9. Architectural Design

The architectural design begs for the analogy from the building industry. A house cannot be built unless an architect designed it (perhaps a shed – yes, but not a house). Similarly, any reasonably large software system cannot be built without a prior architectural design. Software architecture is a foundation on which all other design and programming solutions must be based.

Architecture of the system should be addressed early in the process. Booch et al. (1999) consider architecture as one of only three main characteristics of any development process favoring the UML. The three characteristics of the process are:

- iterative and incremental,
- use case driven, and
- architecture-centric.

What is software architecture? The definitions are abound and multifaceted. A succinct definition may be that software architecture is the organization of software elements into a system aiming at achieving some purpose. More detailed investigation reveals that software architecture addresses a large number of concerns, such as: organization of specific software modules (classes, packages, components), interconnections between modules, assignment of behaviors to modules, scalability of modules to larger solutions, etc.

What is architectural design? Again, definitions vary but the intent is clear. Architectural design is the set of decisions aiming at efficient and effective software architecture together with the rationale for these decisions. The rationale emphasized in this book is the understandability, maintainability and scalability of the system. Larman (2002) identifies four concerns of architectural design, namely that it:

- relates to non-functional requirements (ref. supplementary specification in Section 8-4),
- involves large-scale, system-level fundamental decisions,
- tackles interdependencies and trade-offs, and
- provides for generation and evaluation of alternative solutions.

9.1 Architectural Layers and Dependency Management

Like all software production, architectural design is a continuing, iterative and incremental, effort. Early architectural decisions take a broad view on the software architecture. One of the first decisions to be taken relates to structuring the system into layers of modules and establishing principles of inter-module
communication. This is the concern of this chapter. More detailed architectural solutions, such as intra-module communication, are discussed in relevant places later in the book.

A sound architectural design calls for:

- a hierarchical layering of software modules that reduces complexity and enhances understandability of module dependencies by disallowing direct object intercommunication between non-neighboring layers, and
- an enforcement of programming standards that make module dependencies visible in compile-time program structures and that forbid muddy programming solutions utilizing just run-time program structures.

These observations are particularly, and painfully, true for modern object-oriented software production. The object paradigm equips a software engineer with a multitude of very powerful programming abstractions, which – when used unwisely – result in programs impossible to understand and maintain, even by programmers who wrote them.

### 9.1.1 Architectural Modules

Architectural design is an exercise in managing module dependencies. Module A depends on module B if changes to module B may necessitate changes to module A. It is important that dependencies do not cross dependency firewalls (Martin, 2003). In particular, dependencies should not propagate across non-neighboring layers and must not create cycles.

Architectural design takes a proactive approach to managing dependencies in software. It does so by deciding early in the process on hierarchical layers of software modules and on dependency firewalls between the modules. This is a forward-engineering approach – from design to implementation. The aim is to deliver a software design that minimizes dependencies by imposing an architectural solution on programmers.

Eventually, the proactive approach to managing dependencies must be supported by the reactive approach that aims at measuring dependencies in implemented software. This is a reverse-engineering approach – from implementation to design. The implementation may or may not conform to the desired architectural design. If it does not, the aim is to compare the metric values in the software with the values that the desired architecture would have delivered. The troublesome dependencies must be pinpointed and addressed.

### 9.1.1.1 Design Classes

Previous chapters in Part B of this book concentrated on modeling “business objects” classified as business entities (Chapter 6), domain classes (Chapter 7) and conceptual classes (Chapter 8). But “business objects” are but one set of classes in an object-oriented program.

A typical program needs classes responsible for presenting information on the user’s computer screen, classes to access the database, classes to perform algorithmic calculations, etc. There are different names used to signify the entire set of classes necessary to be designed and implemented in a computer program. They are called interchangeably design classes, software classes, application classes, program classes, system classes or implementation classes. The term used here is design classes or simply classes, but other terms may be more suitable elsewhere in the book (design classes is also the term favored by the UP).
UML defines a class as a description for “a set of objects that share the same specifications of features, constraints, and semantics” (UML, 2002, p.45). Features are attributes and operations. An attribute is a structural (typed) feature of a class. An operation is a behavioral feature of a class that declares a service that can be performed by instances (objects) of that class.

In programming languages, such as Java, features are referred to as class members. An attribute is a data member, frequently called a member variable or a field (note that a local variable defined within an operation is not a data member). An operation is a member function, typically called a method. An act of calling a service provided by a method is referred to as sending a message to an object or a message passing between objects.

### 9.1.1.2 Packages

Design classes are grouped into packages according to an architectural framework adopted for the development project. A package (UML, 2002) is a grouping of modeling elements under an assigned name. A package may contain other packages. Packages can be grouped and structured into hierarchical layers supportive of the chosen software architecture.

In the UML, package is a logical design concept. Eventually, packages must be implemented and mapped to programming language concepts. Modern languages, most notably Java and C#, provide for a direct mapping that uses the notion of the package on the implementation end. The support for the implementation package is in the form of a namespace for classes and for importing other packages.

A package owns its members (elements) – removing the package from the model removes also its members. It follows, that a member (usually a class) can belong to one package only.

A package may have package imports to other packages. This means that package A or element of package A can refer to package B or to its elements. Consequently, a class is owned by only one package but it can be imported to other packages. Imports introduce dependencies between packages and their elements.

Figure 9-1 presents example of the UML package notation. Package can be presented with no members (elements) revealed (Package A). Package can depend on another package (Package A depends on Package B).
The dependency relationship means that some members of Package A refer in some way to some members of Package B (this can mean that Package A imports some elements of Package B). The implications are twofold:

- changes to Package B may affect Package A, normally leading to the need to recompile and re-test Package A,
- Package A can only be used (reused) together with Package B.

Package elements can be revealed by including their graphical representations within the borders of the owner package or by using the cross-plus notation. In Figure 9-1, Package B owns Class X, Package C owns Package D, Package E owns Package F, and Package F owns Class Y and Class Z.

### 9.1.2 Package Dependencies

Objects must intercommunicate for a system to perform its tasks. Objects depend on each other for services. Dependency management does not mean that dependencies as such are a problem. It means that dependencies should be minimized and unnecessary dependencies should be eliminated from architectural design.

Particularly troublesome are circular dependencies between objects. Fortunately, most of the time circular packages can be avoided or made relatively harmless through careful refactoring (re-design) or through programming techniques (Martin, 2003). Figure 9-2 shows two examples of circular dependencies between packages.
In Figure 9-2, Package A depends on Package B and vice versa. This means that a dependency-related change in Package B will demand a change in Package A, which may in turn demand a change in Package B, etc. The final outcome is that the two packages are inseparable with regard to program’s understanding, maintainability and scalability.

Similarly, a change in Package E may demand a change in Package D, and then in Package C. The change in Package C may in turn demand a change in Package E, etc. The path is longer, but the problem is the same.

Adding a new package, as shown in Figure 9-3 can break circular dependencies between packages (Figure 9-2). In Figure 9-3, elements in Package A, on which Package B depended, were factored out into their own Package A2. Package B does not depend any more on Package A – it depends on Package A2. Understandably enough, Package A depends now on Package A2. Similarly in the second example, elements in Package C, on which Package E depended, were factored out into their own Package C2.
The horizontal structures of packages as in Figure 9-3 are called partitions. When circular dependencies between partition packages are broken, by adding new packages as explained in Section 9.1.2, the dependency structure within a partition becomes a hierarchy (rather than a linear horizontal structure).

### 9.1.3 Layer Dependencies

As stated before, packages can be grouped and structured into hierarchical layers supportive of the chosen software architecture. Since a package may contain other packages, a layer is a package itself. In the UML a layer package can be stereotyped as «layer».

From the architectural design perspective, layers are vertical structures (Figure 9-4). Vertical layers consist of partitions of packages (Section 9.1.2). Superimposing vertical structures of layers on horizontal structures of partitions creates a hierarchy of package dependencies. Three critical objectives of good architectural design of layers are that:

- the layer hierarchy does not disintegrate to a network structure (where a package can potentially depend on any other package in the system),
- the layer hierarchy minimizes dependencies between packages,
- the layer hierarchy establishes a stable framework for the lifecycle of system development.

The first objective is intuitively obvious. The complexity of networks grows exponentially with addition of new elements to the structure. In practice, all complex structures that work, including living organisms and human-made systems, are hierarchies.
The second objective states that the layer hierarchy should minimize dependencies between packages. The widely-accepted method of achieving this is by making higher layers depend on lower layers but not vice versa. Unfortunately, the top-down only dependency structure is not quite realistic. In reality, the bottom-up dependencies will exist, but they can be made relatively harmless by skilful design and programming. A desired outcome is that higher layers depend on lower layers while lower layers can still communicate with higher layers without exerting undue (unmanageable) dependencies.

The third objective is that the layer hierarchy is a stable framework. Stability means that something is resistant to change and steadfast of purpose. A stable architectural framework, once designed, is not receptive of changes. A stable framework determines the fixed set of rules for “the game” of software development. Within these rules, the moves of the “game” are flexible.

Note that the stability of vertical layers increases in the top-down direction. Higher layers depend on lower layers. Lower layers are required to be stable because any changes to them may have a ripple effect on higher layers (Martin, 2003).

Layer 2 in Figure 9-4 is stable and Layer 1 is unstable. Layer 1 depends on Layer 2. Layer 2 is independent and can therefore be replaced by a new one without a ripple-effect on the rest of the system. This is the principle and the reason behind allowing a high dependency (high coupling) in the top-down direction and ensuring a low dependency (low coupling) in the bottom-up direction.

Note that the stability condition of layers means that the technique of eliminating circular dependencies between layers by adding new layers is not acceptable. Fortunately, there exist programming techniques to ensure that circular dependencies between layers can be eliminated or made relatively harmless.

### 9.1.4 Class Dependencies

The programming techniques that allow eliminating or incapacitating circular dependencies between layers (and packages at large) have to do with the architectural design of classes rather than packages. The point is that:

- dependencies between layers translate to (are caused by) dependencies between packages,
- dependencies between packages translate to dependencies between classes,
- dependencies between classes translate to dependencies between class members, predominantly between methods.

In Figure 9-5, Layer 1 depends on Layer 2 because there is some class in Layer 1 that depends on a class in Layer 2. Package A depends on Package B because there is some class in Package A that depends on a class in Package B. The consequence is that if offending class dependencies (i.e. class dependencies that introduce cycles) can be eliminated or made harmless then the overall software architecture of layers and packages can be so much more stable.
The main programming technique to break cycles between classes, and therefore contribute to the elimination of cycles between packages and between layers, involves the use of the \textit{interface} concept. A supporting technique is a targeted use of \textit{event processing}, possibly on top of the use of interfaces. The two techniques are explained in Sections 9.1.6 and 9.1.7, after the dependencies between methods are described.

### 9.1.5 Method Dependencies

Dependencies between classes translate to dependencies between class members (features, in the UML parlance). Usually, dependencies between data members can be managed relatively easily (although implementation inheritance of data members may blur the picture). Dependencies between methods create a real challenge, in particular that in practice many method dependencies cannot be tracked down by just analyzing the static compile-time structure of the program.

Figure 9-6 illustrates method dependencies and how they propagate to class dependencies (and, therefore, also to package and layer dependencies). There are two packages called \textit{control} and \textit{entity} (the names not capitalized to follow the usual practice). Class names start with a capital letter signifying the immediate package to which the class belongs. Hence, \texttt{CActioner} denotes that the class is owned by the \texttt{control} package.
CActioner uses method `do1()` to send message `do3()` to EEmployee. Therefore, `do1()` depends on `do3()`. The dependency propagates up on owning classes and packages. CActioner depends on EEmployee, and control depends on entity. Similarly, `do2()` in EOutMessage invokes method `do3()` on EEmployee. Hence, EOutMessage depends on EEmployee.

Note that method dependencies are made explicit in the model on the level of classes by establishing \textit{unidirectional associations} between dependent classes. Both CActioner and EOutMessage have data members `emp` of type EEmployee, which implemented unidirectional associations to EEmployee, called `ActEmp` and `OutEmp`, respectively.

Making method dependencies statically visible in the code by means of explicit associations between classes is a strongly recommended practice (Lee and Tepfenhart, 2002). This is important because method dependencies are frequently not discoverable from the analysis of the source code. In the presence of inheritance and polymorphism and because of the demands of the layered architecture, a message originator (\textit{client object}) frequently does not know the specific receiver of the message (\textit{supplier object}) until runtime.

If the analysis of the program behavior discovers a method dependency between classes with no explicit association then the program may be considered to be in violation of the architectural design. However, in some cases, creating an association to legitimize method dependency may itself be in violation of the architectural design. The introduction of interfaces (Section 9.1.6) and event processing (Section 9.1.7) into the design can solve this dilemma. More complex runtime method dependencies (called \textit{acquaintance dependencies}) may require even more radical design decisions as discussed in Section 9.1.8.
9.1.5.1 Method Dependencies in Presence of Delegation

Message passing is realized as a *synchronous communication* between the client and supplier of a service. A message from a *client object* requests that a *supplier object* performs a service (method). The interpretation of a message and the means of executing it are at the discretion of the supplier object. This could be the *delegation* of the work to another object.

Like in a military chain of command, an object can delegate the authority to perform the work to another object. What is perceived by the client object to be the supplier object is in reality a *delegator object*. Although the work is delegated, the supplier object – alias the delegator object – is not relieved from a contractual responsibility (to the client object) for the work.

The delegation is normally necessary to allow a client object in one layer to get a service from an object in a distant (non-neighboring) layer. Otherwise, the stable framework of vertical hierarchical layers (Section 9.1.3) would disintegrate to a random network of intercommunicating objects with no hope to understand or control the system complexity and evolution.

Figure 9-7 demonstrates method dependencies in the presence of delegation when the layer framework consists of packages named control, entity, mediator, and foundation. For clarity, the unidirectional associations to signify message passing are not drawn (but they are programmed in the Java code presented in the UML notes).
In Figure 9-7, CActioner and EOutMessage request a do3() service from EEmployee. However, EEmployee delegates the execution of the service to MBroker and MBroker delegates it further down to FUpdater. FUpdater performs the service. This is communicated back to the client objects following the same path but upwards. Note that the delegation is frequently necessary to conform to the vertical layer architecture that disallows direct communication between non-neighboring layers.

The consequence of delegation is that a client might not know its real supplier (and it might not even care to know as long as the “goods” are supplied). Unlike in Figure 9-7, the knowledge of the real supplier may not be available from a static analysis of the program code and be hidden behind the dynamicity of inheritance (in particular interface inheritance) and polymorphism.

As the discovery of the program’s runtime behavior is “not much fun” for a system maintainer or a project manager, the practice of explicit associations between classes engaged in message passing is so much more important. Without explicit associations, the impact analysis due to a change in the supplier code may be unachievable.

9.1.6 Interfaces

In the UML 2.0, interface is a declaration of a set of features (Section 9.1.11) that is not directly instantiable, i.e. no objects of it can be directly created. An interface is realized (implemented) by an object of a class, which promises to deliver the structure and behavior of the interface to the rest of the program. The object that implements interface provides “a public façade that conforms to the interface specification” (UML, 2002, p.123).

The UML 2.0 interface concept expands the notion of interface used in popular programming languages (and in earlier versions of the UML). An interface can declare attributes, not just operations. By contrast, in Java an interface can contain data members but they must be constants (defined as static and final).

As a consequence of allowing attributes in interfaces, it is possible to create associations between interfaces and between an interface and a class. Attributes typed as another interface or class represent associations. In the UML 2.0 it is possible to navigate from an interface to a class via an association. This is not possible in Java.

Interface is sometimes said to be a “pure” abstract class. Not quite (although that’s the best one can do in languages that do not support interfaces, such as C++). An abstract class is a class that contains at least one method, which is not (or cannot be) implemented by that class, and therefore it cannot be instantiated. In a pure abstract class all methods are not implemented.

Pure or not, a class is a class is a class. In languages that support only single implementation inheritance, like Java, a class can only extend one base class (abstract or concrete), but it can implement multiple interfaces. This is a huge practical difference.

The related difference is that interfaces allow passing objects typed as interfaces in method calls. The exact class of that interface does not need to be determined until runtime. For the client object it does not matter what the supplier class is. It only matters that it implements the method called (Lee and Tepfenhart, 2002).
When used as the main entry points to concrete classes in packages, interfaces in particular, but also abstract classes, provide mechanisms to hide internal complexity of the package and permitting package extensions without affecting client objects in other packages. The notion of a dominant class can be used to realize these mechanisms within the package. A dominant class implements main interfaces and abstract classes of a package. In words of Rumbaugh et al. (1999, p.219) – “A dominant class subsumes the interface of the component”.

Figure 9-8 shows the UML notations for an interface, an abstract class (italic font) and a class. The interface notation differ in the way the stereotype «Interface» is displayed. It can be displayed, from left to right, as an explicit label, an icon or a “decoration”.

The UML 2.0 introduced another iconic variation where a circle (called a ball) can be supplemented by a half-circle (called a socket). However, both the ball and the socket notation in particular, do not seem to scale up gracefully to larger models. Therefore, the icons will be avoided in this book in favor of the decoration notation (not really directly advised in the UML 2.0, but widely used in practice).

### 9.1.6.1 Implementation Dependency

More than one class may implement an interface and a class may implement more than one interface. The set of interfaces implemented by a class is called its provided interface. This is the class’ promise to its clients that its instances will provide the interface features. Provided interfaces are specified in the UML 2.0 by a dependency relationship between the class and the interface implemented by that class. This is called an implementation dependency.

Figure 9-9 shows the UML notation for implementation dependency. A class at the tail of the arrow implements the interface at the head of the arrow (note that the arrows are shown here on dotted lines, but the UML 2.0 uses solid lines). Class1 implements both interfaces. Class2 implements only Interface2.
Classes provide implementation (the code) for all operations of their interfaces. In case of attributes, the class promises that any of its instances will maintain information about the type and multiplicity of the attribute and that it will deliver that attribute to any client object, but the class itself does not have to have that attribute in its implementation (UML, 2002).

### 9.1.6.2 Usage Dependency

Once declared, interfaces can be used by classes (or other interfaces) that require them. It is said that a class (or an interface) uses **required interfaces**. Required interfaces specify services that a class (or an interface) needs so that it can perform its own services to its clients. Required interfaces are specified in the UML 2.0 by a dependency relationship between the class (or the interface) and the interface that is required. This is called a **usage dependency**.

Figure 9-10 shows the UML notation for usage dependency. A class or interface at the tail of the arrow uses the interface at the head of the arrow (the use of the stereotype «uses» is not compulsory in the UML 2.0, but it is customary). Class1 uses Interface1, which in turn uses Interface2.
Class1 contains method do1(), which calls services of operation o1(). In the static code it is not clear which implementation of the required interface will do the service. This can be an instance of any class, which implements Interface1. The exact instance will be resolved at runtime when an executing instance of Class1 sets the value of data member myInterface to refer to a concrete object of a concrete class.

### 9.1.6.3 Breaking Circular Dependencies with Interfaces

Interfaces can be used with a great success to reduce dependencies in the code. Programming with interfaces allows client objects to be unaware of the specific classes of objects they use and of the classes that actually implement these objects. One of the most important principles of reusable, maintainable and scalable object-oriented design is: “Program to an interface, not an implementation” (Gamma et al., 1995, p.18).

Previous discussion made it clear that circular dependencies in the system must be eliminated at any cost. It turns out that interfaces introduce the most powerful weapon in dealing with circular dependencies (Martin, 2003). To understand why this is the case, consider an example of circular dependency between the presentation and control packages in Figure 9-11.
The cycle in Figure 9-11 is caused by CInit using the services of PPrimaryWindow and by PDialogBox using the services of CActioner. Two explicit unidirectional associations ascertain the message passing between these classes. The cycle is particularly troublesome because it is between «layer» packages and this harms the principle of the top-down dependencies between layers discussed in Section 9.1.3.

A solution to the dilemma comes from a skillful introduction of an interface into the design (Martin, 2003), as shown in Figure 9-12. In the example, the cycle is broken by adding interface ICPresenter in the control package. The interface defines method do2() needed by CInit, which is in the same package. But the interface is actually implemented by PPrimaryWindow in the presentation package. CInit uses the interface and PPrimaryWindow implements it.
9. ARCHITECTURAL DESIGN

Note that to break the cycle, the interface and the class that implements it are in different packages. This may be a surprise to a novice programmer, but it should not be a revelation to a designer responsible for the architectural design of the system. It is frequently more desirable to put interfaces in the package that uses them, not in the package that implements them (Martin, 2003). Fowler calls this the Separated Interface pattern (Fowler, 2003). Alternatively, interfaces may be extracted into a separate package.

9.1.7 Event Processing

Method dependencies (Sections 9.1.5) apply to a synchronous communication between the client and the supplier of a service. A message from a client object requests that a supplier object performs the service (method). The interpretation of a message and the means of executing it are at the discretion of the supplier object (this could be the delegation of the work to another object).

In architectural design, synchronous messages need to be considered separately from asynchronous communication where messages fire and service asynchronous events. In event processing there is a separation between an event originator (publisher object) and various event listeners/observers (subscriber objects) that want to be informed of an event occurrence and take their own, presumably different, actions.

In large systems, usually a separate registrator object performs the subscription, i.e. the “handshaking” between the publisher and subscribers. To register a subscriber with the publisher object, the registrator object acting on the subscriber’s behalf calls the publisher’s addActionListener() method with the subscriber object as an argument. If no registrator object is used, the subscriber object will directly call the addActionListener() method.

A method that intercepts an event (such as a mouse click on a JButton object) sends a “fire event” message to the publisher object. Typically the “fire event” message and the method that services it are in the
same publisher object. However, the method housing the “fire event” message depends only loosely on the
method that services it. It may be even possible that no subscriber object listens to the event and there is no
dependency at all.

Usually, the publisher object creates an event object – the publisher translates the intended meaning of
the event into an event object (called something like BCommandButtonEvent). The event object is
passed (in a callback operation) to all subscriber objects that registered their interests in the mouse click on
the button.

9.1.7.1 Event Processing and Layer Dependencies

In synchronous message passing, if client object A sends a message to supplier object B, then A depends on B
because A expects some results out of the execution of B. In asynchronous event processing, the sender
of the message is the publisher object but the message passing is handled as a callback. In a callback, the
publisher has no knowledge or interest in how the subscriber processes the event. The dependency exists
but it is negligible from the viewpoint of the architectural design.

The hand-shaking of subscribers and publishers causes a stronger dependency. If a registrator object
mediates the hand-shaking, then it depends on both the publisher and the subscriber. If a subscriber object
registers itself, then it depends on the publisher. To loosen dependencies due to hand-shaking, subscribers
can be passed to the registration methods in arguments typed as interfaces (Section 9.1.7.2). The
dependencies are then lowered, but further program analysis will be required to determine the class of a
subscriber. The analysis will involve the determination of classes that implement a subscriber interface.

For readability, the following examples adhere to two naming conventions. Firstly, the methods
performing registration of subscribers to a publisher are named beginning with the phrase add and ending
with the phrase Listener. The name of the event object is placed in-between these two phrases, e.g.
addXXXListener, where XXX is the name of the event object.

The second convention applies to methods that fire events in a publisher class. A publisher class will
normally house “normal” methods as well as methods that fire events. To distinguish between these two
kinds of methods, each event method begins with the phrase fire, such as fireCommandButtonEvent().

A fire method goes through the list of subscribers and, for each subscriber, calls its associated
process method. As a matter of convenience (rather than a rule), a name of a process method may
begin with the phrase process, such as processCommandButtonEvent(). A fire method
does not normally perform excessive calculation or program logic.

Figure 9-13 illustrates event processing involving the presentation and control « layer »
packages. CActioner is the only subscriber to PConsole’s events (PDisplayEvent). PDisplayEventRegistrator does the hand-shaking of PConsole and CActioner. The
do1() method intercepts and interprets the events, and it fireDisplayEvent(). PConsole
creates PDisplayEvent. Note that placing PDisplayEventRegistrator in the
presentation package creates an acceptable downward dependency between the presentation
and control « layer » packages (Section 9.1.3).
9. ARCHITECTURAL DESIGN

9.1.7.2 Event Processing and Interfaces

In Figure 9-13, the registrator class was placed strategically in the presentation «layer» package, and not in the control «layer» package, to avoid undesirable upward dependency between layers (Section 9.1.3). But strategic placement of classes has its limits and in a large design the circular dependencies due to message passing and event processing are increasingly likely. For example, classes in the presentation package may also want to subscribe to publishers in other packages, possibly even non-neighboring packages.

The resolution to cycles in event processing comes again from the skillful use of interfaces (Section 9.1.6.3). Figure 9-14 illustrates how even a downward dependency in Figure 9-13 can be reduced to oblivion. The dependency in Figure 9-13 was caused by a message call fireDisplayEvent() in PConsole to processDisplayEvent() in CActioner. In Figure 9-14, a new interface IPDisplayEventSubscriber separates PConsole from CActioner. PConsole uses IPDisplayEventSubscriber. CActioner implements IPDisplayEventSubscriber.
Figure 9-14 shows also how event processing can be linked together to accomplish a complex system. PConsole creates PDisplayEvent object and uses IPDisplayEvent Subscriber interface to notify CActioner of PDisplayEvent object. CActioner pre-processes PDisplayEvent with its processDisplayEvent() method and creates PDecisionEvent object.

CActioner, now in its capability as a publisher, uses ICDecisionEventSubscriber interface to notify MSynchronizer of PDecisionEvent object. MSynchronizer processes PDecisionEvent with its processDecisionEvent() method.
9.1.8 Acquaintance

As explained in opening paragraphs of Section 9.1, good architectural design requires the creation of dependency firewalls so that dependencies between neighboring layers are minimized, circular dependencies are eliminated and direct dependencies between non-neighboring layers are disallowed. The use of delegation, interfaces and event processing greatly contribute to these objectives.

However, there are situations when objects in non-neighboring layers may communicate without breaking the very principles of good architectural design. This can happen when an object gets acquainted at runtime with another object (in a non-neighboring layer) and wants to take advantage of this acquaintance by requesting services of that object.

**Acquaintance** defines a situation when an object is passed another object in an argument to its method. More precisely, an object A gets acquainted with object B if another object C passes B to A in an argument of the message to A. Object communication due to acquaintance is one of the programming techniques legitimized in the Law of Demeter (Lieberherr and Holland, 1989).

Figure 9-15 illustrates acquaintance. CActioner uses retrieveEmployee() to retrieve an EEmployee object from the database. The retrieval process is not explained, but presumably CActioner delegates the request down to a layer responsible for communication with the database. Once the employee information is retrieved, the EEmployee object is instantiated. CActioner passes EEmployee to PEmpBrowser in the message displayEmployee(emp). PEmpBrowser is now acquainted with EEmployee and sends the message getName() to it to get the employee name for display.
9.1.8.1 Acquaintance Dependencies and Interfaces

As shown in Figure 9-15, acquaintance introduces dependencies. **Acquaintance dependencies** are *method dependencies* acquired dynamically at runtime. The highly-dynamic nature of acquaintance dependencies distinguishes them from "regular" method dependencies discussed in Section 9.1.5.

From architectural perspective, dynamic means difficult to see and understand. Dynamic means also that it is not possible to use explicit associations between classes to visualize (and therefore legitimize) acquaintance dependencies in the code. Sometimes an acceptable solution can come from the introduction of interfaces into the design.
The design in Figure 9-15 has two architectural flaws:

- the control package is upward-dependent on the boundary package (because CActioner calls browser.displayEmployee(emp) on PEmpBrowser),
- the presentation package depends on the non-neighboring domain package (because PEmpBrowser calls emp.getName() on EEmployee).

Both flaws can be resolved with interfaces, as shown in Figure 9-16. The technique is similar to the one used to break circular dependencies with interfaces (Section 9.1.6.3). ICEmpBrowser breaks the upward dependency from control to presentation. IPEmployee breaks the acquaintance dependency from presentation to domain.

9.1.8.2 Acquaintance Package

In more complex scenarios, the use of interfaces and event processing (within the scope discussed in Sections 9.1.6 and 9.1.7) may not provide an obvious architectural solution to acquaintance dependencies. Even the solution in Figure 9-16 is rather unsatisfactory because the implementation dependency (Section 9.1.6.1) from EEmployee to IPEmployee spans non-neighboring packages.
Better solution comes from addressing the acquaintance as a problem in its own right that requires separate management. To this aim a separate acquaintance package may be introduced. The **acquaintance package** is a standalone package consisting of interfaces only. The package is not a layer and is not a part of the layer hierarchy.

Figure 9-17 demonstrates how the acquaintance package separates acquaintance dependencies into an isolated problem that can be managed independently. The model exemplifies the separation of concerns principle that should feature in any good architectural design (Larman, 2002). The acquaintance package consists only of interfaces. IAEmployee is extended by IAEmployeeWithSalary, which in turn is implemented by EEmployee.
As in Figures 9-15 and 9-16, CActioner’s method `initializeDisplay()` uses `retrieveEmployee()` to retrieve from the database and instantiate an `EEmployee` object (this process engages other objects not shown on the diagram). Although `EEmployee` provided interface (Section 9.1.6.1) is `IAEmployeeWithSalary`, CActioner requires only the `IAEmployee` interface (i.e. all employee information except salary).
The retrieved object (typed as IAEmployee) is passed to displayEmployee(emp: IAEmployee) in PEmpBrowser. PEmpBrowser is now acquainted with EEmployee and uses it to getName() – the service that is part of the PEmpBrowser’s required interface (Section 9.1.6.1). If the layer hierarchy is presentation – control – domain (entity) – foundation, then PEmpBrowser uses IAEmployee for downward access to EEmployee in a non-neighboring package.

In the bottom part of Figure 9-17, FUpdater requires the IAEmployeeWithSalary interface in the method saveEmployee(emp : IAEmployeeWithSalary). If some client passes to this method an EEmployee object, then the saveEmployee method can getSalary(), as well as getName(), prior to saving employee information in the database. FUpdater uses IAEmployeeWithSalary for upward access to EEmployee.

9.2 Architectural Frameworks

Framework is a construction on which the solution to a problem is built. In object technology, a framework is a reuse technology (Maciaszek, 2001). The technology applies to the reuse of the design, as opposed to the reuse of the code. Framework provides a skeleton solution to the problem, which needs to be customized and extended to serve a useful function. The customization involves writing the specific code, which “fills the gaps” in the framework (i.e. which implements various elements of the framework while conforming to the overall structural and behavioral framework design).

The best known reuse frameworks are Enterprise Resource Planning (ERP) systems, such as SAP or JDEdwards. ERP systems offer both design and code reuse. They are generic software packages. The software is customized and extended to suit the customer’s needs. The customization must be done within the skeleton design imposed by the framework.

Framework imposes an architectural design on the system. In this sense, every framework is an architectural framework, whether or not it is a software product like an ERP system or just a design idea and prerogative imposed on the development team. The latter is the subject of the discussion that follows.

9.2.1 Model-View-Controller

One of the best known architectural frameworks is the Model-View-Controller (MVC) framework developed as part of the Smalltalk-80 programming environment (Krasner and Pope, 1988). MVC made a huge footprint on object-oriented design. It is a classic example of the use of the principle of the separation of concerns in object-oriented design. In Smalltalk-80, MVC forces the programmers to divide application classes into three groups that specialize and inherit from three Smalltalk-provided abstract classes – Model, View and Controller.

Model objects represent data objects – the business entities and the business rules in the application domain. Changes to model objects are notified to view and controller objects via event processing. This uses the publisher/subscriber technique. Model is the publisher and it is therefore unaware of its views and controllers. View and controller objects subscribe to the Model, but they can also initiate changes to model objects. To assist in this task, Model supplies necessary interfaces, which encapsulate the business data and behavior.
**View objects** represent GUI objects and present the state of the model in the format required by the user, typically on a graphic display. View objects are decoupled from model objects. View subscribes to the Model so that it gets notified of Model changes and it can update its display. View objects can contain subviews, which display different parts of the Model. Typically, each view object is paired with a controller object.

**Controller objects** represent mouse and keyboard events. Controller objects respond to the requests that originate from View and that are results of user interactions with the system. Controller objects give meaning to keystrokes, mouse clicks, etc. and convert them into actions on the model objects. They mediate between view and model objects. By separating user input from visual presentation, they allow changing system response to user actions without changing the GUI presentation, and vice versa – changing GUI without changing system behavior.

Figure 9-18 illustrates an actor’s (user’s) perspective on communication between MVC objects. The arrows represent communication between objects (they are not really meant to represent dependencies as discussed in Section 9.1). The user GUI events are intercepted by view objects and passed to controller objects for interpretation and further action. Mixing the behavior of View and Controller in a single object is considered a bad practice in MVC.

Consider a scenario where a user activates a menu option to display customer details on the presentation screen. A view object receives the event and passes it to its controller object. The controller object requests Model to provide customer data. The model object returns the data to the controller object, which supplies it to View for display. Alternatively, the model object can pass the data to View directly. Any future changes to the state of the Model can be notified to view objects that subscribe to these changes. This way View can refresh its display to reflect the current business data values.

The MVC separation of concerns has numerous advantages. The most important of those are (Gamma *et al.*, 1995; Larman, 2002):

- Permitting the separate development of the GUI and the business data and logic of the model layer.
- Replacing or migrating to a different GUI without any radical changes needed in the model.
- Reorganizing or re-designing the model while retaining the GUI presentation to the user.
9. ARCHITECTURAL DESIGN

- Allowing multiple views on the same model state.
- Changing the way the GUI responds to user events without changing the GUI presentation (a view’s controller can be even changed at runtime).
- Enabling execution of the model without the GUI (e.g. for testing or for batch-mode processing).

9.2.2 Presentation-Control-Mediator-Entity-Foundation

The MVC principles underpin most modern architectural frameworks and related patterns. The Unified Process (UP) uses the MVC principles when separating classes into boundary, control and entity objects – the concepts derived from the Objectory method (Jacobson, 1992). Boundary objects correspond to MVC view objects, control objects are MVC controller objects, and entity objects represent MVC model objects.

The MVC approach has influenced virtually all architectural frameworks that emphasize the need for hierarchical layers of objects (Buschmann et al., 1996, Fowler, 2003) in order to manage object dependencies (Section 9.1). This Section introduces one such framework used in the book’s case-study. The framework is called Presentation-Control-Mediator-Entity-Foundation (PCMEF). The principles of PCMEF are similar to other popular layering schemes (Fowler, 2003).

PCMEF is a layered architectural framework organized as a vertical hierarchy. Each layer is a package that can contain other packages. Packages within each layer are known as partitions (Sections 9.1.2 and 9.1.3). In other words, the architecture consists of vertical layers, which can be partitioned. When the system grows, partitions can be organized into a system of sub-layers, i.e. partitions can be layered. Layering of partitions is outside the scope of the book.

In a strict layered architecture, higher layers utilize the services (and therefore depend on) lower layers, but not vice versa. A strict layered architecture is not fully possible in practice (Section 9.1.3). Nevertheless, with a careful design of layers, it is possible to devise an architectural framework where higher layers are unstable and lower layers are increasingly more stable (Section 9.1.3). Such a framework opens up to downward dependencies and limits upward dependencies in bottom-up object collaboration.

9.2.2.1 PCMEF Layers

The PCMEF framework consists of four layers: presentation, control, domain, and foundation. The domain layer contains two pre-defined packages: entity and mediator. With reference to the MVC framework, presentation corresponds to MVC View, control to Controller, and entity to Model. mediator and foundation do not have MVC counterparts.

The PCMEF main dependencies are downward, as shown by arrows in Figure 9-19 – presentation depends on control, control depends on domain (on mediator and, possibly but not necessarily, on entity), mediator depends on entity and on foundation. Upward dependencies are realized through loose coupling facilitated by interfaces, event processing, acquaintance package and similar techniques discussed in Section 9.1. Dependencies are only permitted between neighboring layers.
The PCMEF vertical layers preset the system architecture – invariant for all iterations of the incremental development process. The downward dependencies are made apparent by the requirement of setting unidirectional associations between dependent classes in neighboring layers. The horizontal dependencies within layers are relatively unrestricted as long as circular dependencies are eliminated.

In Java, the dependencies between packages convert to import statements (Section 9.1.1.2). The dependencies shown in Figure 9-19 correspond to the import statements in Listing 9-1:

```
package presentation;
import control.*;
package control;
import domain.entity.*;
import domain.mediator.*;
package entity;
package mediator;
import entity.*;
import foundation.*;
package foundation;
```
The presentation layer contains classes that define GUI objects. In Microsoft Windows environment, many presentation classes would be subclassed from the MFC (Microsoft Foundation Classes) library. In Java environment, presentation classes can be based on the classes and interfaces of Java Swing library. The user communicates with the system via presentation classes. Accordingly, the class containing the program’s main function is typically housed in the presentation package (alternatively it can reside in the control package).

The control layer handles presentation layer requests. It consists of classes responsible for processing user’s interactions (passed to control from presentation objects). As a result, control is responsible for bulk of the program’s logic, algorithmic solutions, main computations, and maintaining session state for each user.

The entity package of the domain layer handles control layer requests. It contains classes representing “business objects”. They store (in program’s memory) objects retrieved from the database or created in order to be stored in the database. Many entity classes are container classes.

The mediator package of the domain layer establishes a channel of communication that mediates between entity and foundation classes. Mediation serves two main purposes. Firstly, to isolate the two packages so that changes in any one of them can be introduced independently. Secondly, to eliminate a need for control classes to directly communicate with foundation classes whenever new entity objects need to be retrieved from the database (such requests from control classes are then channeled via mediator classes).

The foundation layer is responsible for all communications with database and web services, which manage persistent data required by the application program. This is where the connections to database and web servers are established, queries to persistent data are constructed, and the database transactions are instigated.

9.2.2.2 PCMEF Principles

Apart from the layered architecture, the PCMEF framework defines design principles (Section 9.1) and advocates design patterns (Section 9.3) necessary to properly manage object dependencies and deliver solutions that are understandable, maintainable and scalable. The primary PCMEF principles are:

1. Downward Dependency Principle (DDP)
2. Upward Notification Principle (UNP)
3. Neighbor Communication Principle (NCP)
4. Explicit Association Principle (EAP)
5. Cycle Elimination Principle (CEP)
6. Class Naming Principle (CNP)
7. Acquaintance Package Principle (APP)

The Downward Dependency Principle states that the main dependency structure is top-down. Objects in higher layers depend on objects in lower layers. Consequently, lower layers are more stable than higher layers. They are difficult to change, but (paradoxically) not necessarily difficult to extend (Martin, 2003). Interfaces, abstract classes, dominant classes and similar devices (Section 9.1.6) should encapsulate stable packages so that they can be extended when needed.
The Upward Notification Principle promotes low coupling in bottom-up communication between layers. This can be achieved by using asynchronous communication based on event processing (Section 9.1.7). Objects in higher layers act as subscribers (observers) to state changes in lower layers. When an object (publisher) in a lower layer changes its state, it sends notifications to its subscribers. In response, subscribers can communicate with the publisher (now in the downward direction) so that their states are synchronized with the state of the publisher.

The Neighbor Communication Principle demands that a package can only communicate directly with its neighbor package. This principle ensures that the system does not disintegrate into an incompressible network of intercommunicating objects. To enforce this principle, message passing between non-neighboring objects uses delegation (Section 9.1.5.1). In more complex scenarios, acquaintance package (Section 9.1.8.2) can be used to group interfaces to assist in collaboration that engages distant packages.

The Explicit Association Principle visibly enforces message passing between classes (Section 9.1.5). The principle is used predominantly on classes with downward dependencies (ref. the Downward Dependency Principle) and on classes engaged in frequent collaboration. Associations that are results of this principle are unidirectional (otherwise they would create circular dependencies). It must be remembered, however, that not all associations between classes exist because of message passing between these classes. For example, both-directional associations may be needed to implement referential integrity between classes.

The Cycle Elimination Principle ensures that circular dependencies between layers, between packages and between classes within packages are broken. Circular dependencies violate the separation of concerns guideline and are the main obstacle to reusability. Cycles can be resolved by creating a new package specifically for the purpose (Section 9.1.2) or by forcing one of the communication paths in the cycle to communicate via interface (Section 9.1.6.3).

The Class Naming Principle makes it possible to recognize in the class name to what package it belongs. To this aim, each class name is prefixed in PCMEF with the first letter of the package name (e.g. EInvoice is a class in the entity package). The same principle applies to interfaces. Each interface name is prefixed with two capital letters – the first is the letter “I” (signifying that this is interface) and the second letter identifies the package (e.g. ICInvoice is an interface in the control package).

The Acquaintance Package Principle is the consequence of the Neighbor Communication Principle. The acquaintance package consists of interfaces that an object passes, instead of concrete objects, in arguments to method calls. The interfaces can be implemented in any PCMEF package. This effectively allows communication between non-neighboring packages while centralizing dependency management to a single acquaintance package. The need for acquaintance package was explained in Section 9.1.8.2 and is discussed again next in PCMEF context.

9.2.2.3 Acquaintance in PCMEF+

Acquaintance happens when within a method, a message is sent to the object that is a parameter of the method (Section 9.1.8). Because acquaintance is acquired dynamically at runtime, special care must be exercised to avoid excessive coupling of objects due to acquaintance. The Acquaintance Package Principle ensures that the Neighbor Communication Principle is satisfied in the presence of acquaintance.

In PCMEF, acquaintance in the downward direction to objects in neighboring packages is never necessary because associations link such objects (which means that a message can be sent to an attribute in the client object (this) that links to the supplier object). Acquaintance in the upward direction to objects...
in neighboring packages should employ interfaces to eliminate cycles (Section 9.1.8.1). Acquaintance that in effect spans *non-neighboring* objects should be channeled through interfaces grouped in the *acquaintance* package. Figure 9-17 provides a good example.

Figure 9-20 shows how the *acquaintance* package effectively extends the PCMEF framework into the PCMEF+ framework. Interfaces of the *acquaintance* package can be implemented and/or used by classes of any PCMEF package (ref. the explanation of the implementation and usage dependencies in Sections 9.1.6.1 and 9.1.6.2, respectively). The external package containing PCMEF packages in Figure 9-20 is only for visualization purpose; it does not exist in practice.

Note again that the *acquaintance* package groups interfaces for upward communication and for downward communication to non-neighboring objects. Altogether, there are five possibilities for channeling the object communication via *acquaintance* interfaces:

1. $F$ uses $A$ interfaces implemented in $D$, $C$, or $P$
2. $D$ uses $A$ interfaces implemented in $C$ or $P$
3. $C$ uses $A$ interfaces implemented in $P$
4. $P$ uses $A$ interfaces implemented in $D$ or $F$
5. $C$ uses $A$ interfaces implemented in $F$
The acquaintance package can be seen as an application-specific library of interfaces. Like any library, the acquaintance package should be designed with the stability in mind. As mentioned in Section 9.2.2.2 in the context of the Downward Dependency Principle, a stable package is not easily amenable to changes but it can be extended to accommodate the system growth.

The same rule applies to external libraries used by the application program, such as Java API packages. PCMEF packages using Java data types depend on java.util, PCMEF packages engaged in event processing depend on java.awt.event, etc. The (relative) stability of Java libraries ensures that coupling PCMEF packages to them is unlikely to create a maintenance nightmare in the future.

9.2.2.4 Deployment of PCMEF Layers

The PCMEF architectural framework does not make a statement about the deployment of layers on computer nodes. The layers are intended to be deployable independently as components. Component is defined in the UML 2.0 as “a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment” (UML, 2002, p.322). The UML 2.0 provides also a Component Deployment Diagram to show deployment of components in an execution environment.

Deployment of PCMEF layers as components results in some kind of client/server architecture. As a minimum, the architecture consists of an application client and a database/web server. This is a two-tier architecture, where the application program on one computer node controls all PCMEF software objects and the database/web server on another computer node manages persistent data (Figure 9-19). The two-tier architecture applies to multi-user database applications as well as to single-user desktop applications. Desktop applications run the client and the server process on the same machine.

A two-tier architecture as described above is a thick-client setup. In more complex deployments, PCMEF layers – with the exception of the presentation layer – can be moved to server computers. This results in thin-client architectures. A thin-client architecture can be two tiers or many tiers. For example, the database server can be deployed on two nodes to separate database business and integrity rules (programmed with stored procedures and triggers) from the more mundane database responsibility of storing and managing the data. This results in a three-tier architecture, where PCMEF software objects are still running on a client tier, or possibly some domain and foundation objects are moved to the middle-tier.

Multi-tier architectures are also possible. In a distributed system, some PCMEF layers can be mapped to EJB or .NET components and deployed separately. In applications that have to display and process large volumes of data from a database, the domain layer can be deployed on an object server. An Object Storage API provides then the mapping between the persistent object server, the remaining PCMEF application layers, and the corporate relational database server.

9.3 Architectural Patterns

The discussion of architectural design is not complete without identifying, naming and explaining trade-offs of the design patterns advocated for architectural frameworks, such as PCMEF. A design pattern names and explains the best and widely acknowledged practice and knowledge to solve a design problem. Ever since the seminal work by Gamma et al. (1995), the study of design patterns has flourished and evolved into an important branch of knowledge within software engineering at large. The Gamma et al. (1995) book was
written by four prominent authors. Hence, these patterns are commonly known as Gang of Four (GoF) patterns.

Patterns increase awareness of proper application of common techniques for managing of dependencies, for reusing of functionality, for information hiding, for working with abstraction, etc. The study of patterns makes it possible to avoid “reinventing the wheel”. The previous discussion in this chapter applied many patterns without identifying and describing them. This Section revisits the previous discussion and explains what patterns have been used. All patterns discussed below are GoF patterns.

9.3.1 Facade

The intent of the Facade pattern is described in Gamma et al. (1995, p.185) as to define “a higher-level interface that makes the subsystem easier to use”. The goal is to “minimize the communication and dependencies between subsystems”.

A “higher-level interface” is not necessarily the concept of interface as introduced in Section 9.1.6. This can be an abstract class or concrete dominant class, also explained in Section 9.1.6. The point is that a higher-level interface encapsulates the main functionality of the subsystem (package) and provides the main or even the only entry point for the clients of the package.

Client objects communicate with the package via the façade object. The façade object delegates the work to other package objects as needed. The consequence is the reduction of communication paths between packages and the reduction of the number of objects that clients of the package deal with. In essence, the package gets hidden behind the façade object.

Figure 9-21 demonstrates the EEntity class as the façade to the entity package. Clients of the package, such as CActioner and MBroker, send requests to EEntity, which forwards them to the appropriate objects within the package. EEntity is a concrete dominant class. This could be an outer class and the three remaining classes could be inner classes (In Java, inner classes have their definitions placed within the outer class definition). Making EEntity an interface or abstract class could further reduce the clients’ dependencies on the entity package.
9.3.2 Abstract Factory

The Abstract Factory pattern provides “an interface for creating families of related or dependent objects without specifying their concrete classes” (Gamma et al., 1995, p.87). As opposed to the Façade pattern where a “higher-level interface” could mean a concrete class, an interface in Abstract Factory is a true interface (preferably) or an abstract class (Section 9.1.6).

Abstract Factory enables the application to behave differently by accessing one of several families of objects hidden behind the abstract factory interface. A configuration parameter value can control which family should be accessed.

Figure 9-22 gives an example where the application can execute using either a console display family of objects or a GUI window family of objects. IPPresentation is an abstract factory that defers creation of concrete objects to PConsole or PWindow concrete factories, depending how the system is configured. Client objects (such as PInit) access concrete objects via their interface (IPPresentation).
Because Abstract Factory is an interface, which gets implemented in entire families of classes, extending it to support new families may have a ripple effect on existing concrete classes. Gamma et al. (1995) discuss few implementation solutions to address this problem.

At closer inspection, the Abstract Factory pattern can be seen as a variation of the Façade pattern. The abstract factory interface can be used as a “higher-level interface” through which the communication to a package is channeled and the classes that do real work inside the package are encapsulated.

### 9.3.3 Chain of Responsibility

The intent of the **Chain of Responsibility** pattern is to “avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request” (Gamma et al., 1995, p.223). The chain of responsibility is really just another name for the concept of delegation (Section 9.1.5.1).

If chain of responsibility is involved, a client object that sends a message does not have a direct reference to an object that ultimately supplies the service. This pattern is necessary to enforce the **Neighbor Communication Principle** in the design (Section 9.2.2.2).

Figure 9-23 gives an example for the Chain of Responsibility pattern. PWindow gets displayContact() request and sends a retrieveContact() message to CActioner. CActioner cannot satisfy the request and forwards it to EContact (if the requested EContact object has been instantiated before and exists in the program’s memory) or to MBroker (if the EContact object has to be retrieved from the database). If the object is in memory, EContact will provide the service and the object will be returned to PWindow for display. Otherwise, MBroker will delegate the request to FReader so that the object can be retrieved from the database and instantiated as EContact before it is passed to PWindow.
9.3.4 Observer

The intent of the Observer pattern is to “define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” (Gamma et al., 1995, p.293). Also known as the Publish-Subscribe pattern, the Observer pattern is built into the handling of asynchronous communication in event processing (Section 9.1.7).

The pattern relates two types of objects: subject (the object that is observed) and observer. In Section 9.1.7, subject was called a publisher object and observer was called a subscriber object. A subject may have many observers, which subscribe to it. All observers are notified of subject state changes and can then perform necessary processing to synchronize their states with the subject state. Observers are not related to each other and can do different processing in response to notifications of subject state changes.

Although the definition of the Observer pattern mentions dependencies, the pattern promotes low coupling between subjects and observers. The notifications are broadcast automatically to observers. The subject does not know the concrete classes of observers, provided the notifications are based on interfaces that observers implement (Section 9.1.7.2). Subjects and observers execute in separate threats, thus further enhancing the low coupling. The low coupling of the Observer pattern can be (and should be) taken advantage in layered architectural frameworks to support the upward communication between layers.
The PCMEF framework sanctions the Observer pattern in the *Upward Notification Principle* (Section 9.2.2.2). The inter-layer chain of notifications is bottom-up. The *foundation* layer notifies its state changes to the *mediator* package, which passes them further to the *entity* package and to the *control* layer. The *control* layer sends the notifications up to the *presentation* layer.

Figure 9-24 shows how the Observer pattern is engaged in upward notification from the *entity* package to the *presentation* layer. *PContactBrowser* subscribes to *EContact*. Because the two classes are in non-neighboring packages, the *acquaintance* package is used as per the *Acquaintance Package Principle* (Section 9.2.2.2). The *acquaintance* package contains the *IAContactSubscriber* interface.

There is no dedicated *registrator object* (Section 9.1.7) in Figure 9-24. *EContact* uses `addContactListener()` to register *IAContactSubscriber* as its observer. In reality, the observer is *PContactBrowser*, which implements the *IAContactSubscriber* interface. When the state of *EContact* changes in some way, `fireContactChange()` notifies the change to *PContactBrowser* by calling `processContactChange()` on *IAContactSubscriber*. `processContactChange()` can then request `displayContact()` to display new state of *EContact* in the browser window.

### 9.3.5 Mediator

The *Mediator* pattern defines objects that encapsulate intercommunication between other objects, possibly from different layers. The pattern “promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently” (Gamma et al., 1995, p.273).

The Mediator pattern is given its rightful place in the PCMEF framework, where the *mediator* layer mediates between the *foundation* layer and the *entity* package (Section 9.2.2.1). The control
layer is another example of the Mediator pattern in its capability to mediate between the presentation layer and the entity package.

In PCMEF, the mediator package decouples the entity objects and the foundation objects. It acts also as a façade (Section 9.3.1) between the entity and foundation objects, thus reducing the number of interconnections between them. Façade differs from Mediator in that the Façade interconnection protocol is unidirectional and one-to-many. By contrast, Mediator replaces many-to-many interconnections with one-to-many interconnections to the colleagues that it mediates.

Moreover, the mediator package maintains synchronization between subjects (foundation) and observers (entity), thus promoting an enhanced form of the Observer pattern (Section 9.3.4). In the Observer pattern, the mediator object that encapsulates complex update semantics between subjects and observers is known as a Change Manager (Gamma et al., 1995).

Figure 9-25 is an example. CActioner delegates deleteContact() to MBroker. MBroker task is twofold. Firstly, it needs to delegate deleteContact() further down to FUpdater, which is responsible to delete the contact in the database (the database will also delete any out-messages still in the database to that contact).

Secondly, if the deletion in the database is successful, MBroker needs to check if EContact is in the program's memory, together with any EOutMessages that are destined to it. If it is, then MBroker

```java
public void deleteContact(int contactID) {
    if (updater.deleteContact(contactID)) {
        cnt = findContactInMemory(contactID);
        removeMessages(cnt);
        cnt.delete();
    }
}

protected void removeMessages(EContact cnt) {
    //assume find messages is implemented
    Collection msgs = findMessagesToContact(cnt);
    for(Iterator it=msgs.iterator(); it.hasNext(); ) {
        EOutMessage msg = (EOutMessage) it.next();
        msg.delete();
    }
}
```
sends `removeMessages(cnt)` message to delete `EOutmessages`, followed by to `cnt.delete()` to delete `EContact`.

**Summary**

1. *Software architecture* is defined as the organization of software elements into a system aiming at achieving some purpose.

2. *Architectural design* is a set of decisions aiming at efficient and effective software architecture together with the rationale for these decisions. A sound architectural design uses hierarchical layering of software objects (design classes) and ensures that dependencies between objects are minimized and visible in compile-time program structures.

3. Architectural layers are constructed from *packages*, which contain design classes. Packages can be nested. Accordingly, three levels of structural dependencies must be considered: *layer dependencies*, *package dependencies*, and *class dependencies*.

4. From the behavioral perspective, *method dependencies* are the main category of dependencies in a program. Method dependencies result from message passing between objects. A *client object* depends on a *supplier object*. In the presence of delegation, a supplier object can be a *delegator object*, which merely delegates the service to a real supplier of it.

5. *Interface* is a declaration of a set of features (attributes and operations) that is not directly instantiable. *Java interface* allows only attributes that are constants. *UML interface* allows any attributes, including attributes that create associations to classes and associations between interfaces.

6. Interface is different from *abstract class* or *pure abstract class*. In languages with single implementation inheritance, a class can be a subclass of at most one class (and that one class can be abstract or purely abstract). However, a class can implement multiple interfaces (and the same interface may be implemented by many classes).

7. The use of interfaces results in two kinds of dependencies: *implementation dependency* (when a class implements an interface) and *usage dependency* (when a class uses an interface). Interestingly, and in contrast to other dependencies, implementation and usage dependencies can be quite beneficial in software architectures. In particular, they can be used to break circular dependencies between classes, packages and layers.

8. *Event dependencies* are the second main category of dependencies in a program (apart from method dependencies). To be precise, event dependencies are a kind of method dependencies such that the message passing is asynchronous. In event processing, the dependency is between a *publisher object* and a *supplier object*. Unlike in synchronous method dependencies, event dependencies are weaker, more resistant to program changes, and easier to manage.

9. Combining event processing and interfaces creates the most powerful mechanism to facilitate dependency management in software architectures.
10. **Acquaintance** defines a situation when an object is passed another object in an argument to its method. The object passed can be (and frequently is) an interface. **Acquaintance dependencies** are method dependencies acquired dynamically at runtime. Acquaintance is a very useful programming practice, but its invisibility in compile-time program structures is a problem. One way of addressing the problem is to place interfaces used in acquaintances in a separate acquaintance package.

11. **Architectural framework** is a skeleton solution to a software development, which forces a broad architectural design on developers. One of the best known architectural frameworks is MVC.

12. This book applies architectural framework called PCMEF. PCMEF defines a number of strict software design principles. The aim is to minimize dependencies between software objects and facilitate understandability, maintainability and scalability in the resultant system. The framework is called PCMEF+ when the existence of the acquaintance package needs to be emphasized.

13. **Design pattern** names and explains the best and widely acknowledged practice and knowledge to solve a design problem. When a pattern serves the purpose of architectural design, it is called an architectural pattern.

14. Patterns that most prominently feature in the PCMEF framework are: Façade, Abstract Factory, Chain of Responsibility, Observer, and Mediator. These are all GoF patterns.

### Key Terms

- «communicate» relationship
- actor
- Advertising Expenditure Measurement
- aggregation
- artifact
- association
- Business Object Model
- business actor
- business class model
- business context diagram
- business entity
- Business Object Model
- business use case
- classifier
- data flow
- DOM
- Domain Object Model
- external entity
- generalization
- glossary
- primary actor
- relationship multiplicity
- relationship optionality
- relationship participation
- secondary actor
- stereotype
- unidirectional relationship
- Unified Process
- «include» relationship
- «communicate» relationship
- «extend» relationship
- «include» relationship
- actor
- aggregation
- application under development
- association
- attribute
- base use case
- CASE
- See Computer-Aided Software Engineering class
- CM
- See Contact Management
- Computer-Aided Software Engineering
- Contact Management
- data type
- descriptive attributes
- DOM
- See Domain Object Model
domain
domain class
domain class model
Domain Object Model
domain use case model
domain use case
domain class
domain class
domain class
domain class
extension use case
generalization
glossary
multimedia data
multiplicity
navigation
non-primitive data type
object instance
object instance
object instance
object instance
primitive data type
relationship
repository
See Unified Process

singleton class ................................................................. 42
structured information .............................................. 33
subclass .................................................................. 43
subset class .............................................................. 43
superclass ................................................................ 43
superset class ........................................................... 43
unidirectional relationship ..................................... 35
unstructured information ....................................... 33
use case ..................................................................... 34
visual modeling tool .................................................. 41
Chapter 09
application class ....................................................... 61
association attribute ................................................ 62
basic flow .................................................................. 55, 56
class ................................................................. 55, 56
conceptual class ...................................................... 61
EM ........................................................................ 52
Email Management .................................................. 52
Email Marketing ....................................................... 53
exception flow ........................................................ 56
flow of events .......................................................... 56
functionality ............................................................ 63
FURPS ....................................................................... 63
FURPS* ..................................................................... 63
performance ............................................................ 63
postcondition ........................................................... 54
precondition ............................................................. 54
regression testing ..................................................... 65
reliability ................................................................. 63
subflow .................................................................... 55, 57
supplementary specification ................................... 62
supportability .......................................................... 63
usability ................................................................... 63
use case document ................................................... 54
Chapter 09
View .......................................................................... 75
abstract class .......................................................... 80
pattern ................................................................. 103
acquaintance .......................................................... 89, 99
acquaintance dependency ..................................... 90
acquaintance package .......................................... 92, 100
Acquaintance Package Principle ......................... 99
APP ........................................................................... 99
architectural design ................................................. 70
architectural framework ........................................ 85
attribute ................................................................... 72
CEP ........................................................................ 72
Chain of Responsibility pattern ......................... 104
circular dependencies .......................................... 73
Class Naming Principle ........................................... 99
class ...................................................................... 78
client/server architecture ........................................ 101
CNP .......................................................................... 101
component ............................................................ 101
coupling ................................................................. 76
coupling ................................................................... 76
data member .......................................................... 72
DDP ........................................................................ 99
delegation ............................................................... 79, 104
delegator object ..................................................... 79
dependency ................................................................ 71
dependency firewall ................................................ 71
dependency relationship ......................................... 73
design class ............................................................. 71
design pattern ......................................................... 101
domain layer .......................................................... 98
dominant class ........................................................ 81
Downward Dependency Principle .................... 98
EAP ......................................................................... 98
event ...................................................................... 85
event object ............................................................ 86
explicit association ............................................... 99
Explicit Association Principle ......................... 99
Facade pattern ........................................................ 102
feature .................................................................... 72
field ........................................................................ 72
framework ........................................................... 98
Gang of Four .......................................................... 102
GoF......................................................................... 102
GoF See Gang of Four ............................................. 80
impact analysis ....................................................... 80
implementation dependency ............................... 81
inner class ............................................................. 102
interface .................................................................. 76
layer ........................................................................ 72, 75
layering ................................................................... 71
listener object ........................................................ 71
Mediator pattern ...................................................... 106
member function .................................................... See method
member variable ..................................................... See data member
message .................................................................... 72
message passing ..................................................... 72
method .................................................................... 72
model object .......................................................... 94
Model-View-Controller ......................................... 101
multi-tier architecture ........................................... 101
MVC ......................................................................... 99
Neighborhood Communication Principle ........... 99
Neighborhood Communication Principle See Neighbor Communication Principle
Observer ................................................................ 105
observer object ....................................................... 105
Observer pattern ..................................................... See observer pattern
observation .......................................................... 72
partition ................................................................... 75
PCMF See Presentation-Control-Domain-Foundation
PCMEF+ .................................................................. 100
PCMEF+ See Presentation-Control-Domain-Foundation
presentation layer ................................................ 98
Presentation-Control-Mediator-Entity-Foundation .. 96
proactive dependency management ..................... 71
provided interface .................................................. 81
publisher object ....................................................... 85
Publisher-Subscribe pattern ................................ 80
pure abstract class ................................................ 80
reactive dependency management ....................... 71
registrar object ....................................................... 85
required interface .................................................. 82
Separated Interface pattern ................................. 85
single implementation inheritance ...................... 80
software architecture ............................................. 70
stability .................................................................. 76
stable framework .................................................... 76
stable layer ............................................................. 76
subscriber object ..................................................... 85
supplier object ....................................................... 78
synchronous communication ............................... 79
thin-client ............................................................... 101
thin-client ............................................................... 101
together ................................................................ 102
three-tier architecture .......................................... 102
two-tier architecture ............................................. 102
UNP See Upward Notification Principle ............ 102
Upward Notification Principle ......................... 99
upward notification ................................................ 99
usage dependency .................................................. 97
view object .............................................................. 95
Chapter 10
abstraction ............................................................. 125
alternate key ........................................................... See unique key
association class ................................................... 126
9. Architectural Design

business rule ................................................................. 127
candidate key ............................................................ See unique key
column ........................................................................ 122
capitalization ................................................................ 122
conceptual classes ......................................................... 135
conceptual database model .............................................. 125
cursor ............................................................................ 135
data model ................................................................. See database model
database ........................................................................ 120
Database Management System ...................................... 120
database model ............................................................ 120
database package ........................................................ 130
database schema .......................................................... 123
DBMS .............................................................................. See Database Management System
declarative referential integrity ....................................... 127
embedded SQL .............................................................. 122, 129
Entity-Relationship Model ............................................. 125
ER ................................................................................. See Entity-Relationship Model
foreign key ..................................................................... 124
generalization ................................................................ 141
impedance mismatch ...................................................... 135
index ............................................................................. 131
inheritance .................................................................... 141
interactive SQL ............................................................ 122
logical database model ............................................... 124, 135
navigational link ........................................................ 121, 123
non-unique index ........................................................ 132
null .............................................................................. 123
Object Data Management Group .................................... 135
object database model .................................................. 135
object Storage API ........................................................ 135
OMG-Relational Mapping .............................................. 136
ODM .......................................................... See Object Data Management Group
persistent record ......................................................... 134
physical data independence ......................................... 131
physical database model .............................................. 124
primary key .................................................................... See primary key
primitive data type ........................................................ 122
procedural referential integrity ...................................... 127
programmatic SQL ....................................................... See embedded SQL
record ........................................................................ 121, 122
recursive association .................................................... 139
Referential integrity ........................................................ 124
relation ................................................................. See table
relational database model ............................................ 120
relational model .......................................................... See relational database model
row .............................................................................. See record
self-referential association ........................................... See recursive association
SQL (1999) .............................................................. See Structured Query Language
stored function .......................................................... 130
stored procedure ........................................................ 129
Structured Query Language ......................................... 120
table ............................................................................ 121
transaction manager .................................................... 120
transient objects .......................................................... 134
trigger ........................................................................ 127
unique index ............................................................... 132
unique key ................................................................. 123
Chapter 11
"instantiate" relationship .................................................. 167
activation activity diagram .......................................... 169
application class ........................................................ 152
asynchronous message .............................................. 170
callback message ........................................................ 170
class design ............................................................... 152
class instantiation ......................................................... 166
collaboration .............................................................. 171
collaboration diagram ................................................. See communication diagram
data communications diagram .................................... 153, 170
design class ................................................................. 152
elaboration ................................................................. 160, 184
functional requirements ............................................. 153
gate ........................................................................ 169
interaction .................................................................. 168, 169, 172
interaction design ....................................................... 152
interaction fragment .................................................. 172
interaction occurrence .............................................. 172, 185
interaction overview diagram ..................................... 172, 185
interface ...................................................................... 152
internal structure diagram .......................................... 171
Java interface ............................................................. 160
lifeline ......................................................................... 168
main method .............................................................. 167
method ........................................................................ 168
method reply .............................................................. See callback message
method return .......................................................... 170
non-functional requirements ..................................... 160
object create message ................................................ 170
part .......................................................... See responsibility
programmatic SQL ..................................................... See embedded SQL
sequence diagram .................................................... 153, 168, 184
signature ...................................................................... 153
signature ................................................................ 169
Unified Process .......................................................... 160
UP .............................................................................. See Unified Process
Chapter 12
"anonymous type" class ................................................. 199
acceptance testing ...................................................... 190, 219
agile software development ........................................ 209
anonymous inner class ................................................. 212
argument type ........................................................... 192
association link .......................................................... 196
automated verification ................................................. 221
Business Components for Java ..................................... 201
class data type ........................................................... 191
class interface ............................................................ 191
class method .............................................................. 192
class protocol ............................................................. See class interface
class template ........................................................... See parameterized type
collection ................................................................. 196
compiler ................................................................. See compiler
composite pattern ........................................................ 211
counter ................................................................. See collection
default ................................................................. See Java Development Kit
defect ........................................................................ 220
eJB .............................................................. See Enterprise JavaBeans
eXtreme Programming .................................................. 209
field ................................................................. See data member
fixture ........................................................................ 212
generic class ............................................................. See parameterized type
inner class ............................................................... 210
instantiation .............................................................. 212
instance variable ....................................................... 192
intentional programming ........................................... 215
Java Database Connectivity ......................................... 201
Java Development Kit ................................................ 201
Java Stored Procedures ............................................... 201
JavaBeans ............................................................... 201
JDBC .............................................................. See Java Database Connectivity
JDK .............................................................. See Java Development Kit
JUnit ................................................................. 211
local variable ........................................................... 169
manual verification ................................................... 221
method ................................................................. 192
method return .......................................................... 192
method prototype ..................................................... 192
method signature ...................................................... 192
Review Questions

Discussion Questions

1. Dependencies between software objects can be analyzed from different viewpoints and at various levels of abstraction. Which of these viewpoints or level of abstraction is most relevant for the notion of dependency firewall?

2. The book uses the term design class to mean – depending on the context – any of the following related terms: software class, system class, application class, implementation class, and program class. Explain differences in emphasis of these related terms.

3. There are two main methods to eliminate circular dependencies between packages. What are they?

4. If you were to classify various dependencies according to the impact on architectural design, what would this classification be? By impact, we mean the degree to which they ought to be minimized. Assume that the impact can be troublesome, neutral, or helpful.

5. What would be good alternative names for PCMEF packages?

6. Which of GoF patterns discussed in this chapter is the best match for the Upward Notification Principle (UNP)? Explain.

7. Which of GoF patterns discussed in this chapter can be used to implement a dominant class? Explain.
Problem-Solving Exercises

Case-Study Exercises

1. Consider the subflow \( S1 – View Unsent Messages \) in Section 8.2.3. Think about design classes that may be needed to implement this subflow. Draw a class diagram with PCMEF packages and assign design classes to these packages. You may need to use interfaces to ensure PCMEF principles.

Think about the sequence of operations needed to perform the subflow. Add these operations to classes in the diagram.

2. Consider the subflow \( S1 – View Unsent Messages \) in Section 8.2.3. Relate also to Exercise 1 above. Draw an alternative class diagram for this subflow. Use the PCMEF naming convention to name classes. Do not show packages. You may need to use interfaces to ensure PCMEF principles.

The alternative diagram should consider a scenario where viewing unsent messages is obtained by asking an employee object for outmessages allocated to that employee. If these outmessages are not in the memory cache, the program has to make a “trip” to the database.

Think about the sequence of operations needed to perform this scenario. Add the operations to classes in the diagram.

3. Consider the subflow \( S3 – Email Message \) in Section 8.2.3. Think about design classes that may be needed to implement this subflow. Use the PCMEF naming convention to name classes. Do not show packages. You may need to use interfaces to ensure PCMEF principles.

Consider a scenario where outmessage to be emailed has to be retrieved from the database only if it is not already in the memory cache. In the control package, use separate classes to retrieve the outmessage and to send it.

Think about the sequence of operations needed to perform this scenario. Add the operations to classes in the diagram.

Minicase – Contact Information Management

One of the use cases related to the EM case-study is Manage Contact Information (Figures 7-4 and 8-1). Like EM, Manage Contact Information is a subset of CM (Figure 7-1). CM is a subset of AEM.

**CIM (Contact Information Management)** is concerned with maintaining current information about contacts. As per the definition of contact in the domain glossary (Table 7-1), contact is a person or organization that AEM communicates or does business with. In practice, all contacts are people. It is just that in some cases a person that AEM communicates with is anonymous to AEM (or the personal details may be of no interest to AEM). Also, in some cases, a contact may not represent an organization but a
private person (and yet, AEM wants to keep information about this person). For these reasons, CIM distinguishes between PersonContact and OrganizationContact as two kinds of Contact.

The functionality of CIM consists of four standard operations that can be performed on a business object – Create it, Read it, Update it, and Delete it. This is nicknamed the CRUD functionality. Figure 9-26 shows a use case diagram for CIM. The four main use cases are all modeled as abstract because of the differences in handling PersonContacts and OrganizationContacts.

Read Contact in Figure 9-26 is extended by two other use cases because a contact has to be read (retrieved) from the database before the application program can update it or delete it. Update OrganizationContact is extended by Create PersonContact to signify the fact that a PersonContact can be created in the process of updating OrganizationContact. Once created, PersonContact can be assigned to OrganizationContact by extending the use case Update PersonContact.

Create PersonContact is extended by Assign PersonContact to OrganizationContact. This means that PersonContact can be (but does not have to be) added to OrganizationContact when it is created.
The abstract use case Read Contact retrieves contact information from the database and presents a list of contacts in browser window, as shown in Figure 9-27. The window is for the concrete use case Create OrganizationContact. Recall from the domain glossary (Table 7-1) that there are six predefined organization types: advertisers, advertiser groups, agencies, agency groups, outlets, and providers. An organization can be one, many or none of these types.

<table>
<thead>
<tr>
<th>Organization</th>
<th>Advertiser</th>
<th>Advertiser Group</th>
<th>Agency</th>
<th>Agency Group</th>
<th>Outlet</th>
<th>Provider</th>
<th>Date Created</th>
<th>Emp [CR]</th>
<th>Date Modified</th>
<th>Emp [MD]</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yvesmen's</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>9/3/2003</td>
<td>LAW</td>
<td>4/12/2003</td>
<td>BIL</td>
<td></td>
</tr>
<tr>
<td>802 Test</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>9/3/2003</td>
<td>BIL</td>
<td>4/12/2003</td>
<td>LAW</td>
<td></td>
</tr>
<tr>
<td>174 Tone</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>9/3/2003</td>
<td>BIL</td>
<td>4/12/2003</td>
<td>LAW</td>
<td></td>
</tr>
<tr>
<td>Team R Us</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>9/3/2003</td>
<td>BIL</td>
<td>4/12/2003</td>
<td>LAW</td>
<td></td>
</tr>
<tr>
<td>Rub The B</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>9/3/2003</td>
<td>BIL</td>
<td>4/12/2003</td>
<td>LAW</td>
<td></td>
</tr>
<tr>
<td>Mr Lesser</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>9/3/2003</td>
<td>BIL</td>
<td>4/12/2003</td>
<td>LAW</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 9-27  Browser window for OrganizationContact*

The columns called Date Created, Cr Emp [CR], Date Modified, and Emp [MD] contain dates and initials of employees who created and last-modified contact information. For each OrganizationContact, CIM can identify many, one, or zero PersonContacts. PersonContacts can be retrieved and displayed in a separate browser window (not shown).

The use case Read Contact displays only basic information about contacts. Complete information about contacts includes other details, such as address, phone, etc. This information is addressed by other use cases, as discussed next.

The use case Create Contact presents to the actor “entry windows” (dialog boxes) to insert details about OrganizationContact or PersonContact. The functionality of this use case can be derived from the examples of entry windows presented in Figures 9-28 and 9-29. Note that windows are prototypes only and subject to changes during detailed design. For example, the use case Create Contact should allow for four main action buttons:

- **OK** (i.e. save the entered details in the database, dismiss the window, and return to the browser window),
- **Cancel** (i.e. cancel all entered information and return to the browser window),
- **Clear** (i.e. clear all entered information, do not dismiss the window, and allow to re-enter),
- **Save** (i.e. save the entered details in the database, do not dismiss the window, and allow to enter next contact).

Figure 9-28 shows window prototype with three tab pages for Create OrganizationContact. This facility allows entering organization details, including postal and courier address (by using separate tabs). The window does not provide for entering PersonContacts associated with OrganizationContact. The assignment of PersonContacts to OrganizationContact can be done via use cases Create PersonContact or Update OrganizationContact.
Figure 9.28 Entry window for OrganizationContact

Figure 9.29 shows window prototype for Create PersonContact. The use case allows assigning PersonContact to OrganizationContact. OrganizationContact must exist before PersonContact can be assigned. Action buttons are missing from the prototype window.

Figure 9.30 presents window prototype for the use case Update OrganizationContact. As modeled in the use case diagram (Figure 9.26), this use case can be extended by two other use cases: Create PersonContact and Assign PersonContact to OrganizationContact. For this reason, the window has one more tab page called Contacts.
Figure 9-30  Update window for OrganizationContact

Figure 9-31 shows the update window for OrganizationContact with the Contacts tab page opened. This tab page allows assigning existing PersonContact to the ContactList (by activating the use case Assign PersonContact to OrganizationContact) or to create new PersonContact (Figure 9-29) before assigning it. Creating new PersonContact involves the use case Create PersonContact.

The use case Update PersonContact presents the same window as the use case Create PersonContact (Figure 9-29). The difference is that the fields in the window for Update PersonContact contain values. There are also differences with regard to business rules and allowed actions (e.g., the set of action buttons may be different).

The use case Delete OrganizationContact allows deleting OrganizationContact after relevant information is displayed (Figure 9-32). There is no Contacts tab page in the window. This
means that CIM does not allow deleting in one action `OrganizationContact` and all its associated `PersonContacts`. Also, `OrganizationContact` cannot be deleted if it still has `PersonContacts` associated.

Deleting `PersonContacts` is conducted by the use case `Delete PersonContact`. Window prototype for it is not shown here, but the window fields are the same as in `Create PersonContact` (Figure 9-29).

1. Based on the minicase specifications, develop a conceptual class diagram. Use the informational content of prototype windows to discover classes and their attributes. Define multiplicity and participation of associations. Explain the model.

2. Based on the minicase specifications, develop a design class diagram conforming to the PCMEF+ framework. Name classes according to PCMEF+ conventions. Show only the most important operations. Explain the model.

3. Refer to your solution for Question 2 above. Discuss the use in your model of the five GoF patterns introduced in Chapter 9. If some patterns have not been used, modify your class diagram to take advantage of all five patterns. Explain the model.