the controller decides not to decrement the value if it has already reached 0. »9.2.3

celledit.mvc.MouseController.mouseUp

```
public void mouseUp(MouseEvent e) {
    if (view.getModel() != null) {
        IntCell m = view.getModel();
        if (view.isInDecrementArea(new Point(e.x, e.y)) &&
            m.get() > 0)
            m.set(m.get() - 1);
        else if (view.isInIncrementArea(new Point(e.x, e.y)))
            m.set(m.get() + 1);
    }
}
```

The pattern processes input through view, controller, and model.

The overall goal of the Model-View-Controller pattern can also be seen by tracing the user input through the different roles, until an actual screen update occurs.

1. The view receives the input and hands it to the controller.
2. The controller decides which action to take on the model.
3. The model performs the invoked operation and sends the resulting changes to the view, as one of possibly several observers.
4. »9.4.3 The view interprets the model changes and decides which parts of the screen need to be redrawn.
5. The view refetches the relevant data and paints it on the screen.

This sequence of steps highlights the contributions of the different objects. It also points out that each of them can influence the final outcome: The view will contribute the visual appearance; the controller implements the reaction, since the view simply forwards events; and the model implements the functionality, but does not see the input events.

The central point of model-view separation is seen in steps 2 and 3. First, the controller alone is responsible for interpreting the input events; the model is not aware of the real causes of the invoked operations. Second, the model is not aware of the precise view class, or that there is a user interface at all; it merely supports the Observer pattern.

9.2.2 Benefits of the Model-View-Controller Pattern

The Model-View-Controller pattern is, in fact, rather complex and requires some extra implementation effort, compared to the naive solution of implementing the application's functionality directly in event-listeners attached to the widgets. The investment into the extra structure and indirections introduced by the pattern must therefore be justified.

The user interface remains flexible.

The most important benefit of the pattern derives from its ability to keep the user interface flexible. Because the application's functionality stays safe and sound in the model component and does not depend on the user interface in any way, it will remain valid if the interface changes. This can and will happen surprisingly often over the software's lifetime.

The first reason for changing the user interface is the user. The central goal of a user interface is to support the users' workflows effectively. As these workflows change or the users develop new preferences, the interface should ideally be adapted to match them. Also, different user groups may have different requirements, and new views may need to be developed as these requirements emerge. The MVC pattern confines such changes to the actual interface, unless the new workflows also require new computations and operations. 📖229

The second reason for changes relates to the underlying window system. When APIs change or new widgets or interaction devices are developed, the user interface must exploit them for the users' benefit. Since these aspects are usually not related to the functionality in any way, the MVC keeps the application's core stable.

Finally, it may be desirable to port the application to an entirely different platform. Here, the problem lies mostly in the user interface. In the best case, an analogous set of widgets will be available: Whether you access Windows, MacOS, GTK, or Qt, their widgets offer basically very similar services and events. Nevertheless, the user interface must usually be redeveloped from scratch. The MVC pattern ensures that the valuable core of the application, its functionality, will continue to work in the new environment, since this core uses only standard services such as file or network access, for which cross-platform APIs are available or where the platform differences can be hidden behind simple adapters. ↰2.4.1

Multiple, synchronized views can better support the users' workflows.

Modern IDEs such as Eclipse give us a good grasp on our source code. For example, while we work on the source in a text editor, we see an outline of its structure on the side. When we rename a method in one of the two windows, the other window reflects the change immediately. The reason is simply that both windows are, possibly through an indirection of the Java Model, views for the same text document, which fulfills the role of the view component in the MVC pattern. Similarly, Eclipse's compiler reports an error only once by attaching an `IMarker` object to the file. The marker is reflected in the editor, the problems view, and as a small icon in the package explorer and project navigator.

The Model-View-Controller pattern enables such synchronized views on the application's data structure because views observe the model and are informed about its current state regardless of why changes have occurred.

The display remains up-to-date with the internal state.

At a somewhat more basic level, users will trust an application only if they are never surprised by its behavior. One common source of surprises is inconsistency between the internal data structures and the displayed data. The MVC pattern eliminates this chance completely and ensures that the users always base their actions and decisions on the most up-to-date information about the internal structures.

The application's functionality remains testable.

↰5.4 The single most important technique for making a system reliable and keeping it stable under change is testing. By making the functional core, the model, independent of a user interface, its operations can also be exercised in a testing fixture (see Fig. 5.1 on page 246) and its resulting state can be examined by simple assertions in unit tests. Testing the user interface, ↰5.3.5 in contrast, is much more complex. Since the user interface itself tends to change very often, the effort of adapting the existing test cases and creating new ones will be considerable. The functional core, in contrast, is built to remain stable, so that the investment of testing will pay off easily.

Model-view separation enables protection of the system's core.

The stability of an application's functionality relies heavily on precise con- ↰4.1 tracts. Within this reasoning framework, each method trusts its callers to fulfill the stated pre-condition—that is, to pass only legal arguments and ↰4.5 to call the method only in legal object states. The non-redundancy principle condenses the idea of trust into the development practice of never ↰4.6 ↰1.5.2 checking pre-conditions. At the system boundary, in contrast, the code can never trust the incoming data and requests. Methods must be written to be robust, and to check whether they really do apply.

Model-view separation offers the benefits of localizing these necessary checks in the user interface component and maintaining the functional core in the clean and lean style enabled by the non-redundancy principle.

9.2.3 Crucial Design and Implementation Constraints

↰2.1.2 As with the Observer pattern, the concrete implementation of the Model-View-Controller pattern must observe a few constraints to obtain the ↰9.2.2 expected benefits. We list here those aspects that we have found in teaching to make the difference between the code of novices and that of professionals.

Do not tailor the OBSERVER pattern to a specific view.

The first aspect is the definition of the *Observer* interface for the model. Especially when dealing with complex models and the necessity of incremental screen updates, there is always the temptation to "tweak" the change notifications a bit to simplify the logic that determines which parts of the screen need to be updated. Certainly, one should use the "push" variant of the OBSERVER pattern; that is, the change notifications should be very detailed to enable any view to work efficiently regardless of its possible complexity. »9.4.3 ↰2.1.3

When targeting the messages at specific views, however, one endangers the ability to add a new view or to change the existing one, or to port the application to an entirely different platform. Suppose, for instance, that the model manages a list of objects with some properties. It should then send a change message containing a description of the change. However, it should not use a message `updateTableRow()` simply because the current view is a `Table` widget. A better choice is a message `changedData()`, which reflects the change instead of the expected reaction. If the view displays the properties in a specific order, the model must not send messages `updateTable(int row, int col)`, but rather `changedData(DataObject obj, String property)`. Even if this means that the view must map objects to rows and the property names to column indices, it increases the likelihood that the view can change independently of the model.

The controller never notifies the view about triggered operations.

A second shortcut that one may be tempted to take is to let the controller notify the view directly about any changes it has performed on the model, rather than going through the indirection via the model. First, this shortcut is marginally more efficient at runtime. What is particularly attractive, however, is that it saves the implementation of the general OBSERVER pattern in the model and the perhaps complex logic for translating changes to screen updates in the view. ↰2.1.4

However, the shortcut really destroys the core of the pattern, and nearly all of its benefits. One can no longer have multiple synchronized views. Also, the information on the screen may no longer be up-to-date if the controller neglects internal side effects and dependencies of the model. Finally, the logic for the updates must be duplicated in ports and variations of the user interface. »9.4.2

The controller delegates decisions about the visual appearance to the view.

A comparatively minor point concerns the relationship between the view and the controller. If these roles are implemented as different objects at any point, then one should also strive for a strict separation of concerns—for instance, to keep the controller exchangeable. »9.2.8 »9.2.7

One notable aspect is the possible assumptions about the visual appearance. The controller often receives events that relate back to that visual appearance. For instance, a mouse click happens at a particular point on the screen, and the visual element at this point must determine the correct reaction. If the controller makes any assumptions about this visual element, it is tied to the specific implementation of the view. If several controllers exist, then it becomes virtually impossible to change even simple things such as the font size and spacing, since several controllers would have to change as well. »12.1.2

↰9.2.1 In the following tiny example, we have therefore made the controller ask the model whether the click event `e` occurred in one of the designated "active" areas. The controller now assumes the existence of these areas, but it does not know anything about their location and shape. That knowledge is encapsulated in the view and can be adapted at any time.

celledit.mvc.MouseController.mouseUp

```
if (view.isInDecrementArea(new Point(e.x, e.y)) && m.get() > 0)
    m.set(m.get() - 1);
else if (view.isInIncrementArea(new Point(e.x, e.y)))
    m.set(m.get() + 1);
```

»9.2.8 Even in the common Document-View variant of the MVC, where view and controller are implemented together in one object, it is still useful to obey the guideline by separating the concerns into different methods of the object.

The controller shields the model from the user input.

↰1.5.2 ↰4.6 The user interface is, of course, one of the system's boundaries. Accordingly, all user input must be treated with suspicion: Has the user really entered valid data? Has the user clicked a button only when it makes sense? Does the selected file have the expected format?

Many of these questions are best handled in the controller, because it is the controller that receives the user input and decides which model operations need to be called in response. Since the model is built according to the principles of design by contract, it does not check any stated preconditions. It is the controller's task to ensure that only valid method calls are made. ↰7.11 ↰4.5

9.2.4 Common Misconceptions

The Model-View-Controller pattern is rather complex, so it is not surprising that a few misunderstandings arise when first thinking it through. We have found in teaching that some misunderstandings tend to crop up repeatedly. They seem to arise mostly from the correct impression that the MVC is all about exchangeability and flexibility. However, one has to be careful about what really is exchangeable in the end and must not »12.2

conclude that "all components can be exchanged and adapted to the users' requirements." We hope that highlighting the nonbenefits of the pattern in this section will enhance the understanding of the benefits that it does create.

Model-view separation is not a panacea.

The rather extensive mechanisms and logic necessary for establishing a proper model-view separation must always be seen as an investment. It is an investment that pays off quite quickly, even for medium-sized applications, but it is still an investment. The decision for or against using the MVC must therefore be based on a precise understanding of it benefits, so as to relate them to the application at hand. A small tool written for one project only will never need porting, for example, and if the developer is also its only user, there is little chance of having to change the user interface. A general understanding that the MVC offers "everything that can be wished for" is not enough.

The model is not exchangeable and the view is not reusable.

The view and the controller necessarily target a specific model: They ask the model for data and draw exactly that data; the view registers as an observer and expects certain kinds of change messages; and the controller translates user gestures into specific operations offered by the model. As a result, the model cannot usually be exchanged for a different one; by switch of perspective, this means that the view is usually not reusable.

Of course, it is still possible to implement generic widgets that access the model only through predefined interfaces. For instance, a table on the screen has rows, and the data in each row provides strings for each column. Both JFace and Swing provide excellent examples of generic and reusable tables. However, this is an exercise in library or framework design. To build a concrete user interface, one has to supply adapters that link the generic mechanisms to the specific application model, and one has to implement listeners for generic table events that target the specific available model operations. In this perspective, the generic table is only a building block, not the complete user interface in the sense of the MVC.

»9.3.1 80

↰2.4.1

The controller is usually neither exchangeable nor reusable.

The controller interprets user gestures, such as mouse moves, mouse clicks, and keyboard input. These gestures have a proper meaning, and hence a reliable translation to model operations, only with respect to the concrete visual appearance of the view. It is therefore usually not possible to reuse a controller on a different view. Exchanging the controller is possible, but only within the confines of the event sources offered by the view.

»9.2.7

9.2.5 Behavior at the User Interface Level

Effective user interfaces allow the user to invoke common operations by small gestures. For example, moving a rectangle in a drawing tool takes a mouse click to select the rectangle and a drag gesture to move it. Since many similarly small gestures have similarly small but quite different effects, the application must provide feedback so that the user can anticipate the reaction. For instance, when selecting a rectangle, it acquires drag handles—that is, a visual frame that indicates moving and resizing gestures will now influence this object.

Implement user feedback without participation of the model.

The important point to realize is that feedback is solely a user interface behavior: Different platforms offer different mechanisms, and different users will expect different behavior. The model does not get involved until the user has actually triggered an operation.

↰9.2.1 Suppose, for instance, that we wish to enhance the example widget with the feedback shown in Fig. 9.5. When the mouse cursor is *inside* the widget, a frame appears to indicate this fact (a versus b and c); furthermore, a slightly lighter hue indicates whether a click would increment or decrement the counter (b versus c), and which field is the current *target* of the click.

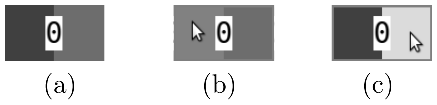


Figure 9.5 User-Interface Behavior: Mouse Feedback

Feedback is triggered by the controller.

The second aspect of feedback concerns the question of which role will actually decide which feedback needs to be shown. The answer here is clear: Because the controller will finally decide which operation is triggered on the model, it must also decide which feedback must be shown to apprise the user of this later behavior. It is similarly clear that the controller will decide on the feedback but will delegate the actual display to the view.

In the implementation of the example, the `Controller` tracks both the general mouse movements into and out of the widget, and the detailed movements inside the widget. The reaction to the `mouseEnter` and `mouseExit` events is straightforward: Just tell the view to draw the frame or to remove it. When the mouse leaves the widget, any target highlight must, of course, also be removed. The `mouseMove` proceeds in parallel to the `mouseUp` method in the basic implementation: It checks which operation it would perform and sets the corresponding highlight. ↰9.2.1

celledit.mvc.MouseController

```
public void mouseEnter(MouseEvent e) {
    view.setInside(true);
}
public void mouseExit(MouseEvent e) {
    view.setInside(false);
    view.setTargetField(View.TARGET_NONE);
}
public void mouseMove(MouseEvent e) {
    if (view.isInDecrementArea(new Point(e.x, e.y)))
        view.setTargetField(View.TARGET_DECREMENT);
    else if (view.isInIncrementArea(new Point(e.x, e.y)))
        view.setTargetField(View.TARGET_INCREMENT);
    else
        view.setTargetField(View.TARGET_NONE);
}
```

The naming of the `View` methods is worth mentioning. They publish the fact that some visual effect can be achieved, but the effect itself remains a private decision of the `View`. This parallels the earlier implementation of `mouseUp`, where the controller did not know the exact shape of the clickable areas within the view.

We said earlier that `mouseExit` must "of course" remove any target highlight. The question is whether this must be as explicit as in the code shown here: Would it not be better if the call `setInside(false)` would also remove the target highlight? In other words, shouldn't the connection between the feedback mechanisms already be established within the `View` class? It would certainly make the controller's methods simpler and more symmetric, and it would ensure a certain consistency within the view. We have chosen the variant in the example to emphasize that all decisions about feedback lie with the controller. In practical implementations, the other options can, however, be equally valid.

Feedback usually requires special state in the view.

In implementing the actual visual feedback within the `View`, we have to take into account one technical detail: Painting always occurs in a callback, at some arbitrary point that the window system deems suitable. The view must be ready to draw both the data and the feedback at that point. We therefore introduce special state components in the view: ↰7.8

celledit.mvc.View

```
private boolean inside = false;
public static final int TARGET_NONE = 0;
public static final int TARGET_DECREMENT = 1;
public static final int TARGET_INCREMENT = 2;
private int targetField = TARGET_NONE;
```

The `View` publishes the new state, but only to its related classes, such as the `Controller`. The setter for the state stores the new value and invokes `redraw()` to request a later painting operation. Since this is potentially expensive, one should always check whether the operation is necessary at all. ↰7.8

celledit.mvc.View

```
protected void setInside(boolean inside) {
    if (this.inside == inside)
        return;
    this.inside = inside;
    redraw();
}
```

The actual painting then merely checks the current feedback state at the right point and creates the visual appearance. Here is the example for highlighting the "decrement" field; the increment field and the "inside" indications are similar.

celledit.mvc.View

```
private void paintControl(PaintEvent e) {
     ...
    if (targetField == TARGET_DECREMENT)
        g.setBackground(getDisplay().getSystemColor(SWT.COLOR_RED));
    else
        g.setBackground(getDisplay().getSystemColor(
                                          SWT.COLOR_DARK_RED));
    g.fillRectangle(bounds.x, bounds.y, bounds.width / 2,
                    bounds.height);
     ...
}
```

Separate view-level state from the application functionality.

The example of the feedback given here has introduced the necessity of state that only lives at the view level but does not concern the application's core data structures. A plethora of similar examples comes to mind immediately: the selection in a text viewer or the selected row in a table; the folding and unfolding of nodes in a tree-structured display, such as SWT's `Tree`; the currently selected tool in an image editor; the position of scrollbars in a list and the first row shown in consequence; the availability of buttons depending on previous choices; and many more.

In the end, the view-level state and the model-level state must be merged in one consistent user interface with predictable behavior. Internally, however, the two worlds must be kept separate: The one part of the state is thrown away, and the other must be stable when the interface changes; the one part is best tested manually, and the other must be rigorously unit-tested. Consequently, one must decide for each aspect of the overall state to which of the worlds it will belong.

The decision may seem rather obvious at first, but some cases might merit deeper discussions and sometimes one may have second thoughts about a decision. For instance, the GIMP image editor treats the selection as part of the model: You can undo and redo selection steps, and the selection even gets saved to the `.xcf` files. The reason is, obviously, that in the image manipulation domain, selection is often a key operation, and several detailed selection steps must be carried out in sequence to achieve a desired result. Being able to undo and redo selection helps users to remedy mistakes in the process. »9.5

9.2.6 Controllers Observing the Model

In the basic MODEL-VIEW-CONTROLLER pattern, the view necessarily observes the model, because it must translate any changes in the data to updates of the display. In many scenarios, the controller will also observe the model. ↰9.2.1

Controllers can observe the model to indicate availability of operations.

A typical example of this behavior is seen in menu items that get grayed out if an operation is not available. For instance, a text editor will gray out the "copy" and "cut" entries if there is currently no selection.

The controller decides on the availability of operations.

It might be tempting to integrate the feedback on available actions directly into the view. After all, the view already observes the model and it can just as well handle one more aspect while it is at work anyway. However, since the controller decides which operations it will invoke for which user input, it is also the controller which decides whether these operations are currently available.

Suppose, for instance, that we wish to gray out the decrement field if the current count is already 0. This requires an extension of both the `View` and the `Controller` classes: The view acquires a new bit-mask stating which of the fields need to be grayed out, and that information is used when choosing the background color in `paintControl()`. The controller observes the model and switches the "gray" flags of the fields according to the current model value.

You might ask whether to bother graying out the "increment" field at all, since the widget's behavior does not assume an upper bound. We feel that keeping the implementation slightly more general and symmetric at very little cost at this point might help in future extensions. After all, similar widgets such as `Slider` and `ScrollBar` all do have upper limits.

Controllers must assume that others modify the model.

One possible pitfall that leads to nonprofessional code lies in the fact that the controller modifies the model itself and therefore seems to know precisely whether an operation causes some action to become unavailable. However, it should be noted that the MVC is built to support multiple synchronized views, and that other controllers may invoke model operations as well. Each controller that depends on the model's state must therefore observe the model.

9.2.7 Pluggable Controllers

»9.2.8
📖146

Even if, as we shall see shortly, the view and controller are often coupled so tightly that it is sensible to implement them in a single object, it is still instructive to consider briefly the concept of making the controller of a view pluggable to implement new interactions with an existing graphical presentation. This flexibility can be achieved only after understanding precisely the division of responsibilities between view and controller.

↰9.2.1

So, let us implement a controller that enables the user to access the number entry field from the introductory example (Fig. 9.4 on page 455) via the keyboard. The new `KeyboardController` waits for keyboard input and modifies the model accordingly. Since the view observes the model, the change will become visible to the user.

celledit.mvc.KeyboardController.keyReleased

```java
public void keyReleased(KeyEvent e) {
    IntCell m = view.getModel();
    switch (e.character) {
    case '+':
        m.set(m.get() + 1);
        break;
    case '-':
        if (m.get() > 0)
            m.set(m.get() - 1);
        break;
    }
     ...
}
```

Keyboard input is different from mouse input in that it is not the current location of some cursor, but the *keyboard focus* of the window system (and SWT) that determines which widget will receive the events. The keyboard focus is essentially a pointer to that target widget, but it has interactions with the window manager (because of modal dialogs) and the tab order of widgets in the window. It is therefore necessary to display feedback to the users so that they know which reaction to expect when they press a key. The new controller therefore registers as a `FocusListener` of the `View`.

↰7.6

celledit.mvc.KeyboardController.attach

```
public void attach(View view) {
     ...
    view.addFocusListener(this);
}
```

The controller then uses the existing "inside" indication on the view for the actual feedback:

celledit.mvc.KeyboardController

```
public void focusGained(FocusEvent e) {
    view.setInside(true);
}
public void focusLost(FocusEvent e) {
    view.setInside(false);
}
```

Another convention is that clicking on a widget with the mouse will give it the focus. This is, however, no more than a convention, and the widget itself has to request the focus when necessary. This reaction can be implemented directly. (Note that the actual indication that the focus has been obtained is shown indirectly, through the event-listener installed previously.)

celledit.mvc.KeyboardController.mouseUp

```
public void mouseUp(MouseEvent e) {
    view.setFocus();
}
```

Finally, it is also useful to give a visual indication, in the form of a short flash of the respective increment/decrement fields, when the user presses the "+" and the "–" keys. This, too, can be achieved with the existing feedback mechanisms. The `keyReleased()` event then resets the target field to "none." The flash will therefore mirror precisely the user's pressing of the respective key.

celledit.mvc.KeyboardController.keyPressed

```
public void keyPressed(KeyEvent e) {
    switch (e.character) {
    case '+':
        view.setTargetField(View.TARGET_INCREMENT);
        break;
    case '-':
        view.setTargetField(View.TARGET_DECREMENT);
        break;
    }
}
```

The new controller emphasizes the division of logic between the view and the controller: The display and highlights remain with the view, and the controller decides what needs to be done in reaction to incoming user input. It is this division that has enabled us to reuse the existing highlight mechanisms for new purposes.

»12.4 You might, of course, be suspicious of this reuse: Was it just coincidence that the existing mechanisms worked out for the new controller? Reuse always requires anticipating the shape of possible application scenarios and keeping the supported ones lean at the cost of excluding others. In the current case, we would argue that the feedback mechanisms that the view provides match the user's understanding of the widget: The user "activates" the widget by "zooming in," either by the mouse or by the keyboard focus, and then "triggers" one of the increment and decrement areas. All of these interactions are then mirrored by the highlights.

Nevertheless, it must be said that views and controllers usually depend heavily on each other, so that exchanging the controller is rarely possible. 214 One example where it is enabled is found in the pluggable *edit policies* of the Graphical Editing Framework, which create a setup where reusable controller-like logic can be attached to various elements of the user interface in a flexible way.

9.2.8 The Document-View Variant

The view and controller in the MVC pattern are usually connected very tightly: The controller can request only those events that the view provides, and it can make use of only those feedback mechanisms that the view implements. Since it is therefore often not possible to use either the view or the controller without the other, one can go ahead and implement both roles in the same object. This leads to the DOCUMENT-VIEW pattern, where the ↰9.1 document contains the application logic and the view contains the entire user interface code. In this way, the interface code can share knowledge about the widget's internals between the logic of the display and the event-listeners. This may facilitate coding and avoids having to design an API that enables the view and the controller classes to communicate.

↰9.2.1 Let us examine this idea through the simple example of incrementing and decrementing an integer value. We start from a technical perspective. ↰7.8 Since we need to implement a widget with custom painting, the overall structure is that of a `Canvas` with attached listeners. The drawing part is actually the same as in the previous implementation. Only the code for the event-listeners is integrated. In the simplest case, we wait for mouse ↰2.1.3 clicks. To avoid publishing this fact by making the `View` class implement `MouseListener`, we attach an anonymous listener that delegates to the outer class.

celledit.docview.View.View

```
addMouseListener(new MouseAdapter() {
    public void mouseUp(MouseEvent e) {
        handleMouseUp(e);
    }
});
```

Keep the code for display and reaction loosely coupled.

On the first try, one is liable to take the freedom of "sharing knowledge" between display and event-listeners very literally. For instance, we know that `paintComponent()` draws the dividing line between the decrement and increment fields right in the middle of the widget's screen space. The event-listener can therefore be written up like this:

celledit.docview.View

```
private void mouseUp1(MouseEvent e) {
    Rectangle area = getClientArea();
    if (cell.get() > 0 && area.width / 2 <= e.x &&
        e.x <= area.width &&
        0 <= e.y && e.y <= area.height)
        cell.set(cell.get() - 1);
    ...
};
```

However, this is highly undesirable: It is not possible to change the visual appearance without going through the entire class and checking which code might be influenced. It is much better to introduce a `private` helper method that decides whether a particular point is in the increment or decrement fields. Placing this helper near the `paintComponent()`—that is, splitting the class logically between display and reaction code—will greatly facilitate maintenance. ↰1.4.5

celledit.docview.View

```
private void handleMouseUp(MouseEvent e) {
    if (cell.get() > 0 && isInDecrementArea(new Point(e.x, e.y)))
        cell.set(cell.get() - 1);
    ...
};
private boolean isInDecrementArea(Point p) {
    ...
}
```

In the end, this implementation is very near the original division between view and controller. One crucial difference is that now the helper method is not an external API that may be accessed from the outside and must therefore be maintained, but rather a `private`, encapsulated detail that may be changed at any time without breaking other parts of the system.

With predefined widgets, access their API directly.

In many cases, the actual display consists of predefined widgets such as text fields or tables. These widgets already encapsulate all painting-related aspects so that it is not necessary to introduce helpers. The DOCUMENT-VIEW pattern then applies very directly, since listeners can get the content or the selection of widgets without further ado.

9.3 The JFace Layer

↰7.1 SWT is a typical user interface toolkit that provides the standard interaction elements, such as text fields, tables, and trees, out of the box. However, it is also designed to be minimal: Since it accesses the native widgets ↰7.4 of the platform that the application executes on, the SWT classes must be ported to every supported platform. For that reason, SWT offers only bare-bones functionality. Any higher-level functionality is factored out into the JFace framework, which is pure Java and portable. JFace facilitates connecting the application data structures to the existing SWT widgets, and is therefore indispensable for effective development of user interfaces. It also provides standard elements such as message dialogs and application windows equipped with a menu, toolbar, and status bar.

From a conceptual point of view, JFace provides a complementary perspective on model-view separation. While usually the model is stable and the user interface remains flexible, JFace provides fixed but generic user interface components that connect flexibly to application-specific models. Studying its mechanisms will enhance the understanding of model-view separation itself.

The JFace layer is contained in the bundle `org.eclipse.jface`, with extensions in »9.3.3 `org.eclipse.jface.databinding` and `org.eclipse.jface.text`. For historical reasons, it also relies on some elements of `org.eclipse.core.runtime`, which can be used ↰7.4.2 outside of the platform in just the way that we launched SWT applications as standard Java applications.

9.3.1 Viewers

»12.2.2 The basic approach of JFace is shown in Fig. 9.6(a). JFace establishes a layer between the application's business logic and the bare-bones SWT widgets. JFace uses methods like `setText` and `setIcon` to actually display the data in widgets and registers for low-level events as necessary. It also offers events to the application itself, but these are special in that they translate from the widget level to the model level. For instance, when a user selects a row in a `Table` widget, SWT reports the index of the row. JFace translates that index into the model element it has previously rendered in the row, and reports that this model element has been selected. In effect, the application is shielded from the cumbersome details and can always work in terms of its own data structures. Of course, it still listens to events such as button

clicks directly on the SWT widgets, and translates those into operations on ↰9.2.1
the model. JFace follows model-view separation in getting the data to be ↰9.1
displayed from the model and listening to change notifications of the model
to keep the display up-to-date.

We will now discuss the various roles and relationships depicted in Fig. 9.6. This section focuses on the *viewers* and their collaborators. The *listeners*, which implement the application's reactions to user input in the sense of controllers, are discussed in Section 9.3.2. ↰9.2.1

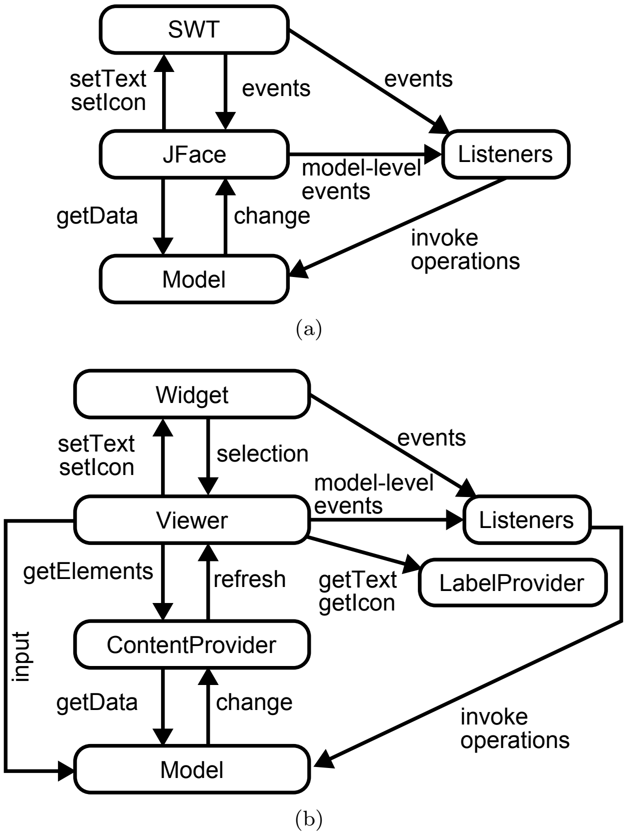


Figure 9.6 JFace Architecture

JFace viewers target specific widget types.

A core contribution of the JFace layer relates to its selection of generic *viewers*, each of which targets a specific type of widget: A `TableViewer` targets `Tables`, a `ComboViewer` targets a `Combo` combo box, and so on [Fig. 9.6(b), at the top]. Viewers use the widget-specific methods for displaying data and listen for widget-specific events.

JFace viewers access the application data through adapters.

One question not addressed in Fig. 9.6(a) is how JFace will actually access the application-specific data: How is a generic viewer supposed to know the right `getData` method and the implementation of the OBSERVER pattern of the specific data structures? Fig. 9.6(b) supplies this detail. First, each viewer holds a reference to the model, in its property *input*. However, that input is a generic `Object`, so the viewer never accesses the model itself. ↰1.3.4 ↰2.4.1 Instead, the viewer is parameterized by two adapter objects that enable it to inspect the model just as required:

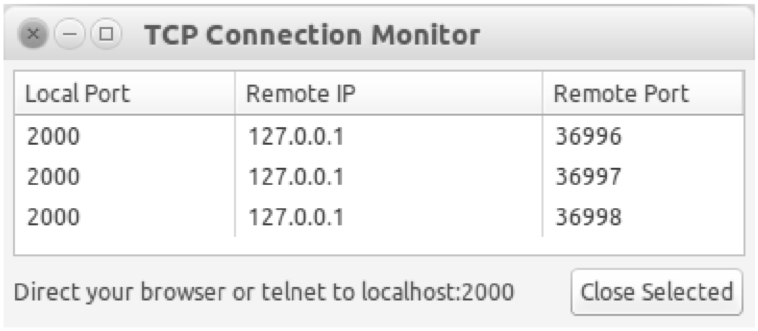
- The *content provider* is responsible for traversing the overall data structure and for splitting it up into *elements* for display purposes. For a table or list, it provides a linear sequence of elements; for a tree-like display, it also accesses the child and parent links between ↰2.4.1 the elements. Furthermore, the content provider must observe the model and notify the viewer about any changes that it receives.
- The *label provider* is called back for each element delivered by the content provider, usually to obtain concrete strings and icons to represent the element on the screen. A `ListViewer` will request one text/icon combination per element; a `TableViewer` or `TreeViewer` will request one combination for each column. The viewer will also observe the label provider to be notified about changes of the text and icons to be displayed.

The text-related viewers `TextViewer` and `SourceViewer` deviate from this schema in that they expect an implementation of `IDocument` as their model. The document itself then includes the text-specific access operations, without requiring a separate adapter.

The framework includes a deliberate redundancy regarding changes in the model: When values change within a data element, then those may be translated for the viewer either by the content provider, by calling the viewer's `update()` method, or by the label provider, by firing change events. Each mechanism has its merits. On the one hand, the content provider observes the model anyway, so the label provider can often remain passive. On the other hand, some generic label providers, such as those used in »9.3.3 data binding, may wish to avoid relying on specific content providers.

Let us start with a simple example, in which an application accepts and monitors incoming TCP connections (Fig. 9.7). Whenever a new client connects, the corresponding information gets shown. When the client disconnects, its row is removed from the table.

Keep the model independent of JFace.



TCP Connection Monitor

Local Port	Remote IP	Remote Port
2000	127.0.0.1	36996
2000	127.0.0.1	36997
2000	127.0.0.1	36998

Direct your browser or telnet to localhost:2000

Figure 9.7 Connection Monitor

We start by developing the model of the application, with the intention of keeping it independent of the user interface, and more specifically the ↰9.1 JFace API. The model here maintains a list of connections (which contain a `Socket` as the endpoint of a TCP connection). Furthermore, it implements the OBSERVER pattern, which explains the registration (and omitted de-registration) of listeners (lines 13–16), as well as the `fire` method for notifying the listeners (lines 18–20). The method `opened()` and corresponding method `closed()` will be called back from the actual server code. Since that code runs in a separate thread, all access to the internal data structures ↰8.1 needs to be protected by locking. Finally, we decide that the notification of the observers can be performed in an open call (line 10), without holding ↰8.5 on to the lock.

connections.ConnectionList

```
 1 public class ConnectionList {
 2     private ArrayList<Connection> openConnections =
 3                              new ArrayList<Connection>();
 4     private ListenerList listeners = new ListenerList();
 5
 6     void opened(Connection c) {
 7         synchronized (this) {
 8             openConnections.add(c);
 9         }
10         fireConnectionOpened(c);
11     }
12      ...
13     public synchronized void addConnectionListListener(
14                                    ConnectionListListener l) {
15         listeners.add(l);
16     }
17      ...
18     protected void fireConnectionOpened(Connection c) {
19          ...
20     }
21      ...
22 }
```

We use `synchronized` for locking because the simplicity of the use case makes it unlikely that we will ever need the flexibility of the library tools advocated in the chapter on multithreading. ↰8

The important point about the model is that it is independent of the user interface: It serves as a central list in which the server code manages the open connections, it synchronizes the different possible accesses, and it notifies interested observers. These observers are completely agnostic of a possible implementation in the user interface as well:

connections.ConnectionListListener

```
public interface ConnectionListListener extends EventListener {
  void connectionOpened(ConnectionList p, Connection c);
  void connectionClosed(ConnectionList p, Connection c);
}
```

This finishes the model in Fig. 9.6(b). We will now fill in the remaining bits.

Create the widget and its viewer together.

The viewer in Fig. 9.6(b) is linked tightly to its SWT widget: The type of widget is fixed, and each viewer can fill only a single widget, since it keeps track of which data it has displayed at which position within the widget. One therefore creates the viewer and the widget together. If a viewer is created without an explicit target widget, it will create the widget by itself. The viewer constructor also takes the parent widget and flags, as usual for SWT. ↰7.1 The SWT widget is not encapsulated completely, since the display-related services, such as computing layouts, are accessed directly.

connections.Main.createContents

```
connectionsViewer = new TableViewer(shell, SWT.BORDER);
connections = connectionsViewer.getTable();
connections.setLayoutData(new GridData(
                               SWT.FILL, SWT.FILL, true, true,
                               2, 1));
connections.setHeaderVisible(true);
```

Connect the viewer to the model through a special content provider.

Each model has, of course, a different structure and API, so that each model will also require a new content provider class. The viewer then receives its own instance of that class.

connections.Main.createContents

```
connectionsViewer.setContentProvider(
                     new ConnectionListContentProvider());
```

The reason for this one-to-one match between content provider object and viewer object is that the content provider usually has to be linked up very tightly between the viewer and its input [Fig. 9.6(b)]. The life cycle of the content provider clarifies this. Whenever the viewer receives a new input, it notifies its content provider through the `inputChanged()` method. The method must also make sure to de-register from the previous input (lines 8–9). When the viewer is disposed, with the SWT widget, it calls the method again with a new input of `null`. The logic for de-registering from the old model therefore also kicks in at the end of the life cycle. At this point, the viewer calls the content provider's `dispose()` method for any additional cleanup that may be necessary. ↰2.1.2

connections.ConnectionListContentProvider

```
 1 public class ConnectionListContentProvider implements
 2         IStructuredContentProvider, ConnectionListListener {
 3     private ConnectionList list;
 4     private TableViewer viewer;
 5     public void inputChanged(Viewer viewer, Object oldInput,
 6                              Object newInput) {
 7         this.viewer = (TableViewer) viewer;
 8         if (list != null)
 9             list.removeConnectionListListener(this);
10         this.list = (ConnectionList) newInput;
11         if (list != null)
12             list.addConnectionListListener(this);
13     }
14     public void dispose() {}
15      ...
16 }
```

Line 7 in this code assumes that the viewer is a `TableViewer`. This can be justified by stating in the class's contract that the content provider may be used only with that kind of viewer. The non-redundancy principle then decrees that line 7 must not check whether the contract is actually obeyed. Many content providers in the Eclipse code base are more defensive, or general, at this point and do something sensible for different kinds of viewers. ↰4.5

The content provider knows how to traverse the model's structure.

The content provider in Fig. 9.6(b) is an adapter that provides the interface expected by the JFace viewer on top of the application's model. Designing this interface is an interesting task: Which kind of common structure can one expect to find on all models? The approach in JFace is to start from the minimal requirements of the `TableViewer`, as the (main) client: A table is a linear list of rows, so the viewer has to be able to get the data elements behind these table rows. In the current example, each row is a `Connection` and the model already provides a method to obtain the current list. The ↰3.2.2

`inputElement` is the viewer's input model passed to `inputChanged()`; passing it again enables stateless and therefore shareable content providers.

connections.ConnectionListContentProvider.getElements

```
public Object[] getElements(Object inputElement) {
    return ((ConnectionList) inputElement).getOpenConnections();
}
```

To see more of the idea of generic interface components, let us consider briefly a tree, rendered in a `TreeViewer`. A tree has more structure than a flat table: The single elements may have children, and all but the top-level elements have a parent. Tree-like widgets usually enable multiple top-level elements, rather than a single root, so that the content provider has the same method `getElements()` as the provider for flat tables.

org.eclipse.jface.viewers.ITreeContentProvider

```
public interface ITreeContentProvider
                        extends IStructuredContentProvider {
    public Object[] getElements(Object inputElement);
    public Object[] getChildren(Object parentElement);
    public Object getParent(Object element);
    public boolean hasChildren(Object element);
}
```

Now the JFace viewer can traverse the application model's data structure by querying each element in turn. As long as the model has a table-like or tree-like structure, respectively, it will fit the expectations of the JFace layer. In general, each viewer expects a specific kind of content provider stated in its documentation, according to the visual structure of the targeted widget.

You may find it rather irritating that all viewers offer only the generic method shown next, which does not give an indication of the expected type. The deeper reason is that it is in principle not possible to override a method and specialize its the parameter types, because this *co-variant* overriding breaks polymorphism: A client that works with only the base class might unsuspectingly pass a too-general object. Java therefore requires overriding methods to have exactly the same parameter types. 60

org.eclipse.jface.viewers.StructuredViewer

```
public void setContentProvider(IContentProvider provider)
```

For simple display cases where the model does not change, one can also use the `ArrayContentProvider`, which accepts a `List` or an array and simply returns its elements. Since it does not have any state, it implements the SINGLETON pattern. ↰1.3.8

The label provider decides on the concrete visual representation.

In the end, SWT shows most data on the screen as text, perhaps with auxiliary icons to give the user visual hints for interpreting the text, such as a green check mark to indicate success. The label provider attached to JFace viewers implements just this transformation, from data to text and icons. In the example, the table has three columns for the local port, the remote IP, and the remote port. All of this data is available from the `Socket` stored in the connection, so the label provider just needs to look into the right places and format the data into strings.

connections.ConnectionListLabelProvider

```
public class ConnectionListLabelProvider
                       extends LabelProvider
                       implements ITableLabelProvider {
     ...
    public String getColumnText(Object element, int columnIndex) {
        Connection c = (Connection) element;
        switch (columnIndex) {
        case 0: return Integer.toString(c.getLocalPort());
        case 1: return c.getRemoteAddr().getHostAddress();
        case 2: return Integer.toString(c.getRemotePort());
        default:
            throw new IllegalArgumentException();
        }
    }
}
```

A corresponding `getIcon()` method remains empty here. If icons are allocated for the specific label provider, they must be freed in its `dispose()` method, which the viewer calls whenever the widget disappears from the screen. ↰7.4.1

The base class `LabelProvider`, or actually its superclass `BaseLabelProvider`, implements an observer pattern that enables concrete label providers to notify viewers about changes in the choice of text or icon. Model changes are usually handled through the content provider, as seen next.

By separating the concerns of model traversal and the actual display, JFace gains flexibility. For instance, different viewers might show different aspects and properties of the same model, so that the same content provider can be combined with different label providers.

The viewer manages untyped `Objects`.

We have found that at this point it is useful to get a quick overview of the viewer's mechanisms, so as to better appreciate the respective roles and the interactions of the viewer, the content provider, and the label provider. At the same time, these interactions illustrate the concept of generic mechanisms, which will become fundamental in the area of frameworks and for providing extensibility. »11.1 »12.3

Fig. 9.8 shows what happens from the point where the application supplies the model until the data shows up on the screen. The input is forwarded to the content provider, which chops up the overall model into elements. The viewer passes each of these elements to the label provider and receives back a string. It then displays that string on the screen. For deeper structures, the viewer queries children of elements, and again hands each of these to the label provider, until the structure is exhausted.

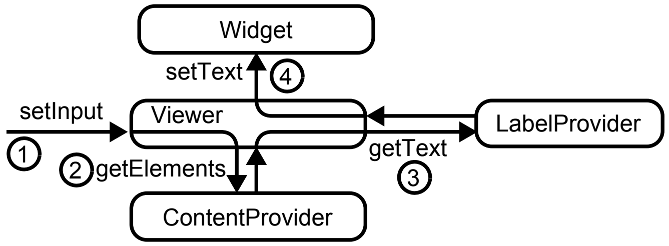


Figure 9.8 The Sequence for Displaying Data Through Viewers

In the end, the viewer's role is to manage untyped objects belonging to the application's model: It keeps references to the model and all elements as `Objects`. Whenever it needs to find out more about such an object, it passes the object to the content or label provider. In this way, the viewer can implement powerful generic display mechanisms without actually knowing anything about the application data.

Forward change notifications to the viewer.

We have now set up the display of the initial model. However, the model changes over the time, and it fires change notifications. Like any adapter [Fig. 2.10(b) on page 137], the content provider must also translate those notifications for the benefit of the viewer [Fig. 9.6(b)].

Toward that end, JFace viewers offer generic notification callbacks that reflect the possible changes in the abstract list or tree model that they envision in their content provider interface. A `TableViewer`, for instance, has callbacks for additions, insertions, deletions, and updates of single elements. The difference between `update()` and `refresh()` is that the first method locally recomputes the labels in a single table entry, while the latter indicates structural changes at the element, though it is relevant only for trees.

org.eclipse.jface.viewers.AbstractTableViewer

```
public void add(Object element)
public void insert(Object element, int position)
public void remove(Object element)
public void update(Object element, String[] properties)
public void refresh(Object element)
```

Even though these methods are `public` as a technical necessity, they are not for general use but are intended for the content provider only. In particular, they do *not* add or remove the given elements from the underlying model.

As a matter of optimization, viewers offer variants of these methods for bulk updates of several objects passed in an array. For large models, `TableViewer` and `TreeViewer` also support lazy population of the widget through the `SWT.VIRTUAL` flag passed to the constructor. In this case, the content provider can implement `ILazyContentProvider` or `ILazyTreeContentProvider`, respectively. The viewer will call the content provider only for rows that become visible—for instance, by scrolling or unfolding of tree nodes. The overhead of initially filling the entire widget is avoided.

In the running example, connections can be added to and removed from the list of current connections. The content provider listens to these changes and notifies the viewer accordingly. Since the server uses several threads for the processing of client connections, the content provider must also switch to the event thread to notify the viewer. ↰7.10.1

connections.ConnectionListContentProvider.connectionOpened

```
public void connectionOpened(ConnectionList p, final Connection c) {
    viewer.getControl().getDisplay().asyncExec(new Runnable() {
        public void run() {
            viewer.add(c);
        }
    });
}
```

Swing's `JTable` takes a different—and very interesting—approach to these notifications, which renders the idea of an "adapter" more clearly. Its method `setModel()` ↰2.4.1 accepts any implementation of `TableModel`, and that model must provide the OBSERVER ↰2.1 pattern for `TableModelListener`s. The generic `JTable` widget merely registers as one such listener. The JFace perspective, in contrast, is that the content provider is a close collaborator of a single viewer, which it notifies directly. This approach saves the effort of implementing the OBSERVER pattern. ↰2.1.4

Viewers provide higher-level services at the application level.

JFace viewers offer more services than just a mapping from application model to screen display. For instance, they enable the application code to work almost entirely at the level of the application model. Consequently, SWT widgets, for example, represent the concept of "selection" by publishing the indices of selected elements. JFace viewers, in contrast, publish `IStructuredSelection` objects, which are basically sets of model elements. Furthermore, viewers do not map elements directly, but perform preprocessing steps for filtering and sorting. As a final example, they implement mechanisms for inline editing: When the user clicks "into" a table cell, the table viewer creates a small overlay containing an application-specific `CellEditor` that fills the cell's screen space but is, in fact, a stand-alone widget. ↰7.6

↰9.1 Sorting and filtering are interesting in themselves as an instance of model-view separation: The fact that a user prefers, in certain situations and for certain tasks, to see only a selection of elements in a particular order, must be dealt with independently of the core functionality—after all, the next view or the next user may have entirely different preferences. ↰2.4.1 For instance, Eclipse's Java Model reflects the structure of the Java source code. The underlying abstract syntax tree keeps declarations in the order of their appearance within a class. At the interface level, the user may prefer seeing only `public` members or having the members be ordered alphabetically, as seen in the *Package Explorer.*

9.3.2 Finishing Model-View-Controller with JFace

↰9.2.1 JFace viewers already cover much of the MODEL-VIEWER-CONTROLLER pattern, in that the screen reliably mirrors the state of the application's functional core. The only missing aspect is that of controllers, which interpret the raw user input as requests for performing operations on the model. This will happen in the event-listeners shown in Fig. 9.6.

JFace enables controllers to work on the application model.

Suppose that we wish to implement the button labeled "Close" in Fig. 9.7. Since the button itself is an SWT widget independent of any viewer, we attach a listener as usual: ↰7.1

connections.Main.createContents

```
Button btnClose = new Button(shell, SWT.NONE);
btnClose.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        handleCloseSelected();
    }
});
```

The method `handleCloseSelected()` then relies heavily on support from JFace. Line 3 retrieves the viewer's selection, which maps the indices of rows selected in the table widget to the model elements shown in those rows. As a result, line 5 can ask for the first (and only) selected element and be sure to obtain a `Connection`, because the viewer's content provider has delivered instances of only that class. The crucial point now is that the actual logic for implementing the desired reaction in line 7 remains at the application level: The model's `Connection` objects also offer a method `close()` for terminating the TCP connection with the client. ↰9.3.1

connections.Main

```
1 protected void handleCloseSelected() {
2     IStructuredSelection s =
3             (IStructuredSelection) connectionsViewer.getSelection();
4     Connection selectedConnection =
5                 (Connection) s.getFirstElement();
6     if (selectedConnection != null) {
7         selectedConnection.close();
8     }
9 }
```

The implementation of the `Connection`'s close method at first seems simple enough: We simply have to close the underlying TCP connection.

connections.Connection

```
public void close() throws IOException {
    channel.close();
}
```

However, this method finally runs in the event thread, while the server is concurrently processing client input in background threads. This use case is not supported by the basic `Socket` API, but only by the *asynchronously closeable* TCP connections introduced with the NIO API in Java 1.4. The details are explained in the documentation of the interface `InterruptibleChannel`. ↰7.10.1 ↰7.10

Screen updates follow the MVC pattern.

Let us finally reconsider the fundamental reaction cycle of the Model-View-Controller pattern: The window system delivers events to the view, which forwards them to the controller, which interprets them as requests for operations on the model, which sends change notifications to the view, which repaints parts of the data on the screen. So far, we have seen the first half: SWT delivers the button click to the application's event-listener, which serves as a controller and decides that the selected connection should be closed. ↰9.2.1

And now something really interesting happens, because the model is not a simple list, but involves side effects on the underlying TCP connections. Executing `close()` on the connection goes down to the operation system, which will declare the connection terminated some time later. This, in turn, causes the `read()` method accepting client input (line 4 in the next code snippet) to return with result "end of stream," which terminates the server loop (lines 4–6). As a result, this particular server thread terminates (line 10), but not before notifying the `ConnectionList` about this fact (line 8).

connections.Server.run

```
 1 public void run() {
 2     list.opened(conn);
 3      ...
 4     while (channel.read(buf) != -1) {
 5          ...  send input back to client as demo
 6     }
 7      ...
 8     list.closed(conn);
 9      ...
10 }
```

Upon receiving this latter signal, the MVC mechanisms kick in to effect the screen update: The `ConnectionListContentProvider` observes the model and translates the incoming `connectionClosed()` event into a `remove()` notification of the table viewer, which removes the corresponding row from the SWT display. That's it.

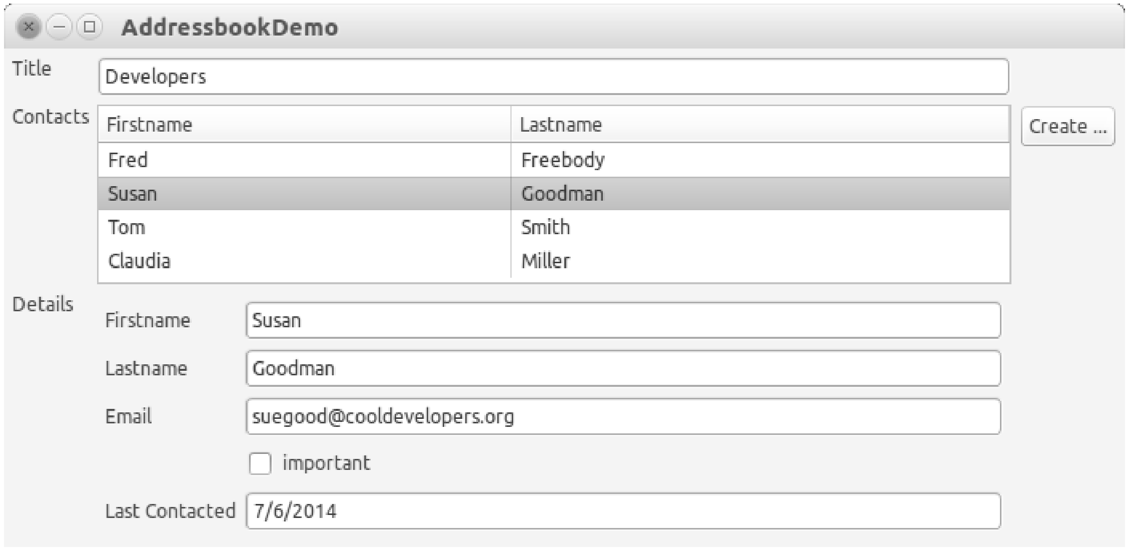
9.3.3 Data Binding

The mechanisms of JFace presented so far make it fairly simple to display
data so that the screen is kept up-to-date when the data changes. How-
ever, the content and label providers have to be programmed by hand, and
changing the data is not supported by the framework at all. The concept
of *data binding* addresses both concerns. Broadly speaking, data binding
↰1.3.3 maps the individual properties of beans to widgets such as text fields or
lists. One also says that the properties are *bound to* the widgets, or more
symmetrically that the property and the widget are *bound*.

↰7.2 The WindowBuilder includes a graphical tool for creating bindings, so
that data binding makes it simple to bridge the model-view separation by
quickly creating input masks for given model elements. The usage is mostly
intuitive: Select two properties to be bound and click the "bind" button.
We will therefore discuss only the few nonobvious cases.

📖201 Many frameworks cover the concept of data binding. For instance, *JavaServer Faces*
📖234 (JSF) allows you to bind input components to model fields declaratively through spe-
📖222 cial *Expression Language* (EL) annotations. *Ruby on Rails* is famed for its effective way
of creating input masks through simple form helpers. Microsoft's *Windows Presentation
Foundation* (WPF) offers an especially comprehensive treatment of data binding.

We will discuss the details of data binding in JFace using the example
of editing an *address book*, which is essentially a list of *contacts* (Fig. 9.9).
↰1.3.3 The `AddressBook` and its `Contact` objects are simple Java beans; that
↰2.1 is, their state consists of public properties and they send change notifi-
cations. From top to bottom in Fig. 9.9, we see the following features of



AddressbookDemo

Title: Developers

Contacts

Firstname	Lastname
Fred	Freebody
Susan	Goodman
Tom	Smith
Claudia	Miller

Details

Firstname	Susan
Lastname	Goodman
Email	suegood@cooldevelopers.org
	☐ important
Last Contacted	7/6/2014

Figure 9.9 Address Book Editor

data binding, ordered by increasing complexity: The address book's *title* property is bound to a text field; its *contacts* property is a list of `Contact` beans shown in a JFace viewer. In a master/detail view, the details of the currently selected contact are shown in the lower part. Here, the *first name*, *last name*, and *email* properties of the contact are, again, bound directly to text fields. The *important* property holds a Boolean value and demonstrates the support for different types. Finally, the *last contacted* property introduces the challenge of converting between the internal `Date` property and the `String` content of the text field. ↰9.3.1

↰7.2.2 ↰9.1

The JFace data binding framework lives in several plugins, which must be set as dependencies in the `MANIFEST.MF` of any plugin using the framework. In the spirit of model-view separation, those parts not dealing with the user interface reside in `org.eclipse.core.databinding`, `org.eclipse.core.databinding.beans`, and `org.eclipse.core.property`. Those connected to the display directly reside in the plugin `org.eclipse.jface.databinding`.

9.3.3.1 Basics of Data Binding

The data binding framework is very general and is meant to cover many possible applications. Fig. 9.10 gives an overview of the elements involved in one binding. The endpoints, to the far left and right, are the widget and bean created by the application. The purpose of a *binding* is to synchronize the value of selected properties in the respective beans. Bindings are, in principle, symmetric: They transfer changes from one bean to the other, and vice versa. Nevertheless, the terminology distinguishes between a *model* and the *target* of a binding, where the target is usually a widget. The figure also indicates the role of data binding in the general scheme of model-view separation. ↰9.1

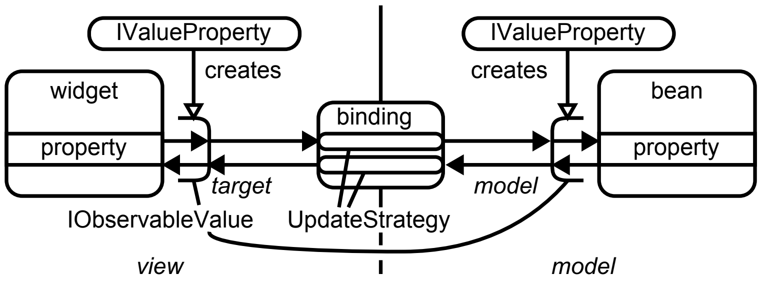


Figure 9.10 Overview of JFace Data Binding

To keep the data binding framework independent of the application objects, these are adapted to the `IObservableValue` interface in the next code snippet, as indicated by the half-open objects beside the properties in Fig. 9.10. The adapters enable getting and setting a value, as well as observing changes, as would be expected from the basic MVC pattern. The ↰2.4.1 ↰9.2.1

value type is used for consistency checking within the framework, as well as for accessing the adaptees efficiently by reflection.

org.eclipse.core.databinding.observable.value.IObservableValue

```
public interface IObservableValue extends IObservable {
    public Object getValueType();
    public Object getValue();
    public void setValue(Object value);
    public void addValueChangeListener(
                      IValueChangeListener listener);
    public void removeValueChangeListener(
                      IValueChangeListener listener);
}
```

The `IObservableValue` in this code captures values of atomic types. There are analogous interfaces `IObservableList`, `IObservableSet`, and `IObservableMap` to bind properties holding compound values.

Creating these adapters often involves some analysis, such as looking up the getter and setter methods for a named property by reflection. The adapters are therefore usually created by `IValueProperty` objects, which serve as abstract factories. Again, analogous interfaces `IListProperty`, ↰1.4.12 `ISetProperty`, and `IMapProperty` capture factories for compound value properties.

org.eclipse.core.databinding.property.value.IValueProperty

```
public interface IValueProperty extends IProperty {
    public Object getValueType();
    public IObservableValue observe(Object source);
     ...  observing parts of the value
}
```

We have now discussed enough of the framework to bind the *name* property of an `AddressBook` in the field `model` to a text field in the interface. Lines 1–2 in the next code snippet create an `IValueProperty` for the *text* property of an SWT widget and use it immediately to create the adapter for the `bookname` text field. The code specifies that the property is considered changed whenever the user leaves the field (event `SWT.FocusOut`); setting the event to `SWT.Modify` updates the model property after every keystroke. Lines 3–4 proceed analogously for the *name* property of the `AddressBook`. Finally, lines 5–6 create the actual binding.

databinding.AddressBookDemo.initDataBindings

```
1 IObservableValue observeTextBooknameObserveWidget =
2     WidgetProperties .text(SWT.FocusOut).observe(bookname);
3 IObservableValue nameModelObserveValue =
4     BeanProperties.value("name") .observe(model);
5 bindingContext.bindValue(observeTextBooknameObserveWidget,
6                          nameModelObserveValue, null, null);
```

A *binding context* manages a set of bindings. The two `null` values in line 6 indicate that no update strategies (Fig. 9.10) are required.

↰7.10.1 The framework anticipates the possibility of multithreading in the model, which requires switching to the event dispatch thread at appropriate moments. Each observable value is said to live in a specific *realm*. One realm, accessible by `SWTObservables.getRealm()`, is associated with the event thread. A *default realm* can be set with `Realm.runWithDefault()`, so that it is usually not necessary to specify a realm explicitly for individual values.

9.3.3.2 Master/Detail Views

Fig. 9.9 includes a typical editing scenario: The list *contacts* is a *master* list showing an overview; below this list, several fields give access to the *details* of the currently selected list element. The master list itself involves only binding a property, as seen in the following code snippet. On the viewer side, special content and label providers then accomplish the data access and updates. ↰9.3.1

databinding.AddressBookDemo.initDataBindings

```
IObservableList contactsModelObserveList = BeanProperties
        .list("contacts").observe(model);
contactsViewer.setInput(contactsModelObserveList);
```

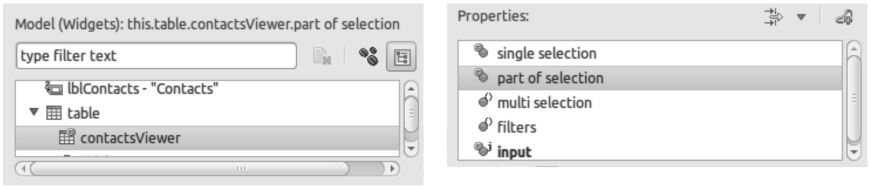
The actual master/detail view is established by a two-step binding of properties. Lines 3–4 in the next example create a possibly changing value that tracks the currently selected `Contact` element as a value: Whenever the selection changes, the value of the property changes. Building on this, lines 5–8 create a two-step access path to the *first name* property: The `observeDetail()` call tracks the current `Contact` and registers as an observer for that contact, so that it also sees its property changes; the `value()` call then delivers an atomic `String` value for the property. Through these double observers, this atomic value will change whenever either the selection or the *first name* property of the current selection changes.

databinding.AddressBookDemo.initDataBindings

```
1 IObservableValue observeTextTxtFirstObserveWidget =
2     WidgetProperties.text(SWT.Modify).observe(txtFirst);
3 IObservableValue observeSingleSelectionContactsViewer =
4     ViewerProperties.singleSelection().observe(contactsViewer);
5 IObservableValue contactsViewerFirstnameObserveDetailValue =
6     BeanProperties
7         .value(Contact.class, "firstname", String.class)
8         .observeDetail(observeSingleSelectionContactsViewer);
```

```
bindingContext.bindValue(observeTextTxtFirstObserveWidget,
        contactsViewerFirstnameObserveDetailValue, null, null);
```

At this point, the usage of the WindowBuilder is somewhat unintuitive, because the "model" side of the binding involves the JFace-level selection. The two panels shown here appear on the right-hand, model side of the WindowBuilder's *bindings* page. In the first, one has to select the *widgets* tree, instead of the *beans*, in the upper-right corner. From the table *contactsViewer* appearing in the second panel, one then chooses *part of selection*. The subsequent dialog requests a choice of the selection's content type and desired detail field.



9.3.3.3 Data Conversion and Validation

We finish this section on data binding by discussing the crucial detail of *validation and conversion*. The need arises from the fact that the model's data is stored in formats optimized for internal processing, while the user interface offers only generic widgets, so that the data must often be displayed and edited in text fields. One example is the *last contacted* property of a `Contact`, which internally is a `Date`, but which is edited as a text with a special format (Fig. 9.9).

The basic property binding follows, of course, the master/detail approach. The new point is the use of *update strategies* (Fig. 9.10), as illustrated in the next code snippet. Each binding can be characterized by separate strategies for the two directions of synchronization. Lines 1–5 specify that the text entered in the interface should be converted to a `Date` to be stored in the model, and that this transfer should take place only if the text is in an acceptable format. The other direction in lines 6–8 is less problematic, as any `Date` can be converted to a string for display. Lines 9–11 then create the binding, with the specified update policies. ↰9.3.3.2

databinding.AddressBookDemo.initDataBindings

```
1 UpdateValueStrategy targetToModelStrategy =
2                             new UpdateValueStrategy();
3 targetToModelStrategy.setConverter(new StringToDateConverter());
4 targetToModelStrategy.setAfterGetValidator(
5                            new StringToDateValidator());
6 UpdateValueStrategy modelToTargetStrategy =
7                            new UpdateValueStrategy();
8 modelToTargetStrategy.setConverter(new DateToStringConverter());
9 bindingContext.bindValue(observeTextTxtLastcontactedObserveWidget,
        contactsViewerLastContactedObserveDetailValue,
        targetToModelStrategy, modelToTargetStrategy);
```

To demonstrate the mechanism, let us create a custom converter, as specified by the `IConverter` interface. The method `convert()` takes a string. It returns `null` for the empty string and otherwise parses the string into a specific format. It treats a parsing failure as an unexpected occurrence. ↰1.5.7

databinding.StringToDateConverter

```
public class StringToDateConverter implements IConverter {
    static SimpleDateFormat formatter =
                        new SimpleDateFormat("M/d/yyyy");
     ...  source and destination types for consistency checking
    public Object convert(Object fromObject) {
        String txt = ((String) fromObject).trim();
        if (txt.length() == 0)
            return null;
        try {
            return formatter.parse(txt);
        } catch (ParseException e) {
            throw new IllegalArgumentException(txt, e);
        }
    }
}
```

The validator checks whether a particular string matches the application's expectations. In the present case, it is sufficient that the string can be converted without error, which is checked by attempting the conversion. In other cases, further restrictions can be suitable.

databinding.StringToDateValidator

```
public class StringToDateValidator implements IValidator {
    public IStatus validate(Object value) {
        try {
            StringToDateConverter.formatter.parse((String) value);
            return Status.OK_STATUS;
        } catch (ParseException e) {
            return ValidationStatus.error("Incorrect format");
        }
    }
}
```

Conversion and validation are specified separately since they often have to vary independently. Very often, the converted value has to fulfill further restrictions beyond being convertible, such as a date being within a specified range. Also, even data that is not constrained by the internal type, such as an email address stored as a `String`, must obey restrictions on its form.

Very often, it is simpler to validate the converted value rather than the raw format. Update strategies offer `setAfterConvertValidator` and `setBeforeSetValidator` for this purpose. Both work on the result of conversion. The only difference is that the latter may not be called in case the update strategy is configured not to update the model at all (see `ValueBinding.doUpdate()` for the details).

The class `MultiValidator` provides mechanisms for checking cross-field constraints, such as the end date of some activity being later than its start date.

Conversion and validation touch upon a central aspect of user interfaces—namely, the fact that the interface belongs to the system boundary. The boundary has the special obligation to check all incoming data to avoid corrupting the system's internal structures and to prevent malicious attacks. Furthermore, it must convert all data into the internal formats, to prepare it for efficient processing. Validators therefore do not simply check that the data is convertible, but also check that the data is acceptable to the system as a whole. Conversion and validation therefore create a uniform framework to handle these aspects, and this explains their presence in many of the major interface toolkits. ↰1.5.2 ↰4.6

Another observation concerns the relation between validation and conversion. Most converters cannot handle all inputs allowed by their expected input types. In other words, their `convert()` method has an implicit pre-condition. The role of the validator is to check that the pre-condition is fulfilled before the framework attempts the actual conversion. This relation also explains why the example validator refers back to the converter: It simply ensures a perfect match of the checked condition and the required condition. ↰4.1

9.3.4 Menus and Actions

We have seen that JFace viewers connect generic SWT widgets such as lists or tables to an application model [Fig. 9.6(b) on page 473]: The viewer queries the data structures and maps the data to text and icons within the widget. It also listens to model changes and updates the corresponding entries in the widget.

A similar mechanism is used for adding entries to menus and toolbars. SWT offers only basic `MenuItem`s, which behave like special `Button`s and notify attached listeners when they have been clicked. SWT menu items, just like other widgets, are passive: While they can show a text and icon, and can be enabled or disabled, they wait for the application to set these properties. ↰7.1

To keep this chapter self-contained, the presentation here refers to the example application MiniXcel, a minimal spreadsheet editor to be introduced in Section 9.4. For now, it is sufficient to understand that at the core, a `SpreadSheetView` displays a `SpreadSheet` model, as would be expected from the MODEL-VIEW-CONTROLLER pattern. ↰9.2.1

Actions represent application-specific operations.

JFace connects SWT menus to application-specific *actions*, which implement `IAction` (shown next). Actions wrap code that can act directly on the application's model (lines 6–7). But actions also describe themselves for display purposes (lines 3–4), and they identify themselves to avoid showing duplicates (line 2). Finally, it is anticipated that an action's properties will change, in much the same way that an application's model changes (lines 9–12). ↰1.8.6 ↰9.1

org.eclipse.jface.action.IAction

```
1  public interface IAction {
2      public String getId();
3      public String getText();
4      public ImageDescriptor getImageDescriptor();
5
6      public void run();
7      public void runWithEvent(Event event);
8
9      public void addPropertyChangeListener(
10                             IPropertyChangeListener listener);
11     public void removePropertyChangeListener(
12                             IPropertyChangeListener listener);
13      ...  setters for the properties and further properties
14 }
```

100 »9.5.1 The concept of an "action" that acts as a self-contained representation of some operation is virtually universal. One variant of the COMMAND pattern captures the idea: Swing has a very similar interface `Action`, Qt has a `QAction` class, and so on.

Contribution items connect menu items to actions.

To connect SWT's passive menu items to the application's available actions, JFace introduces *menu managers* and *contribution items* (Fig. 9.11, upper part). Each menu is complemented by a menu manager that fills the menu and updates it dynamically when the contributions change. Each SWT menu item is complemented by a contribution item that manages its appearance. Initially, it fills the menu item's text, icon, and enabled state. Whenever a property of the action changes, the contribution item updates the menu item correspondingly. In the reverse direction, the contribution item listens for clicks on the menu item and then invokes the action's `run()` method (or more precisely, the `runWithEvent()` method).

Actions are usually shared between different contribution managers.

One detail not shown in Fig. 9.11 is that action objects are independent of the concrete menu or toolbar where they get displayed. They are not

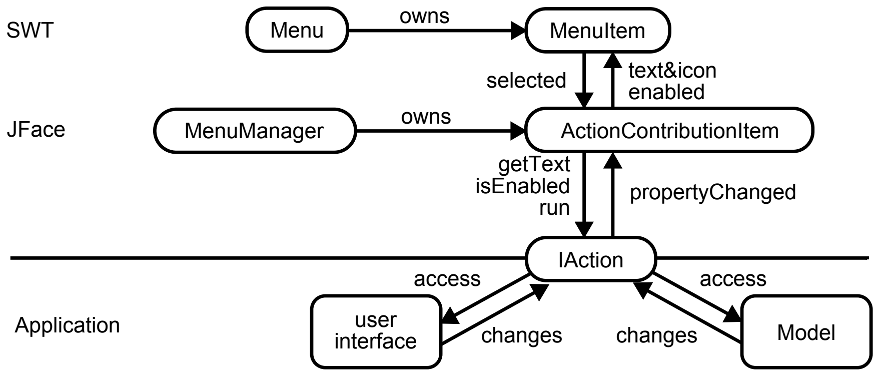


Figure 9.11 Menus and Actions in JFace

simply an elegant way of filling a menu, but rather represent an operation and thus have a meaning in themselves. Eclipse editors usually store their actions in a local table, from where they can be handed on to menus and toolbars. In the example, we use a simple hash map keyed on the action's `ids`.

minixcel.ui.window.MainWindow

```
private Map<String, IAction> actions =
                        new HashMap<String, IAction>();
```

Create the menu manager, then update the SWT widgets.

Once the table holds all actions, a concrete menu can be assembled quickly: Just fill a menu manager and tell it to update the menu. For instance, the MiniXcel spreadsheet application has an edit menu with typical undo and redo actions, as well as a "clear current cell" action. Lines 1–8 create the structure of nested menu managers. Lines 9–11 flush that structure into the visible SWT menu. »9.4

minixcel.ui.window.MainWindow.createContents

```
 1 MenuManager menu = new MenuManager();
 2  ...  set up File menu
 3 MenuManager editMenu = new MenuManager("Edit");
 4 menu.add(editMenu);
 5 editMenu.add(actions.get(UndoAction.ID));
 6 editMenu.add(actions.get(RedoAction.ID));
 7 editMenu.add(new Separator("cellActions"));
 8 editMenu.add(actions.get(ClearCellAction.ID));
 9 shlMinixcel.setMenuBar(menu.createMenuBar(
10                               (Decorations)shlMinixcel));
11 menu.updateAll(true);
```

The cast to `Decorations` in line 10 is necessary only because an overloaded method taking a `Shell` argument is now deprecated.

Actions are usually wired to some context.

The lower part of Fig. 9.11 highlights another aspect of action objects: They are self-contained representations of some operation that the user can invoke through the user interface. The `run()` method is the entry point; everything else is encapsulated in the concrete action. This means, however, that the action will be linked tightly to a special context. In the example, the action that clears the currently selected cell must certainly find and access that cell, so it needs a reference to the `SpreadSheetView`. (The command processor `cmdProc` is required for undoable operations, as seen later on.) »9.5

minixcel.ui.window.MainWindow

```
private void createActions() {
     ...
    actions.put(ClearCellAction.ID,
                new ClearCellAction(spreadSheetView, cmdProc));
}
```

174 The same phenomenon of exporting a selection of possible operations is also seen in Eclipse's wiring of actions into the global menu bar. There, again, the actions are created inside an editor component but get connected to the global menu and toolbar. This larger perspective also addresses the question of how global menu items are properly linked up to the currently open editor.

9.4 The MVC Pattern at the Application Level

So far, we have looked at the basic Model-View-Controller pattern and its implementation in the JFace framework. The examples have been rather small and perhaps a little contrived, to enable us to focus on the mechanisms and crucial design constraints. Now it is time to scale the gained insights to the application level. The question we will pursue is how model-view separation influences the architecture of the overall product. Furthermore, we will look at details that need to be considered for this scaling, such as incremental repainting of the screen.

The running example will be a minimal spreadsheet application *Mini-Xcel* (Fig. 9.12). In this application, the user can select a cell in a special widget displaying the spreadsheet, and can enter a formula into that cell, possibly referring to other cells. The application is responsible for updating all dependent cells automatically, as would be expected.

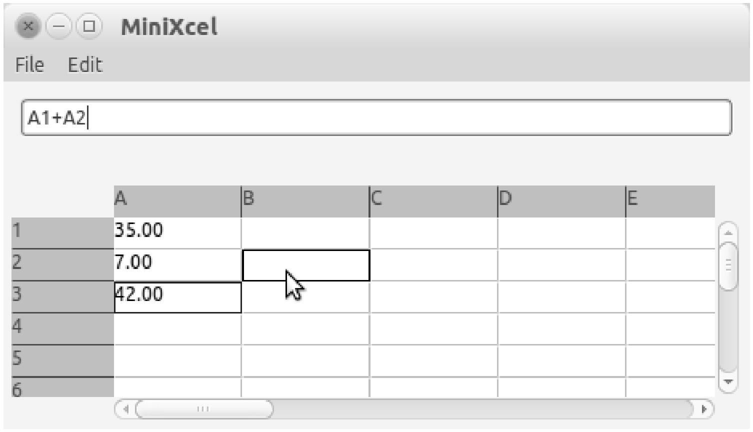


Figure 9.12 The MiniXcel Application

The application offers enough complexity to explore the points mentioned previously. First, the model contains dependencies between cells in the form of formulas, and the parsing of and computation with formulas constitutes a nontrivial functionality in itself. At the interface level, we need a custom-painted widget for the spreadsheet, which must also offer view-level visual feedback and a selection mechanism to link the spreadsheet to the input line on top.

9.4.1 Setting up the Application

The overall structure of the application is shown in Fig. 9.13. The `SpreadSheet` encapsulates the functional core. It manages *cells*, which can be addressed from the outside by usual *coordinates* such as `A2` or `B3`, as well as their interdependencies given by the stored formulas. A *formula* is a tree-structured COMPOSITE that performs the actual computations. A simple (shift-reduce) *parser* transforms the input strings given by the user into structured formulas. The core point of model-view separation is implemented by making all functionality that is not directly connected to the user interface completely independent of considerations about the display. ↰9.1 ↰2.3.1 📖2 ↰9.1

The main window (Fig. 9.12) consists of two parts: the `SpreadSheetView` at the bottom and the `CellEditor` at the top. These two are coupled loosely: The `SpreadSheetView` does not assume that there is a single `CellEditor`. Instead, it publishes a generic `IStructuredSelection` containing the currently selected `Cell` model element. When the user presses "enter," the cell editor can simply call `setFormula` on that `Cell`. This has two effects. First, the dependent cells within the spreadsheet are updated by reevaluating their formulas. Second, all updated cells will notify the view, through their surrounding `SpreadSheet` model. »12.1 ↰9.3.2 ↰2.2.4

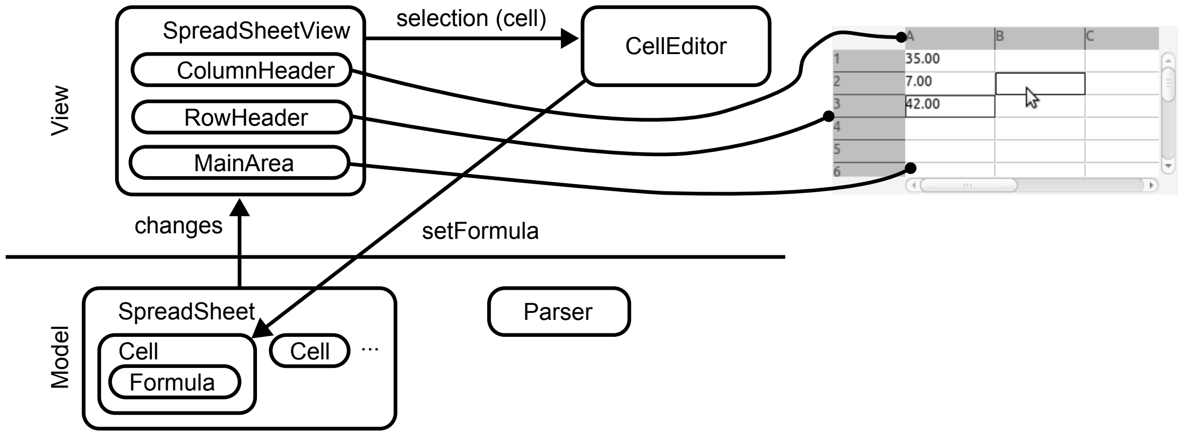


Figure 9.13 Structure of the MiniXcel Application

Despite the visual similarity between Fig. 9.13 and Fig. 9.3, the `CellEditor` is *not* the controller for the `SpreadSheetView`. The `CellEditor` is a stand-alone widget that, as we will see, contains a view and a controller, where the controller invokes the `set Formula` operation noted in Fig. 9.13.

9.4.2 Defining the Model

We can give here only a very brief overview of the model code and highlight those aspects that shape the collaboration between user interface and model. The central element of the model is the `SpreadSheet` class. It keeps a sparse mapping from coordinates to `Cells` (line 2) and creates cells on demand as they are requested from the outside (lines 5–12). The model implements the OBSERVER pattern as usual to enable the view to remain up-to-date (lines 4, 14–16, 18–20). The class `Coordinates` merely stores a row and column of a cell.

minixcel.model.spreadsheet.SpreadSheet

```java
1  public class SpreadSheet {
2      private final HashMap<Coordinates, Cell> cells =
3                        new HashMap<Coordinates, Cell>();
4      private final ListenerList listeners = new ListenerList();
5      public Cell getCell(Coordinates coord) {
6          Cell res = cells.get(coord);
7          if (res == null) {
8              res = new Cell(this, coord);
9              cells.put(coord, res);
10         }
11         return res;
12     }
13
14     public void addSpreadSheetListener(SpreadSheetListener l) {
15         listeners.add(l);
16     }
17      ...
18     void fireCellChanged(Cell cell) {
19          ...
20     }
21      ...
22 }
```

A real-world implementation that scales to hundreds and thousands of rows full of data would probably create a matrix of cells, rather than a hash map. However, it must be noted that each cell in the spreadsheet will have to carry additional information, such as the dependencies due to formulas, so it might be useful to make cells into objects in any case. Only their organization into the overall spreadsheet would differ.

Application models usually have internal dependencies.

Each `Cell` in the spreadsheet must store the user's input (line 4 in the next code snippet) and must be prepared to evaluate that formula quickly (line 5). Since the view will query the current value rather frequently and other cells will require it for evaluating their own formulas, it is sensible to cache that value rather than repeatedly recomputing it (line 6). As further basic data, the cell keeps its owner and the position in that owner (lines 2–3). ↰2.2.1

---

10 2 We have decided to keep the original formula string, because the parsed formula loses information about parentheses and whitespace. Real-world spreadsheets keep an intermediate form of *tokens* (called "parse thing," or PTG in this context) resulting from lexing, rather than full parsing. If whitespace is kept, the original representation can be restored. If the tokens are stored post-order, formula evaluation is quick as well. ↰2.3.4 A further advantage of this representation is that references can be updated when cell contents are moved.

The example of spreadsheets also shows that an application model is rarely as simple as, for instance, a list of Java beans. ↰1.3.3 Usually, the objects within the model require complex interdependencies and collaborations to implement the desired functionality. In `Cells`, we store the (few) cross references introduced by the `formula` in two lists: `dependsOn` lists those cells whose values are required in the `formula`; `dependentOnThis` is the inverse relationship, which is required for propagating updates through the spreadsheet.

minixcel.model.spreadsheet.Cell

```
1  public class Cell {
2      final SpreadSheet spreadSheet;
3      private final Coordinates coord;
4      private String formulaString = "";
5      private Formula formula = null;
6      private Value cachedValue = new Value();
7      private final List<Cell> dependsOn = new ArrayList<Cell>();
8      private final List<Cell> dependentOnThis =
9                                  new ArrayList<Cell>();
10
11      ...
12 }
```

Clients cannot adequately anticipate the effects of an operation.

One result of the dependencies within the model is that clients, such as the controllers in the user interface, cannot foresee all the changes that are effected by an operation they call. As a result, the controller of the MVC could not reliably notify the view about necessary repainting even without interference from other controllers. This fact reinforces the crucial design decision of updating the view by observing the model. ↰9.2.3

In the current example, the prototypical modification is setting a new formula on a cell. The overall approach is straightforward: Clear the old dependency information, and then set and parse the new input. Afterward, we can update the new dependencies by asking the formula for its references and recomputing the current cached value.

minixcel.model.spreadsheet.Cell

```java
public void setFormulaString(String formulaString) {
    clearDependsOn();
    this.formulaString = formulaString;
     ...  special cases such as an empty input string
    formula = new Formula(spreadSheet.getFormulaFactory(),
                          formulaString);
    fillDependsOn();
     ...  check for cycles
    recomputeValue();
}
```

The update process of a single cell now triggers updating the dependencies as well: The formula is evaluated and the result is stored.

minixcel.model.spreadsheet.Cell.recomputeValue

```java
private void recomputeValue() {
     ...
    setCachedValue(new Value(formula.eval(
                              new SpreadSheetEnv(spreadSheet))));
     ...  error handling on evaluation error
}
```

The cache value is therefore the "current" value of the cell. Whenever that changes, two stakeholders must be notified: the dependent cells within the spreadsheet and the observers outside of the spreadsheet. Both goals are accomplished in the method `setCachedValue()`:

minixcel.model.spreadsheet.Cell

```java
protected void setCachedValue(Value val) {
    if (val.equals(cachedValue))
        return;
    cachedValue = val;
    for (Cell c : dependentOnThis)
        c.recomputeValue();
    spreadSheet.fireCellChanged(this);
}
```

This brief exposition is sufficient to highlight the most important points with respect to model-view separation. Check out the online supplement for further details—for instance, on error handling for syntax errors in formulas and cyclic dependencies between cells.

9.4.3 Incremental Screen Updates

Many applications of model-view separation are essentially simple, with small models being displayed in small views. Yet, one often comes across the other extreme. Even a simple text viewer without any formatting must be careful to repaint only the portion of text determined by the scrollbars, and from that only the actually changing lines. Otherwise, the scrolling and editing process will become unbearably slow. The MiniXcel example is sufficiently complex to include a demonstration of the necessary processes.

Before we delve into the details, Fig. 9.14 gives an overview of the challenge. Put very briefly, it consists of the fact that even painting on the screen is event-driven: When a change notification arrives from the model, one never paints the corresponding screen section immediately. Instead, one asks to be called back for the job later on. In some more detail, the model on the left in Fig. 9.14 sends out some change notification to its observers. The view must then determine where it has painted the modified data. That area of the screen is then considered "damaged" and is reported to the window system. The window system gathers such damaged areas, subtracts any parts that are not visible anyway, coalesces adjacent areas, and maybe performs some other optimizations. In the end, it comes back to the view requesting a certain area to be repainted. At this point, the view determines the model elements overlapping this area and displays them on the screen. ↰7.8

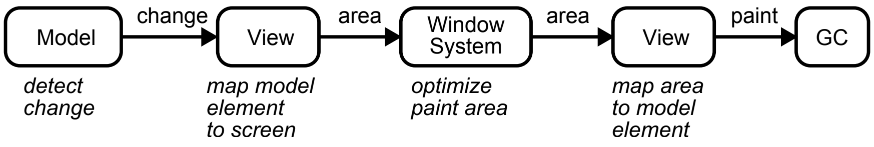


Figure 9.14 Process of Incremental Screen Updates

A further reason for this rather complex procedure, besides the possibility of optimizations, is that other events, such as the moving and resizing of windows, can also require repainting, so that the right half of Fig. 9.14 would be necessary in any case. The extra effort of mapping model elements to screen areas in the left half is repaid by liberating the applications of optimizing the painting itself.

Let us track the process in Fig. 9.14 from left to right, using the concrete example of the MiniXcel `SpreadSheetView`. At the beginning, the view receives a change notification from the model. If the change concerns a single cell, that cell has to be repainted.

minixcel.ui.spreadsheet.SpreadSheetView.spreadSheetChanged

```
public void spreadSheetChanged(SpreadSheetChangeEvent evt) {
    switch (evt.type) {
    case CELL:
        redraw(evt.cell.getCoordinates());
        break;
        ...
    }
}
```

It will turn out later that cells need to be repainted on different occasions, such as to indicate selection or mouse hovering. We therefore implement the logic in a helper method, shown next. The method `redraw()` called on the `mainArea` of the view is provided by SWT and reports the area as damaged. »9.4.4 ↰1.4.8 ↰1.4.5

minixcel.ui.spreadsheet.SpreadSheetView

```
public void redraw(Coordinates coords) {
    Rectangle r = getCellBounds(coords);
    mainArea.redraw(r.x, r.y, r.width, r.height, false);
}
```

In a real implementation, the method `getCellBounds()` would determine the coordinates by the sizes of the preceding columns and rows. To keep the example simple, all columns have the same width and all rows have the same height in MiniXcel. This finishes the left half of Fig. 9.14. Now it is the window system's turn to do some work.

minixcel.ui.spreadsheet.SpreadSheetView

```
protected Rectangle getCellBounds(Coordinates coords) {
    int x = (coords.col - viewPortColumn) * COL_WIDTH;
    int y = (coords.row - viewPortRow) * ROW_HEIGHT;
    return new Rectangle(x, y, COL_WIDTH, ROW_HEIGHT);
}
```

In the right half of Fig. 9.14, the `MainArea` is handed a paint request for a given rectangular area on the screen, in the form of a `PaintEvent` passed to the method shown next. This method determines the range of cells touched by the area (line 3). Then, it paints all cells in the area in the nested loops in lines 7 and 11. As an optimization, it does not recompute the area covered by each cell, as done for the first cell in line 5. Instead, it moves that area incrementally, using cells that are adjacent in the view (lines 9, 14, 16).

minixcel.ui.spreadsheet.MainArea.paintControl

```
1  public void paintControl(PaintEvent e) {
2       ...  prepare colors
3      Rectangle cells = view.computeCellsForArea(e.x, e.y, e.width,
4                                                 e.height);
5      Rectangle topLeft = view.computeAreaForCell(cells.x, cells.y);
6      Rectangle cellArea = Geometry.copy(topLeft);
7      for (int row = cells.y; row < cells.y + cells.height; row++) {
8          cellArea.height = SpreadSheetView.ROW_HEIGHT;
9          cellArea.x = topLeft.x;
10         cellArea.width = SpreadSheetView.COL_WIDTH;
11         for (int col = cells.x;
12              col < cells.x + cells.width; col++) {
13             paintCell(col, row, cellArea, gc);
14             cellArea.x += cellArea.width;
15          }
16          cellArea.y += cellArea.height;
17      }
18  }
```

»11.1 »11.2 Note that the `MainArea` delegates the actual computation of cell areas in lines 3–5 to its owner, the `SpreadSheetView`. Since that object was responsible for mapping cells to areas, it should also be responsible for the inverse computations, to ensure that any necessary adaptations will be performed consistently to both.

The actual painting code in `paintCell()` is then straightforward, if somewhat tedious. It has to take into account not only the cell content, but also the possible selection of the cell and a mouse cursor being inside, both of which concern view-level logic treated in the next section. Leaving all of that aside, the core of the method determines the current cell value, formats it as a string, and paints that string onto the screen (avoiding the creation of yet more empty cells):

minixcel.ui.spreadsheet.MainArea

```java
private void paintCell(int col, int row,
                       Rectangle cellArea, GC gc) {
    if (view.model.hasCell(new Coordinates(col, row))) {
        cell = view.model.getCell(new Coordinates(col, row));
        Value val = cell.getValue();
        String displayText;
        displayText = String.format("%.2f", val.asDouble());
        gc.drawString(displayText, cellArea.x, cellArea.y, true);
    }
}
```

This final painting step finishes the update process shown in Fig. 9.14. In summary, incremental repainting achieves efficiency in user interface programming: The view receives detailed change notifications, via the "push" ↰2.1.3 variant of the OBSERVER pattern, which it translates to minimal damaged areas on the screen, which get optimized by the window system, before the view repaints just the model elements actually touched by those areas.

9.4.4 View-Level Logic

↰9.2.5 We have seen in the discussion of the MVC pattern that widgets usually include behavior such as visual feedback that is independent of the model itself. MiniXcel provides two examples: selection of cells and feedback about the cell under the mouse. We include them in the discussion since this kind of behavior must be treated with the same rigor as the model: Users consider only applications that react consistently and immediately as trustworthy.

Treat selection as view-level state.

Most widgets encompass some form of selection. For instance, tables, lists, ↰9.3.2 and trees allow users to select rows, which JFace maps to the underlying model element rendered in these rows. The interesting point about selection is that it introduces view-level state, which is orthogonal to the application's core model-level state.

We will make our `SpreadSheetView` a good citizen of the community by implementing `ISelectionProvider`. That interface specifies that clients can query the current selection, set the current selection (with appropriate elements), and listen for changes in the selection. The last capability will also enable us to connect the entry field for a cell's content to the spreadsheet (Fig. 9.13). For simplicity, we support only single selection and introduce a corresponding field into the `SpreadSheetView`.

minixcel.ui.spreadsheet.SpreadSheetView

```
Cell curSelection;
```

The result of querying the current selection is a generic `ISelection`. Viewers that map model elements to screen elements, such as tables and trees, usually return a more specific `IStructuredSelection` containing these elements. We do the same here with the single selected cell.

minixcel.ui.spreadsheet.SpreadSheetView.getSelection

```
public ISelection getSelection() {
    if (curSelection != null)
        return new StructuredSelection(curSelection);
    else
        return StructuredSelection.EMPTY;
}
```

Since the selection must be broadcast to observers and must be mirrored on the screen, we introduce a private setter for the field.

minixcel.ui.spreadsheet.SpreadSheetView

```
private void setSelectedCell(Cell cell) {
    if (curSelection != cell) {
        Cell oldSelection = curSelection;
        curSelection = cell;
        fireSelectionChanged();
         ...  update screen from oldSelection to curSelection
    }
}
```

The remainder of the implementation of the OBSERVER pattern for selection is straightforward. However, its presence reemphasizes the role of selection as proper view-level state. ↰2.1

Visual feedback introduces internal state.

The fact that painting is event-driven, so that a widget cannot paint visual feedback immediately, means that the widget must store the desired feedback as private state, determine the affected screen regions, and render the feedback in the callback (Fig. 9.14). ↰7.8

For MiniXcel, we wish to highlight the cell under the mouse cursor, so that users know which cell they are targeting in case they click to select it. The required state is a simple reference. However, since the state is purely ↰9.4.2 view-level, we are content with storing its coordinates; otherwise, moving over a yet unused cell would force the model to insert an empty `Cell` object.

minixcel.ui.spreadsheet.SpreadSheetView

```
Coordinates curCellUnderMouse;
```

Setting a new highlight is then similar to setting a new selected cell:

minixcel.ui.spreadsheet.SpreadSheetView

```
protected void setCellUnderMouse(Coordinates newCell) {
    if (!newCell.equals(curCellUnderMouse)) {
        Coordinates oldCellUnderMouse = curCellUnderMouse;
        curCellUnderMouse = newCell;
         ...  update screen from old to new
    }
}
```

The desired reactions to mouse movements and clicks are implemented by the following simple listener. The `computeCellAt()` method returns the cell's coordinates, also taking into account the current scrolling position. While selection then requires a real `Cell` object from the model, the targeting feedback remains at the view level.

minixcel.ui.spreadsheet.SpreadSheetView.mouseMove

```
public void mouseMove(MouseEvent e) {
    setCellUnderMouse(computeCellAt(e.x, e.y));
}
public void mouseDown(MouseEvent e) {
    setSelectedCell(model.getCell(computeCellAt(e.x, e.y)));
}
```

The painting event handler merges the visual and model states.

The technical core of visual feedback and view-level state, as shown previously, is not very different from the model-level state. When painting the widget, we have to merge the model- and view-level states into one consistent overall appearance. The following method achieves this by first painting the cell's content (lines 4–5) and overlaying this with a frame, which is either a selection indication (lines 9–12), the targeting highlight (lines 13–17), or the usual cell frame (lines 19–23).

minixcel.ui.spreadsheet.MainArea

```
1 private void paintCell(int col, int row,
2                         Rectangle cellArea, GC gc) {
3     ...
4         displayText = String.format("%.2f", val.asDouble());
5         gc.drawString(displayText, cellArea.x, cellArea.y, true);
```

```
    Rectangle frame = Geometry.copy(cellArea);
    frame.width-;
    frame.height-;
    if (view.curSelection != null && view.curSelection == cell) {
        gc.setForeground(display.getSystemColor(
                                    SWT.COLOR_DARK_BLUE));
        gc.drawRectangle(frame);
    } else if (view.curCellUnderMouse != null
            && view.curCellUnderMouse.col == col
            && view.curCellUnderMouse.row == row) {
        gc.setForeground(display.getSystemColor(SWT.COLOR_BLACK));
        gc.drawRectangle(frame);
    } else {
        gc.setForeground(display.getSystemColor(SWT.COLOR_GRAY));
        int bot = frame.y + frame.height;
        int right = frame.x + frame.width;
        gc.drawLine(right, frame.y, right, bot);
        gc.drawLine(frame.x, bot, right, bot);
    }
}
```

According to this painting routine, the view-level state is always contained within the cells to which it refers. It is therefore sufficient to repaint these affected cells when the state changes. For the currently selected cell, the code is shown here. For the current cell under the mouse, it is analogous.

minixcel.ui.spreadsheet.SpreadSheetView

```
private void setSelectedCell(Cell cell) {
    if (curSelection != cell) {
         ...
        if (oldSelection != null)
            redraw(oldSelection.getCoordinates());
        if (curSelection != null)
            redraw(curSelection.getCoordinates());
    }
}
```

This code is made efficient through the incremental painting pipeline shown in Fig. 9.14 on page 500 and implemented in the code fragments shown earlier. Because the pipeline is geared toward painting the minimal necessary number of cells, it can also be used to paint single cells reliably and efficiently.

9.5 Undo/Redo

Users make mistakes all the time, especially with highly developed and optimized user interfaces, where small graphical gestures have powerful effects. Most of the time, they realize their mistakes immediately afterward, because the screen gets updated with the new application state and ↰453
the result does not match their expectations. A fundamental requirement for any modern application is the ability to cancel operations immediately through an “undo” action and to “redo” them if it turns out that the effect

was desired after all. This section discusses the established technique for solving this challenge: The application maintains a list of incremental and undoable changes to the model. We first consider a minimal version to highlight the technique, then we briefly examine various implementations within the Eclipse platform to get an overview of practical issues involved.

9.5.1 The Command Pattern

↰1.1 The fundamental obstacle for undoing editing operations is, of course, the stateful nature of objects: Once existing data has been overwritten, it cannot be restored. For instance, the `CellEditor` in the spreadsheet application (at the top of Fig. 9.12 on page 495) enables the user to enter the new formula or value for the selected cell. When the user presses "enter," the new formula gets set on the model, as shown in the next code snippet. The model automatically updates the dependent cells. ↰9.4.2 After executing this code, the previous formula is irretrievably lost and it is not possible to "undo" the operation.

minixcel.ui.celledit.CellEditor

```
protected void putFormula() {
    if (curCell != null) {
        curCell.setFormulaString(txtFormula.getText());
    }
}
```

To implement undo/redo, the overall goal is to create a conceptual *history* of operations, as shown in Fig. 9.15. At each point in time, the current model state is the result of executing a sequence of operations. These operations can be undone, with the effect that the model reverts to a previous state. Operations that have been undone become redoable, so that later model states can be reached again if necessary.

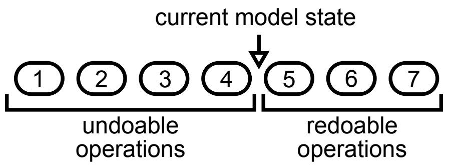


Figure 9.15 History for Undo/Redo

Controllers delegate the invocation of operations to *Command* objects.

To implement undo/redo, one modifies the MODEL-VIEW-CONTROLLER pattern from Fig. 9.3 (page 454) in one tiny detail into the version shown in Fig. 9.16: The *controller* no longer invokes model operations directly, but creates *Command* objects that invoke the operations.

📖100 This central insight is captured by the COMMAND pattern.

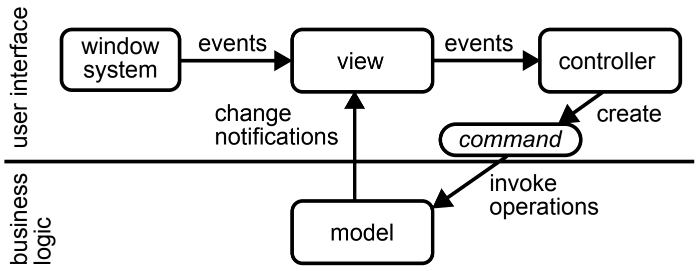


Figure 9.16 MVC with Undoable Operations

PATTERN: COMMAND

If you need undoable operations, or need to log or store operations, encapsulate them as objects with `execute()` and `undo()` methods.

1. Define an interface `Command` with methods `execute()`, `undo()`, and `redo()`.
2. Provide an abstract base class defining `redo()` as a call to `execute()`.
3. Define a command class, implementing the `Command` interface, for each operation on the model. Store all necessary parameters as fields in the *Command* object. This includes in particular references to the target objects that the operation works with.
4. Let each command's `execute()` method invoke methods on the model to perform the operation. Before that, let it store the state it destroys in fields inside the command.
5. Let each command's `undo()` method revert the change to the model using the stored previous state.

We will now explore the details of this concept and the implementation at the example of the spreadsheet editor. Steps 1 and 2, and their motivation, are deferred to Section 9.5.2.

Let the commands capture incremental state changes.

The central point of the pattern is that commands must capture enough of the previous model state to be able to restore it. In the example of setting the formula in a spreadsheet cell, we just have to keep the cell's previous formula. In the code snippet that follows, line 4 sets the new formula, but only after saving the old value in line 3. In this way, the operation can be undone in line 7.

minixcel.commands.SetCellFormulaCommand

```
public void execute() {
    Cell c = model.getCell(coordinates);
    oldFormulaString = c.getFormulaString();
    c.setFormulaString(formulaString);
}
public void undo() {
    model.getCell(coordinates).setFormulaString(oldFormulaString);
}
```

Making each operation into a separate command object then has the advantage of creating a space for that additional data. In the current example, it consists of a single field `oldFormulaString`, but more may be required for more complex operations.

It is important for efficiency to keep an incremental record of the changed data—that is, to store only those data items that are actually necessary for restoring the model to the previous state. For instance, when deleting a (small) part of a text document in a `DeleteTextCommand`, you should keep only the deleted text, not the entire document.

Do not fetch the old state already in the command's constructor. At first glance, the difference seems negligible, because one usually creates a command and executes it immediately afterward (by passing it to the command processor, as seen in Section 9.5.2). However, when composing commands, as seen later in this section, other commands may actually intervene between the construction and the execution of a command, so that the data stored for the later undo is actually wrong. The only reliable technique is to fetch the old state in the `execute()` method, just before actually changing the state.

↰4.1

Thinking in terms of assertions is the crucial trick at this point: If you want to establish, in the example, that "`oldFormulaString` holds the content seen before setting the new formula," the only reliable way of achieving this is to actually look up that string right before setting the new one.

Introduce a `CompoundCommand` to make the approach scalable.

Very often, one operation from the user's perspective requires a series of method invocations on the model. To achieve this effectively, it is useful to introduce a `CompoundCommand`, which maintains a list of commands and executes and undoes them as suggested by the concept of a history.

minixcel.commands.CompoundCommand

```
public class CompoundCommand implements Command {
    private List<Command> commands;
     ...
    public void execute() {
        for (int i = 0; i != commands.size(); i++) {
            commands.get(i).execute();
        }
    }
    public void undo() {
        for (int i = commands.size() - 1; i >= 0; i--) {
            commands.get(i).undo();
        }
    }
    ...
}
```

The overall effort of implementing undo/redo then becomes manageable: One has to go through writing a command class for every elementary operation offered by the model once, but afterward the operations required by the user interface can be composed almost as effectively as writing a sequence of method calls.

The method `redo()` must leave exactly the same state as `execute()`.

Commands usually come with a separate method `redo()` that is invoked after `undo()` and must reexecute the command's target operation. More precisely, this method must leave the model in exactly the same state as the original `execute()` did, because the later operations in the history (Fig. 9.15) may depend on the details of that state.

In the current case of setting a spreadsheet cell, the `execute()` method is so simple that `redo()` can behave exactly the same way:

minixcel.commands.SetCellFormulaCommand.redo

```
public void redo() {
    execute();
}
```

In some situations, however, `redo()` may differ from `execute()` and will then require a separate implementation:

- If `execute()` creates new objects and stores them in the model, then `redo()` must store exactly the same objects, rather than creating new ones, because later operations may contain references to the new objects so as to access or modify them.
- If `execute()` accesses some external state, such as the clipboard, a file, or some data from another editor, which may not be governed by the same history of commands, then that state must be stored, because it might change between `execute()` and `redo()`.
- Similarly, if `execute()` makes decisions based on some external state, that state—or better still the decision—must be stored and used in the redo operation.
- If `execute()` asks the user, through dialogs, for more input or a decision, then that input or decision must be stored as well.

Again, the COMMAND offers just the space where such additional information is stored easily.

Make a command a self-contained description of an operation.

To be effective, commands must store internally all data necessary for executing the intended operation. Obviously, this includes the parameters passed to the invoked method. It also includes any target objects that the operation works on. In the example, we have to store the spreadsheet itself, the cell to be modified, and the formula to be stored in the cell. For simplicity, we keep the coordinates of the cell, not the `Cell` object itself.

minixcel.commands.SetCellFormulaCommand

```
public class SetCellFormulaCommand implements Command {
    private SpreadSheet model;
    private Coordinates coordinates;
    private String formulaString;
    private String oldFormulaString;
    public SetCellFormulaCommand(SpreadSheet model,
                                 Coordinates coordinates,
                                 String formulaString) {
        ...
    }
    ...
}
```

↰2.2.1

⇄? Alternatively, one could have said that the command is not about `SpreadSheet`s at all, but about single `Cell`s, which may happen to be contained in a `SpreadSheet`. Then, the first two fields would be replaced by a single field `Cell cell`, with a change to the constructor to match.

Be sure to make `undo()` revert the model state exactly.

One challenge in defining the commands' methods is that they must match up exactly: Invoking `execute()` and then `undo()` must leave the model in exactly the same state as it was at the beginning. The reason is seen in Fig. 9.15 on page 506: Each operation in the sequence in principle depends on the model state that it has found when it was first executed. Calling `undo()` must then reconstruct that model state, because the later `redo()` will depend on the details. A `DeleteTextCommand`, for instance, may contain the offset and length of the deletion, and it would be disastrous if undoing and redoing a few operations were to invalidate that text range.

↰4.1

The necessary precision can be obtained by thinking in terms of assertions: The contracts of the invoked operations specify their effects precisely, so that the command can gauge which parts of the state need to be stored for the undo.

Do not neglect possible internal dependencies of the model.

Let us reconsider the example code from the perspective of a precise `undo()` method. The `execute()` method sets a given cell. Ostensibly, it just changes a single property in line 3 in the next code snippet. The `undo()` method reverts that property to `oldFormulaString`, so that everything should be fine.

minixcel.commands.SetCellFormulaCommand.execute

```
1 Cell c = model.getCell(coordinates);
2 oldFormulaString = c.getFormulaString();
3 c.setFormulaString(formulaString);
```

Two effects may cause the internal model state to deviate from the original. ↰9.4.2 First, the call to `getCell()` in line 1 might actually create the cell object in the data structure. Second, and perhaps more importantly, the call in line 3 implicitly updates all dependent cells.

However, both points are irrelevant in regard to the overall goal of keeping the undo/redo history intact. Clients cannot distinguish whether a `Cell` they receive from `getCell()` has just been created or had already existed. The model treats the sparse representation of the spreadsheet content as a strictly internal issue. The dependencies between cells do not cause problems either, because the reevaluation of formulae is strictly deterministic, so that setting the old formula also resets all dependent cells to their previous values.

This explanation rests on the idea of the externally visible state, which is captured in an object's model fields: The command stores all relevant public state before the modification and restores that state to undo the operations. Since clients cannot actually observe any internal difference between two states that are indistinguishable from an external perspective, their behavior cannot depend on the difference either. ↰4.1 ↰4.2.2

One snag in the example concerns the external format of the spreadsheet written to disk: The model may choose to write out the cell created in line 1 of the previously given `execute()` method, even if that cell has been emptied out by `undo()` in the meantime. In the present case, one can argue that any programmatic access, after reloading the spreadsheet document, can still not observe the difference. In other cases, where the external format is the really important thing, such differences may not be acceptable. As an example, Eclipse's editor for OSGi bundles is really just a front-end for the underlying configuration files such as `plugin.xml` and `MANIFEST.MF`. Adding some extension and then undoing that addition should leave the file structure untouched. »12.3.3 »A.1.2

Java's Swing framework introduces an interesting alternative perspective on undo/redo, which already integrates the possible necessity of tracking changes to the model's internals. Rather than requiring commands to store the previous state, the model itself sends out `UndoableEdit` notifications upon any change. These notifications contain sufficient internal information to undo the change and offer public `undo()` and `redo()` methods. For a typical example, see Swing's `HTMLDocument`. Clients, such as editors, have to track only these notifications, using the provided `UndoManager`.

Use mementos to encapsulate internal state, but only if really necessary.

In some rare cases, the internal state is so complex that you would rather not rely on all effects being reliably undone when resetting the public state to the previous value. In particular, if the internal dependencies are nondeterministic, or may become nondeterministic in the future, some further measures have to be taken. We mention the idea only very briefly and refer you to the literature for the details. 100

PATTERN: MEMENTO

If clients must store snapshots of the internal state for later reference, package those snapshots into impenetrable *Memento* objects.

Define a `public` *Memento* class with only `private` fields and no `public` accessors as a nested class inside the model. The private fields hold copies of particular state elements from the model. Although clients can handle such objects—the pattern says they are *Caretakers*—they can never inspect the internal state wrapped up in the memento objects. For the *Caretakers*, introduce a method `createMemento()` that captures the current state and a method `setMemento()` to revert to that state.

Fig. 9.17 illustrates the idea: The application model has some complex internal state. It also offers public methods for copying out some of the state, but that state remains hidden inside the memento object, as indicated by the double lines. Further public methods enable the clients to restore old states by passing the memento back to the model. As suggested in the COMMAND pattern, it is usually sensible to keep only incremental updates inside the mementos. 100

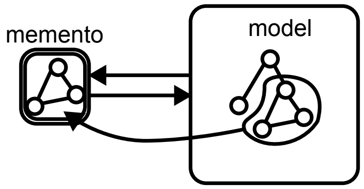


Figure 9.17 Idea of the Memento Pattern

⚠ Do not introduce MEMENTO without good reasons. The pattern is rather disruptive to the model's implementation, because any operation must track all changes it makes in a memento, which is both complex and possibly inefficient. For an example of such overhead, you might want to look at Swing's `HTMLDocument` class. Conceptually, one can also argue that the pattern partially violates model-view separation, because view-level requirements infiltrate the model's definition. As a benefit, the availability of the extra information might make undo/redo much more efficient.

9.5.2 The Command Processor Pattern

We have now finished examining the core of undo/redo: Any operation on the model is represented as a *Command* object, and that object is responsible for keeping enough of the previous model state for restoring that state later on. It remains, however, to manage the overall sequence of commands executed on the model. As Fig. 9.15 (on page 506) has clarified, each command in the overall history implicitly assumes that all previous commands have executed properly so that it can perform its own operation. The COMMAND PROCESSOR pattern handles exactly this new aspect. ↰9.5.1 59

PATTERN: COMMAND PROCESSOR

If you introduce COMMAND for undo/redo, also centralize the execution and reversal of the operations that they represent. The *Controller*s, or other parts wanting to interact with the model, create *Command*s and pass them to a *CommandProcessor*. The *CommandProcessor* alone decides about and keeps track of the proper order of calls to the *Command*s' methods.

1. Maintain the command history in fields (Fig. 9.15).
2. Offer public `execute(Command)`, `undo()`, and `redo()` methods.
3. Implement the OBSERVER pattern for history changes.

As a preliminary prerequisite to introducing such a command processor, all commands must have a uniform structure. As already envisaged in the COMMAND pattern, we introduce an interface to capture the available methods. Since `redo()` in the majority of cases is the same as `execute()`, it is useful to have an abstract base class where `redo()` just calls `execute()`. ↰9.5.1 ↰3.1.4

minixcel.commands.Command

```
public interface Command {
    void execute();
    void undo();
    void redo();
}
```

The command processor can then implement the history from Fig. 9.15 in the form of two stacks of commands. We also lay the foundation for the OBSERVER pattern.

minixcel.commands.CommandProcessor

```
public class CommandProcessor {
    private Stack<Command> undoList;
    private Stack<Command> redoList;
    private ListenerList listeners = new ListenerList();
     ...
}
```

Associate each model with a unique command processor.

↰9.15 The nature of a command history implies that no modifications must ever circumvent the mechanism: If the current model state changes by a direct invocation of model methods, the undoable commands as well as the redoable commands may fail because they originally executed in different situations. For instance, when one deletes some text in a text document directly, then any command storing the start and length of a character range may suddenly find that it is using illegal positions.

It is therefore necessary to create a (or to choose an existing) unique command processor that manages all changes to a given model. One command processor may, of course, manage several models at once to enable »9.5.4 operations that work across model boundaries.

Channel all operations on the model through its command processor.

Whenever the user, or some part of the system, wishes to work with the model, it will create a command object and pass it to the command processor. In the MiniXcel example, the `CellEditor` enables the user to input a new formula for the selected cell by creating a `SetCellFormulaCommand`.

minixcel.ui.celledit.CellEditor

```
protected void putFormula() {
    if (curCell != null) {
        cmdProc.execute(new SetCellFormulaCommand(getCurSheet(),
                curCell.getCoordinates(), txtFormula.getText()));
    }
}
```

The command processor's `execute()` method executes the given command (line 3 in the next code snippet). However, because it is responsible for managing all command executions, it does some more bookkeeping. Since the new command changes the model state, all previously redoable commands become invalid (line 2), and the new command becomes un-↰2.1 doable (line 4). Finally, the command processor is observable and sends out `commandHistoryChanged` messages (line 5), for reasons shown in a minute.

minixcel.commands.CommandProcessor

```
1 public void execute(Command cmd) {
2     redoList.clear();
```

```
    cmd.execute();
    undoList.add(cmd);
    fireCommandHistoryChanged();
}
```

Undo and redo are services offered by the command processor.

Of course, the overall undo/redo functionality is not itself implemented in the form of commands, but rather resides in the command processor. Its `undo()` method must be called only if there is, indeed, an undoable command. The method then moves that command to the redoable stack and calls its `undo()` method. Finally, it notifies the observers.

minixcel.commands.CommandProcessor

```
public void undo() {
    Assert.isTrue(!undoList.isEmpty());
    Command cmd = undoList.pop();
    cmd.undo();
    redoList.push(cmd);
    fireCommandHistoryChanged();
}
```

The user triggers undo usually through a toolbar button or menu item. These should be disabled if no command can currently be undone. The JFace method of achieving this is to create an `Action` that listens for state changes. In the current example, the base class `CommandProcessor Action` already implements this mechanism in a template method and calls `checkEnabled()` whenever the command history has changed. The action's `run()` method does the obvious thing. ↰9.3.4 ↰1.4.9

minixcel.commands.UndoAction

```
public class UndoAction extends CommandProcessorAction {
    public static final String ID = "undo";
    public UndoAction(CommandProcessor cmdProc) {
        super("Undo", cmdProc);
        setId(ID);
    }
    public void run() {
        cmdProc.undo();
    }
    protected boolean checkEnabled() {
        return cmdProc.canUndo();
    }
}
```

The implementation of a corresponding `RedoAction` is analogous.

9.5.3 The Effort of Undo/Redo

After finishing the standard mechanisms for implementing undo/redo, it is useful to pause briefly and consider the overall effort involved. Although in the end there will be no alternative to going through with it to satisfy the users, it is best to maintain a good overview so as not to underestimate the effort, but also to look actively for supporting infrastructure.

All serious UI frameworks come with undo/redo infrastructure.

The first observation is that the overall mechanisms are fairly rigid and will reoccur whenever undo/redo is required: `Command`s capture and revert changes, and some `CommandProcessor` keeps track of all executed `Command`s. The interaction between the two is limited to generic `execute()`, `undo()`, and `redo()` methods, probably together with some similarly standard extensions. »9.5.4

Many frameworks and libraries provide variants of this scheme, and one then simply has to create new types of commands for the application-specific models. ↰3.2.1 📖235 For instance, the Eclipse Modeling Framework defines a `Command` interface and a `BasicCommandStack` command processor; the Graphical Editing Framework 📖214 defines an abstract class `Command` and a `CommandStack` command processor; and Eclipse's core runtime defines an interface `IUndoableOperation` for commands and a class `DefaultOperationHistory` as a command processor.

Create commands for atomic operations, then build `CompoundCommand`s.

↰9.5.1 When using command processors, any operation on the model must be wrapped in a command at some point. However, writing a new command class for every single task that the user performs in the user interface simply does not scale. It is better to create a set of basic commands for the single operations offered by the model and to combine these as necessary using a `CompoundCommand`, which will also be available in any framework.

Write `Command`s at the model level.

A second concern is to keep the command definitions as independent of the concrete user interface as possible. When modifying or porting the user interface, as enabled by model-view separation, ↰9.1 the effort spent on more specific commands may be lost. At the same time, commands and the command processor are solely concerned with the model, and not with the user interface, so that they can be implemented at the model level.

⇄? In contrast, undo/redo is an interface-level concern, so one might argue that commands should be defined in the interface-level components. Both alternatives can be found in the Eclipse platform: »9.5.2 EMF provides generic commands on the models and in the model-level package `org.eclipse.emf.common.command`, while the IDE places workspace operations in `org.eclipse.ui.ide.undo`.

Many model-level frameworks provide atomic operations as commands.

Many libraries and frameworks are, of course, aware that professional applications require undo/redo. For instance, Eclipse's resources come equipped with commands to create, copy, move, and delete resources. The Eclipse 📖235
Modeling Framework provides modifications of bean properties of general `EObjects`, such as those created from a specific EMF model.

9.5.4 Undo/Redo in the Real World

So far, everything has been rather straightforward and to the point: While executing a command, keep enough data to enable reverting the change; to undo a command, play back that data. In real-world applications, things become somewhat more complex because side-conditions and special cases must be observed. These intricacies also explain why one cannot give a single implementation that covers all applications. We will look at three examples from the Eclipse platform: GEF, EMF, and the core platform. In each case, it is sufficient to analyze the various definitions of the command, since the command processors follow.

The Graphical Editing Framework provides powerful abstractions and 📖214
mechanisms for creating general drawing editors, in the form of editors that are not limited to standard widgets for displaying the model but create truly ↰7.1
graphical representations. Its `Command` class defines the three basic methods `execute()`, `undo()`, and `redo()`. The first practical extension is the `label` property, which is used for indicating the nature of the change in undo and ↰1.3.3
redo menu items. The remaining methods are discussed subsequently.

org.eclipse.gef.commands.Command

```java
public abstract class Command {
    public void execute()
    public void undo()
    public void redo()
    public String getLabel()
    public void setLabel(String label)
     ...
}
```

Test the applicability of commands before `execute()` and `undo()`.

One detail about commands not yet discussed is that the `execute()`, `undo()`, and `redo()` methods do not declare any thrown exceptions. This is not a careless omission, but a conceptually necessary restriction following from the overall approach: Commands are executed and undone as atomic steps in a history and they must execute either completely or not at all—any model left in a state "in between" can never be repaired, in particular not by calling `undo()`. In short, commands are best understood as transactions 📖86
on the model.

Practical frameworks therefore add methods `canExecute()` and `canUndo()` that are called before the respective methods are invoked. The command must return `true` only if these methods can run through without faults immediately afterwards.

org.eclipse.gef.commands.Command

```
public boolean canExecute()
public boolean canUndo()
```

You may rightly ask whether there should not be a `canRedo()` as well. However, since `execute()` and `redo()` must essentially perform the same operation, the latter is covered by `canExecute()` as well. An exception is seen and explained later.

GEF's implementation of the command processor will silently ignore any commands that are not executable, to avoid corrupting the model.

org.eclipse.gef.commands.CommandStack

```
public void execute(Command command) {
    if (command == null || !command.canExecute())
        return;
     ...
}
```

↰4.1 ↰6.4.1 A deeper reason for having the checking methods is that the model operations invoked by `execute()` and `undo()` will in general have pre-conditions. However, these special pre-conditions cannot be declared for `execute()` and `undo()`, because both inherit their pre-conditions from the `Command` interface (or abstract base class). The only solution is to make `canExecute()` the pre-condition of the interface's `execute()` method, so that clients are responsible for calling `canExecute()` before calling `execute()`. This reasoning also explains the implementation of `execute()` in the `CommandStack` shown here. For a ↰4.2 similar example, you can go back to the class `ThresholdArrayIterator`, where `next()` has pre-condition `hasNext()`.

Be aware of the interaction between `canExecute()` and `CompoundCommand`s. The method `canExecute()` in a `CompoundCommand` usually asks each of the commands in turn whether it can execute. This means, however, that each one checks this condition on the initial model state. During the actual execution, the contained commands are executed in order, so that they see a different state—the earlier commands in the sequence may invalidate the condition of being executable for the later commands. In most situations, this is not problematic, as long as the developer is aware of the limitation. A more faithful rendering would have to execute the commands in sequence and undo them later on—an overhead that is usually not justified. In case this becomes relevant to your application, look at `StrictCompoundCommand` from the Eclipse Modeling Framework.

Chaining enables the framework to accumulate commands easily.

In many situations, the overall operation on a group of objects can be constructed by performing the operation on each object in turn. For instance, when the user selects several elements in a drawing and presses the "delete" key, then each element can be deleted by itself to achieve the effect. The `chain()` method of a command supports the framework in assembling this operation.

org.eclipse.gef.commands.Command
```
public Command chain(Command command)
```

Expect commands to have proper life cycles.

Commands may in general need to allocate resources, such as to store some image copied from the clipboard, or they may need to listen to the model. When the command is no longer held in the history, it must free those resources, or de-register as an observer. Like other objects, commands therefore need a well-defined life cycle. The command processor is responsible for calling their `dispose()` method when they are removed from the history and are no longer required. ↰7.4.1 ↰1.1

org.eclipse.gef.commands.Command
```
public void dispose()
```

EMF adds the ability to define results and highlight target objects.

The `Command` interface defined by EMF offers the same methods as that of GEF shown earlier, and adds two more. First, the method `getResult()` allows commands to provide some abstract "result" of their execution. The `CutToClipboardCommand`, for instance, decorates a `RemoveCommand`. The `RemoveCommand` deletes a given set of objects from the model and defines those as its result; the decorator sets them as the current content of EMF's clipboard. Second, the method `getAffectedObjects()` is meant to identify objects that should be highlighted in the view, for instance by selecting them in a JFace viewer displaying the model. Both methods represent special scenarios that arise in EMF's application domain of building structured models for editors. ↰2.4.2 ↰9.3.1

org.eclipse.emf.common.command.Command
```
public interface Command {
     ...
    Collection<?> getResult();
    Collection<?> getAffectedObjects();
}
```

Possibly external operations require further measures.

Eclipse's resources framework defines the structure of the overall workspace, with projects, folders, and files. The framework also includes `IUndoable`

`Operations` that represent changes to the workspace and that allow users to revert actions on those external entities.

These undoable operations by their nature act on an external model, which explains two deviations in the `execute()` method: First, a progress monitor parameter anticipates a possibly long runtime; second, the presence of an `IStatus` return value and a declared exception indicates that these commands can actually fail, perhaps due to external circumstances such as missing files. Of course, the concrete implementations should still ensure that the model they work with is not corrupted—that they are exception-safe and preserve at least the model's invariants. Because the external state may have changed after the last `undo()`, commands are also given the chance to check that state in `canRedo()` before `redo()` gets called. ↰7.10.2 ↰1.5.6 ↰6.3

org.eclipse.core.commands.operations.IUndoableOperation

```
public interface IUndoableOperation {
    IStatus execute(IProgressMonitor monitor, IAdaptable info)
            throws ExecutionException;
     ...
    boolean canRedo();
     ...
}
```

Eclipse anticipates cross-model changes and a global history.

Many operations in Eclipse, such as refactorings on Java sources, in principle affect many files. Eclipse therefore tags each command which a *context* to which it applies. The context is then used to filter the available history.

org.eclipse.core.commands.operations.IUndoableOperation

```
boolean hasContext(IUndoContext context);
IUndoContext[] getContexts();
void addContext(IUndoContext context);
void removeContext(IUndoContext context);
```

For instance, suppose we create three classes `A`, `B`, and `C`, where `B` calls a method `f()` from `A`, but `C` does not. Then we use *Refactor/Rename* to rename the method `f()` to a method `g()`. Then the *Edit* menu for editors of `A` and `B` will show the entry "Undo rename method," while `C` shows the previous local modification to the source code there—the context of the renaming command includes the source files of `A` and `B`, but not of `C`.

9.6 Wrapping Up

This chapter touches on the core of professional software engineering. Any gifted layman can use the WindowBuilder to create a nice small application for a specific purpose, but it takes much more foresight to create a software

product that can grow and change with the demands of its users, that can
be maintained for years and decades, and that delivers its functionality
reliably throughout.

Nevertheless, the chapter may appear surprisingly long when we reduce
its content to the two fundamental concepts: Model-view separation enables ↰9.1 ↰9.2.2
testing to make the functionality reliable, and it liberates that functionality
from concerns about the ever-changing user interface. Undoable operations ↰9.5
are simply encapsulated as command objects, which are managed by a
command processor.

The challenge in this chapter is not the concepts, but their faithful
rendering in concrete software: It is just too simple to destroy the principles
and all their benefits by seemingly minor glitches in the implementation.
After all, cannot a single reference to a view be tolerated in the model if it ↰9.2.3
saves days of coding? Does it really matter if an observer interface is tailored
to the user interface that we have to deliver in three days' time? Cannot ↰9.4.3
repainting of changed data be much more efficient if the view remembers
the changed data elements?

Professional developers know two things that will prevent them from
falling into such traps. First, the extra effort of introducing model-view ↰9.4 ↰9.5.3
separation is rather extensive, but it is also predictable. Going through a
series of well-known and well-rehearsed steps is psychologically less taxing
than grappling with an endless list of poorly understood details. Second,
they know the motivation behind all of those steps and see the necessity and
implications of each. As a result, they perceive a proper overall structure
as a necessary investment in achieving their future goals more easily. Both
points together—the following of known steps and the understanding of
implications—also enable professionals to be sure of their details, such as
when recreating the previous and subsequent states in a command's `undo()` ↰9.5.1
and `redo()` methods very precisely, if necessary by reasoning in detail about ↰4.1 ↰4.2.2
the contracts of invoked model operations. This chapter has introduced the
landmarks for proceeding with forethought in this way.

This page intentionally left blank

Index