APPENDIX A

Design Patterns

Design patterns are partial solutions to common problems, such as separating an interface from a number of alternate implementations, wrapping around a set of legacy classes, protecting a caller from changes associated with specific platforms. A design pattern is composed of a small number of classes that, through delegation and inheritance, provide a robust and modifiable solution. These classes can be adapted and refined for the specific system under construction. In addition, design patterns provide examples of inheritance and delegation.

Since the publication of the first book on design patterns for software [Gamma et al., 1994], many additional patterns have been proposed for a broad variety of problems, including analysis [Fowler, 1997] [Larman, 1998], system design [Buschmann et al., 1996], middleware [Mowbray & Malveau, 1997], process modeling [Ambler, 1998], dependency management [Feiler et al., 1998], and configuration management [Brown et al., 1999]. The term itself has become a buzzword that is often attributed many different definitions. In this book, we focus only on the original catalog of design patterns, as it provides a concise set of elegant solutions to many common problems. This appendix summarizes the design patterns we use in this books. For each pattern, we provide pointers to the examples in the book that use them. Our goal is to provide a quick reference that can also be used as an index. We assume from the reader a basic knowledge of design patterns, object-oriented concepts, and UML class diagrams.

A.1 Abstract Factory: Encapsulating platforms

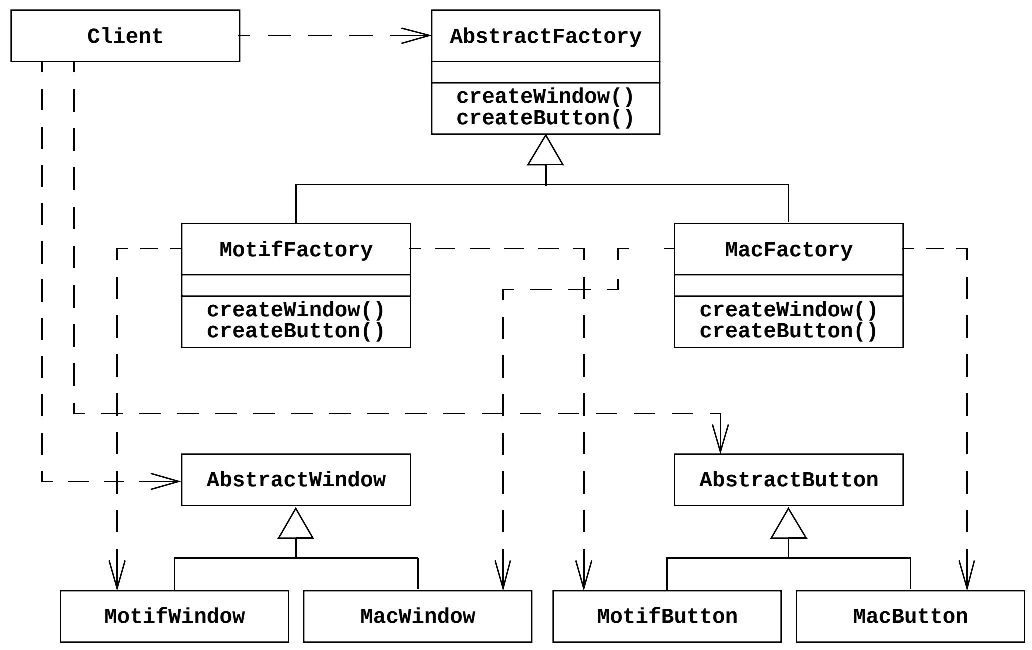

Figure A-1 `AbstractFactory` design pattern (UML class diagram).

Purpose This pattern is used to shield an application from the concrete classes provided by a specific platform, such as a windowing style or an operating system. Consequently, by using this pattern, an application can be developed to run uniformly on a range of platforms.

Description Each platform (e.g., a windowing style) is represented by a `Factory` class and a number of `ConcreteClass`es for each concept in the platform (e.g., window, button, dialog). The `Factory` class provides methods for creating instances of the `ConcreteClass`es. Porting an application to a new platform is then reduced to implementing a `Factory` and a `ConcreteClass` for each concept.

Examples

- statically encapsulating windowing styles (Figure 7-28 on page 267)
- dynamically encapsulating windowing styles (Swing, [JFC, 1999])

Related concepts Increasing reuse (Section 7.4.9 on page 265), removing implementation dependencies (Section 7.4.10 on page 267).

A.2 Adapter: Wrapping around legacy code

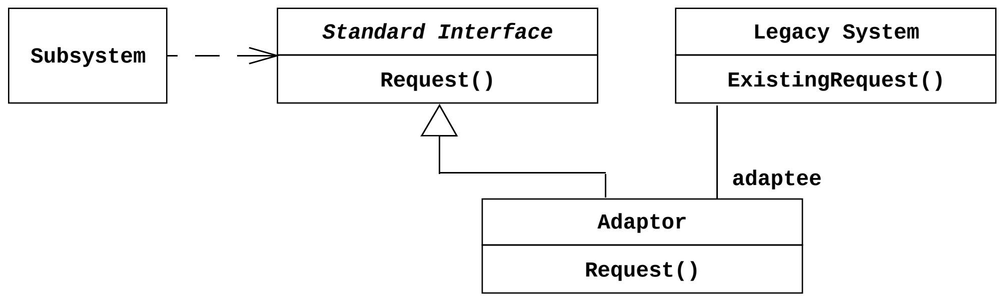

Figure A-2 `Adapter` pattern (UML class diagram). The `Adapter` pattern is used to provide a different interface (*`New Interface`*) to an existing component (`Legacy System`).

Purpose This pattern encapsulates a piece of legacy code that was not designed to work with the system. It also limits the impact of substituting the piece of legacy code for a different component.

Description Assume a `Calling Subsystem` needs to access functionality provided by an existing `Legacy System`. However, the `Legacy System` does not comply with a *`Standard Interface`* used by the `Calling Subsystem`. This gap is filled by creating an Adaptor class which implements the *`Standard Interface`* using methods from the `Legacy System`. When the caller only accesses the *`Standard Interface`*, the `Legacy System` can be later replaced with an alternate component.

Example

- Sorting instances of an existing `String` class with an existing `sort()` method (Figure 6-34 on page 202): `MyStringComparator` is an `Adaptor` for bridging the gap between the `String` class and the `Comparator` interface used by the `Array.sort()` method.

Related concepts The `Bridge` (Section A.3) fills the gap between an interface and its implementations.

A.3 Bridge: Allowing for alternate implementations

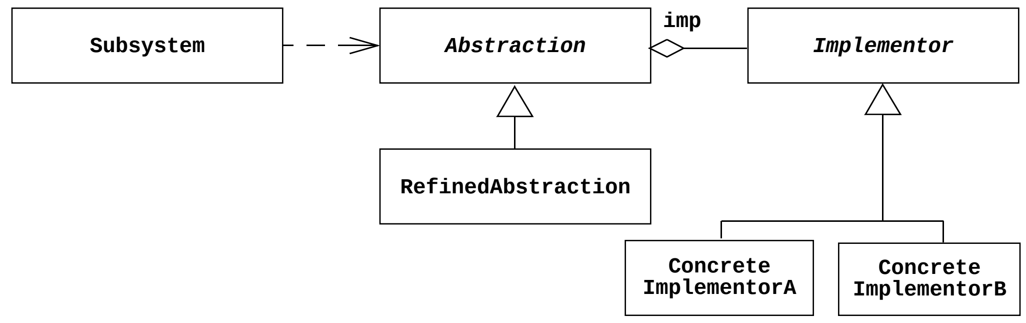

Figure A-3 `Bridge` pattern (UML class diagram).

Purpose This pattern decouples the interface of a class from its implementation. Unlike the `Adapter` pattern, the developer is not constrained by an existing piece of code. Both the interface and the implementation can be refined independently.

Description Assume we need to decouple an *`Abstraction`* from an *`Implementor`*, because we need to substitute different *`Implementors`* for a given *`Abstraction`* without any impact on a calling `Subsystem`. This is realized by providing an *`Abstraction`* class that implements its services in terms of the methods of an *`Implementor`* interface. `Concrete Implementors` that need to be substituted refine the *`Implementor`* interface.

Examples

- Vendor independence (Figure 6-37 on page 206): The `ODBC` interface (the `Abstraction`) decouples a caller from a database management system. For each database management system, an `ODBC Driver` refines the `ODBC Implementation` (the *`Implementor`*). When the *`Abstraction`* makes no assumptions about the `Concrete Implementors`, `Concrete Implementors` can be switched without the calling `Subsystem` noticing, even at run-time.
- Unit testing (Figure 9-11 on page 342): the `Database Interface` (the `Abstraction`) decouples the `User Interface` (the `Subsystem`) from the `Database` (a `Concrete Implementor`), allowing the `User Interface` and the `Database` to be tested independently. When the `User Interface` is tested, a `Test Stub` (another `Concrete Implementor`) is substituted for the `Database`

Related concepts The `Adapter` pattern (Section A.2) fills the gap between two interfaces.

A.4 Command: Encapsulating control

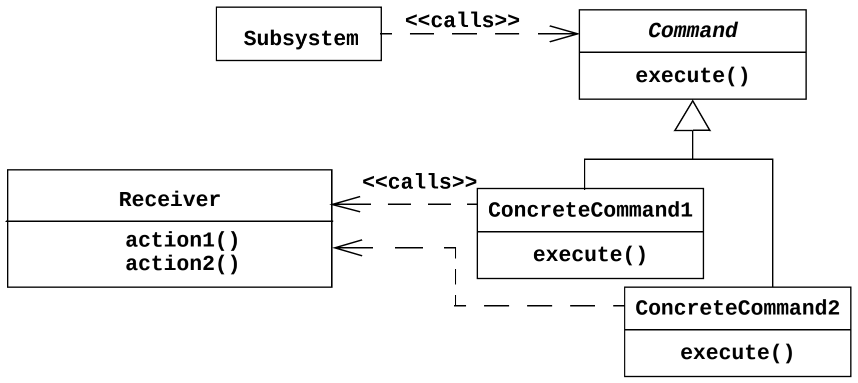

Figure A-4 Command pattern (UML class diagram).

Purpose This pattern enables the encapsulation of control such that user requests can be treated uniformly, independent of the specific request. This pattern protects these objects from changes resulting from new functionality. Another advantage of this pattern is that control flow is centralized in the command objects as opposed to being distributed across interface objects.

Description An abstract *`Command`* interface defines the common services that all `ConcreteCommand`s should implement. `ConcreteCommand`s collect data from the Client `Subsystem` and manipulate the entity objects (`Receiver`s). `Subsystem`s interested only in the general *`Command`* abstraction (e.g., an undo stack), only access the *`Command`* abstract class. Client `Subsystem`s do not access directly the entity objects.

Examples

- Providing an undo stack for user commands: All user-visible commands are refinements of the *`Command`* abstract class. Each command is required to implement the `do()`, `undo()`, and `redo()` methods. Once a command is executed, it is pushed onto an undo stack. If the user wishes to undo the last command, the Command object on the top of the stack is sent the message `undo()`.
- Decoupling interface objects from control objects (Figure 6-45 on page 215, see also Swing `Action`s, [JFC, 1999]): All user visible commands are refinements of the *`Command`* abstract class. Interface objects, such as menu items and buttons, create and send messages to *`Command`* objects. Only *`Command`* objects modify entity objects. When the user interface is changed (e.g., a menu bar is replaced by a tool bar), only the interface objects are modified.

Related concepts MVC architecture (Figure 6-15 on page 184).

A.5 Composite: Representing recursive hierarchies

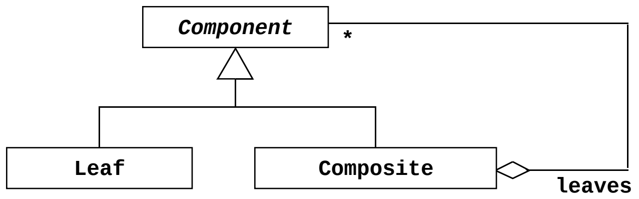

Figure A-5 `Composite` pattern (UML class diagram).

Purpose This pattern represents a recursive hierarchy. The services related to the containment hierarchy are factored out using inheritance, allowing a system to treat a leaf or a composite uniformly. Leaf specific behavior can be modified without any impact on the containing hierarchy.

Description The `Component` interface specifies the services that are shared among `Leaf` and `Composite` (e.g., `move(x,y)` for a graphic element). A `Composite` has an aggregation association with *`Component`*s and implements each service by iterating over each contained *`Component`* (e.g., the `Composite.move(x,y)` method iteratively invokes the `Component.move(x,y)`). The `Leaf` services do the actual work (e.g., `Leaf.move(x,y)` modifies the coordinates of the `Leaf` and redraws it).

Examples

- Recursive access groups (Lotus Notes): A Lotus Notes access group can contain any number of users and access groups.
- Groups of drawable elements: Drawable elements can be organized in groups that can be moved and scaled uniformly. Groups can also contain other groups.
- Hierarchy of files and directories (Figure 5-7 on page 137): Directories can contain files and other directories. The same operations are available for moving, renaming, and uniformly removing files and directories.
- Describing subsystem decomposition (Figure 6-3 on page 173): We use a `Composite` pattern to describe subsystem decomposition. A subsystem is composed of classes and other subsystems. Note that subsystems are not actually implemented as `Composite`s to which classes are dynamically added.
- Describing hierarchies of tasks (Figure 6-8 on page 177): We use a `Composite` pattern to describe the organizations of `Tasks` (`Composites`) into `Subtask`s (*`Component`*s) and `ActionItem`s (`Leaves`). We use a similar model to describe `Phase`s, `Activities`, and `Task`s (Figure 12-6 on page 462).

Related concepts `Facade` pattern (Section A.6).

A.6 Facade: Encapsulating subsystems

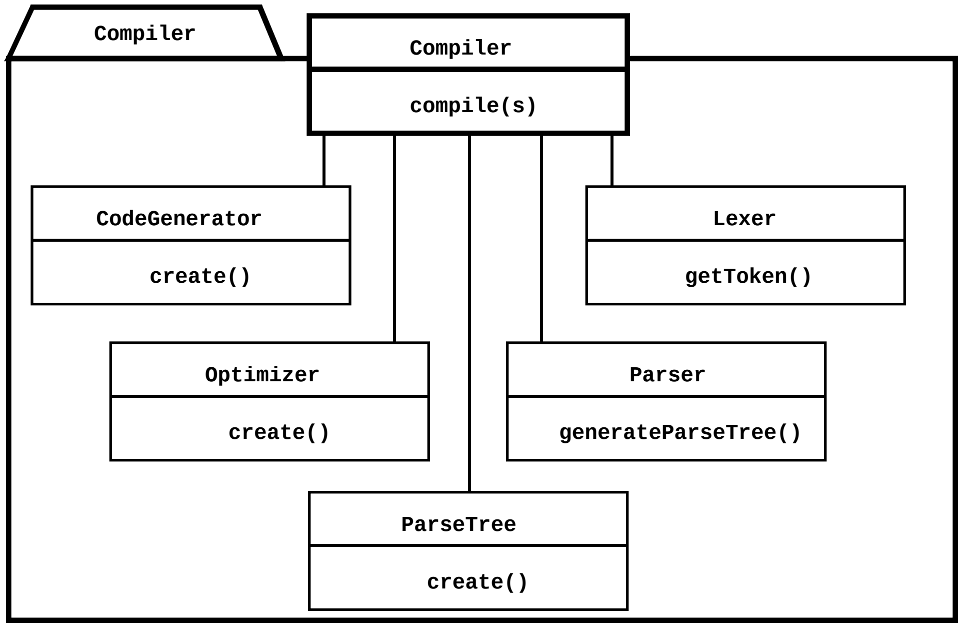

Figure A-6 An example of `Facade` pattern (UML class diagram).

Purpose The `Facade` pattern reduces dependencies among classes by encapsulating a subsystem with a simple unified interface.

Description A single Facade class implements a high-level interface for a subsystem by invoking the methods of lower level classes. A Facade is opaque in the sense that a caller does not access the lower level classes directly. The use of Facade patterns recursively yields a layered system.

Example

- Subsystem encapsulation (Figure 6-30 on page 198): A `Compiler` is composed of `Lexer`, `Parser`, `ParseTree`, a `CodeGenerator`, and an `Optimizer`. When compiling a string into executable code, however, a caller only deals with the `Compiler` class, which invokes the appropriate methods on the contained classes.

Related concepts Coupling and coherence (Section 6.3.3 on page 174), layers and partitions (Section 6.3.4 on page 178), `Composite` pattern (Section A.5).

A.7 Observer: Decoupling entities from views

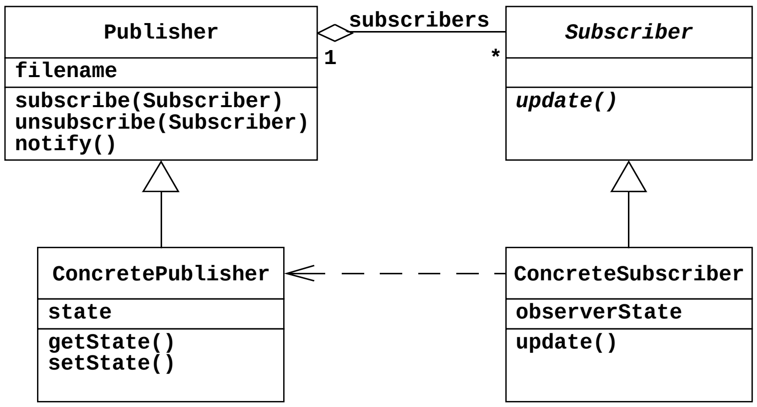

Figure A-7 The `Observer` pattern (UML class diagram).

Purpose This pattern allows to maintain consistency across the states of one `Publisher` and many *`Subscriber`*s.

Description A `Publisher` (called a `Subject` in [Gamma et al., 1994]) is an object whose primary function is to maintain some state; for example, a matrix. One or more *`Subscriber`*s (called *`Observer`*s in [Gamma et al., 1994]) use the state maintained by a `Publisher`; for example, to display a matrix as a table or a graph. This introduces redundancies between the state of the `Publisher` and the *`Subscriber`*s. To address this issue, *`Subscriber`*s invoke the `subscribe()` method to register with a `Publisher`. Each `ConcreteSubscriber` also defines an `update()` method to synchronize the state between the `Publisher` and the `ConcreteSusbcriber`. Whenever the state of the `Publisher` changes, the `Publisher` invokes its `notify()` method, which iteratively invoke each `Subscriber.update()` method.

Examples

- The `Observer` interface and `Observable` class are used in Java to realize an `Observer` pattern ([JFC, 1999]).
- The Observer pattern can be used for realizing subscription and notification in an Model/ View/Controller architecture (Figure 6-15 on page 184).

Related concepts Entity, interface, control objects (Section 5.3.1 on page 134).

A.8 Proxy: Encapsulating expensive objects

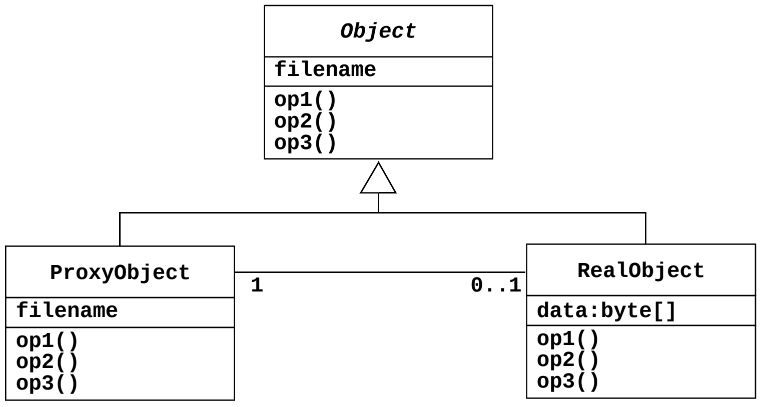

Figure A-8 The `Proxy` pattern (UML class diagram).

Purpose This pattern improves the performance or the security of a system by delaying expensive computations, using memory only when needed or checking access before loading an object into memory.

Description The `ProxyObject` class acts on behalf of a `RealObject` class. Both classes implement the same interface. The `ProxyObject` stores a subset of the attributes of the `RealObject`. The `ProxyObject` handles certain requests completely (e.g., determining the size of an image), whereas others are delegates to the `RealObject`. After delegation, the `RealObject` is created and loaded in memory.

Examples

- Protection proxy (Section 6-38 on page 210): An `Access` association class contains a set of operations that a `Broker` can use to access a `Portfolio`. Every operation in the `PortfolioProxy` first checks with `isAccessible()` if the invoking `Broker` has legitimate access. Once access has been granted, `PortfolioProxy` delegates the operation to the actual `Portfolio` object. If access is denied, the actual `Portfolio` object is not loaded into memory.
- Storage proxy (Section 7-31 on page 273): An `ImageProxy` object acts on behalf of an `Image` stored on disk. The `ImageProxy` contains the same information as the `Image` (e.g., width, height, position, resolution) except for the `Image` contents. The `ImageProxy` services all contents independent requests. Only when the Image contents need to be accessed (e.g., when it is drawn on the screen), the `ImageProxy` creates the `RealImage` object and loads its contents from disk.

Related concepts Caching expensive computations (Section 7.4.13 on page 272).

A.9 Strategy: Encapsulating algorithms

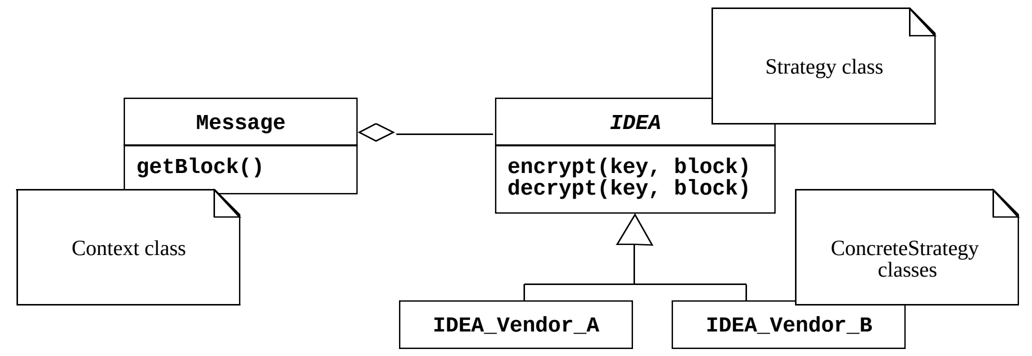

Figure A-9 An example of `Strategy` pattern encapsulating multiple implementation of the IDEA encryption algorithm (UML class diagram). The `Message` and *`IDEA`* classes cooperate to realize the encryption of plain text. The selection of an implementation can be done dynamically.

Purpose This pattern decouples an algorithm from its implementation(s). It serves the same purpose than the adapter and bridge patterns except that the encapsulated unit is a behavior.

Description An abstract `Algorithm` class provides methods for initializing, executing, and obtaining the results of an `Algorithm ConcreteAlgorithm` classes refine `Algorithm` and provide alternate implementations of the same behavior. `ConcreteAlgorithms` can be switched without any impact on the caller.

Examples

- Encryption algorithms (Figure 6-40 on page 212): Vendor-supplied encryption algorithms pose an interesting problem: How can we be sure that the supplied software does not include a trap door? Moreover, once a vulnerability is found in a widely used package, how do we protect the system until a patch is available? To address both issues, we can use redundant implementations of the same algorithm. To reduce the dependency on a specific vendor, we encapsulate these implementations with a single `Strategy` pattern.

Related concepts Adapter pattern (Section A.2) and Bridge pattern (Section A.3).

References

[Ambler, 1998]	S. W. Ambler, *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University, Cambridge, 1998.
[Brown et al., 1999]	W. J. Brown, H. W. McCormick, & S. W. Thomas, *AntiPatterns and Patterns in Software Configuration Management*, Wiley, New York, 1999.
[Buschmann et al., 1996]	F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, & M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, U.K., 1996.
[Feiler et al., 1998]	P. Feiler & W. Tichy. “Propagator: A family of patterns,” in the *Proceedings of TOOLS-23'97*, Jul. 28–Aug. 1 1997, Santa Barbara, CA.
[Fowler, 1997]	M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, MA, 1997.
[Gamma et al., 1994]	E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design Patters: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1994.
[JFC, 1999]	*Java Foundation Classes*, JDK Documentation. Javasoft, 1999.
[Larman, 1998]	C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, Upper Saddle River, NJ, 1998.
[Mowbray & Malveau, 1997]	T. J. Mowbray & R. C. Malveau. *CORBA Design Patterns*. Wiley, New York, 1997.