Chapter 16

GRASP: DESIGNING OBJECTS WITH RESPONSIBILITIES

The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents.

—Nathaniel Borenstein

Objectives

- Define patterns.
- Learn to apply five of the GRASP patterns.

Introduction

Object design is sometimes described as some variation of the following:

> After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.

Such terse advice is not especially helpful, because there are deep principles and issues involved in these steps. Deciding what methods belong where, and how the objects should interact, is terribly important and anything but trivial. It takes careful explanation, applicable while diagramming and programming.

And this is a critical step—this is at the heart of what it means to develop an object-oriented system, not drawing domain model diagrams, package diagrams, and so forth.

GRASP as a Methodical Approach to Learning Basic Object Design

It *is* possible to communicate the detailed principles and reasoning required to grasp basic object design, and to learn to apply these in a methodical approach that removes the magic and vagueness.

The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles is based on *patterns of assigning responsibilities.*

16.1 Responsibilities and Methods

The UML defines a **responsibility** as "a contract or obligation of a classifier" [OMG01]. Responsibilities are related to the obligations of an object in terms of its behavior. Basically, these responsibilities are of the following two types:

- knowing
- doing

Doing responsibilities of an object include:

- o doing something itself, such as creating an object or doing a calculation
- o initiating action in other objects
- o controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- o knowing about private encapsulated data
- o knowing about related objects
- o knowing about things it can derive or calculate

Responsibilities are assigned to classes of objects during object design. For example, I may declare that "a *Sale* is responsible for creating *SalesLineItems"* (a doing), or "a *Sale* is responsible for knowing its total" (a knowing). Relevant responsibilities related to "knowing" are often inferable from the domain model, because of the attributes and associations it illustrates.

The translation of responsibilities into classes and methods is influenced by the granularity of the responsibility. The responsibility to "provide access to relational databases" may involve dozens of classes and hundreds of methods, packaged in a subsystem. By contrast, the responsibility to "create a *Sale"* may involve only one or few methods.

A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities. Responsibilities are implemented using methods that either act alone or collaborate with other methods and objects. For example, the *Sale* class might define one or more methods to know its total; say, a method named *getTotal.* To fulfill that responsibility, the *Sale* may collaborate with other objects, such as sending *agetSubtotal* message to each *SalesLineltem* object asking for its subtotal.

16.2 Responsibilities and Interaction Diagrams

The purpose of this chapter is to help methodically apply fundamental principles for assigning responsibilities to objects. This will often be done while programming. Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams (which are part of the UP Design Model), whose basic notation we examined in the previous chapter.

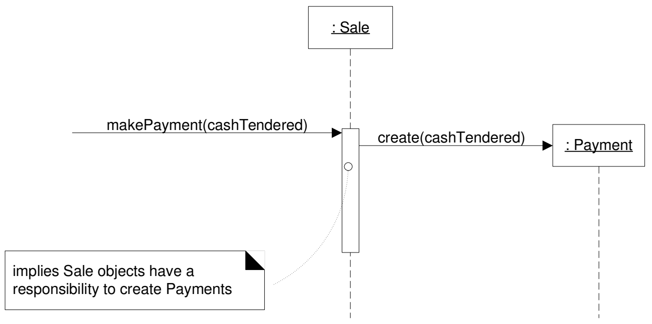

: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
implies Sale objects have a responsibility to create Payments

Figure 16.1 Responsibilities and methods are related.

Figure 16.1 indicates that *Sale* objects have been given a responsibility to create *Payments,* which is invoked with a *makePayment* message and handled with a corresponding *makePayment* method. Furthermore, the fulfillment of this responsibility requires collaboration to create the *SalesLineltem* object and invoke its constructor.

In summary, interaction diagrams show choices in assigning responsibilities to objects. When created, decisions in responsibility assignment are made, which are reflected in what messages are sent to different classes of objects. This chapter emphasizes fundamental principles—expressed in the GRASP patterns—to guide choices in where to assign responsibilities. These choices are reflected in interaction diagrams.

16.3 Patterns

Experienced object-oriented developers (and other software developers) build up a repertoire of both general principles and idiomatic solutions that guide them in the creation of software. These principles and idioms, if codified in a structured format describing the problem and solution, and given a name, may be called **patterns.** For example, here is a sample pattern:

Pattern Name:	Information Expert
Solution:	Assign a responsibility to the class that has the information needed to fulfill it.
Problem It Solves:	What is a basic principle by which to assign responsibilities to objects?

In object technology, a **pattern** is a named description of a problem and solution that can be applied to new contexts; ideally, it provides advice in how to apply it in varying circumstances, and considers the forces and trade-offs.[1] Many patterns provide guidance for how responsibilities should be assigned to objects, given a specific category of problem.

Most simply, a **pattern** is a named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs.

"One person's pattern is another person's primitive building block" is an object technology adage illustrating the vagueness of what can be called a pattern [GHJV94]. This treatment of patterns will bypass the issue of what is appropriate to label a pattern, and focus on the pragmatic value of using the pattern style as a vehicle for naming, presenting, learning, and remembering useful software engineering principles.

Repeating Patterns

New pattern could be considered an oxymoron, if it describes a new idea. The very term "pattern" is meant to suggest a repeating thing. The point of patterns is not to express new design ideas. Quite the opposite is true—patterns attempt to codify *existing* tried-and-true knowledge, idioms, and principles; the more honed and widely used, the better.

1. The formal notion of patterns originated with the (building) architectural patterns of Christopher Alexander [AIS77]. Patterns for software originated in the 1980s with Kent Beck, who became aware of Alexander's pattern work in architecture, and then were developed by Beck with Ward Cunningham [BC87, Beck94].

Consequently, the GRASP patterns—which will soon be introduced—do not state new ideas; they are a codification of widely used basic principles. To an object expert, the GRASP patterns—by idea if not by name—will appear very fundamental and familiar. That's the point!

Patterns Have Names

All patterns ideally have suggestive names. Naming a pattern, technique, or principle has the following advantages:

- It supports chunking and incorporating that concept into our understanding and memory.
- It facilitates communication.

Naming a complex idea such as a pattern is an example of the power of abstraction—reducing a complex form to a simple one by eliminating detail. Therefore, the GRASP patterns have concise names such as *Information Expert, Creator, Protected Variations.*

Naming Patterns Improves Communication

When a pattern is named, we can discuss with others a complex principle or design idea with a simple name. Consider the following discussion between two software designers, using a common vocabulary of patterns *(Creator, Factory,* and so on) to decide upon a design:

Fred: "Where do you think we should place the responsibility for creating a *SalesLineltem?* I think a *Factory."*

Wilma: "By *Creator,* I think *Sale* will be suitable."

Fred: "Oh, right—I agree."

Chunking design idioms and principles with commonly understood names facilitates communication and raises the level of inquiry to a higher degree of abstraction.

16.4 GRASP: Patterns of General Principles in Assigning Responsibilities

To summarize the preceding introduction:

- The skillful assignment of responsibilities is extremely important in object design.
- Determining the assignment of responsibilities often occurs during the cre ation of interaction diagrams, and certainly during programming.

- Patterns are named problem/solution pairs that codify good advice and principles often related to the assignment of responsibilities.

Question:	What are the GRASP patterns?
Answer:	They describe fundamental principles of object design and responsibility assignment, expressed as patterns.

Understanding and being able to apply these principles during the creation of interaction diagrams is important because a software developer new to object technology needs to master these basic principles as quickly as possible; they form the foundation of how a system will be designed.

GRASP is an acronym that stands for General Responsibility Assignment Software Patterns.[2] The name was chosen to suggest the importance of *grasp ing* these principles to successfully design object-oriented software.

How to Apply the GRASP Patterns

The following sections present the first five GRASP patterns:

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

There are others, introduced in a later chapter, but it is worthwhile mastering these five first because they address very basic, common questions and fundamental design issues.

Please study the following patterns, note how they are used in the example interaction diagrams, and then apply them during the creation of new interaction diagrams. Start by *mastering Information Expert, Creator, Controller, High Cohesion,* and *Low Coupling.* Later, learn the remaining patterns.

16.5 The UML Class Diagram Notation

A UML class box used to illustrate software classes often shows three compartments; the third illustrates the methods of the class, as shown in Figure 16.2.

2. Technically, one should write "GRAS Patterns" rather than "GRASP Patterns," but the latter sounds better.

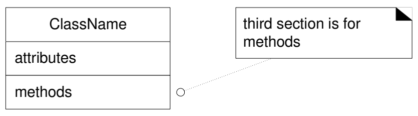


Figure 16.2 Software classes illustrate method names.

The details of this notation are explored in a subsequent chapter. In the following discussion on patterns, this form of class box will occasionally be used.

16.6 Information Expert (or Expert)

Solution Assign a responsibility to the information expert—the class that has the *information* necessary to fulfill the responsibility.

Problem What is a general principle of assigning responsibilities to objects?

A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. Done well, systems tend to be easier to understand, maintain, and extend, and there is more opportunity to reuse components in future applications.

Example In the NextGEN POS application, some class needs to know the grand total of a sale.

Start assigning responsibilities by clearly stating the responsibility.

By this advice, the statement is:

Who should be responsible for knowing the grand total of a sale"?

By *Information Expert,* we should look for that class of objects that has the information needed to determine the total.

Now we come to a key question: Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed? The Domain Model illustrates conceptual classes of the real-world domain; the Design Model illustrates software classes.

Answer:

1. If there are relevant classes in the Design Model, look there first.
2. Else, look in the Domain Model, and attempt to use (or expand) its represen tations to inspire the creation of corresponding design classes.

For example, assume we are just starting design work and there is no or a minimal Design Model. Therefore, we look to the Domain Model for information experts; perhaps the real-world *Sale* is one. Then, we add a software class to the Design Model similarly called *Sale,* and give it the responsibility of knowing its total, expressed with the method named *getTotal.* This approach supports *low representational gap* in which the software design of objects appeals to our concepts of how the real domain is organized.

To examine this case in detail, consider the partial Domain Model in Figure 16.3.

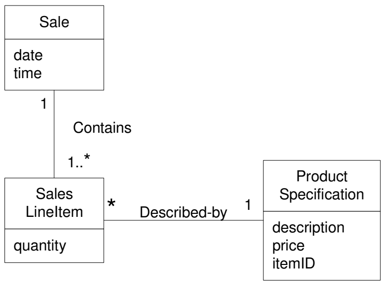


Figure 16.3 Associations of Sale.

What information is needed to determine the grand total? It is necessary to know about all the *SalesLineltem* instances of a sale and the sum of their subtotals. A *Sale* instance contains these; therefore, by the guideline of Information Expert, *Sale* is a suitable class of object for this responsibility; it is an *information expert* for the work.

As mentioned, it is in the context of the creation of interaction diagrams that these questions of responsibility often arise. Imagine we are starting to work through the drawing of diagrams in order to assign responsibilities to objects. A partial interaction diagram and class diagram in Figure 16.4 illustrate some decisions.

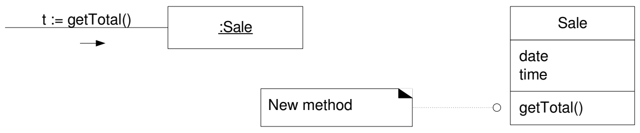


Figure 16.4 Partial interaction and class diagrams.

We are not done yet. What information is needed to determine the line item subtotal? *SalesLineltem.quantity* and *ProductSpecification.price* are needed. The *SalesLineltem* knows its quantity and its associated *ProductSpecification;* therefore, by Expert, *SalesLineltem* should determine the subtotal; it is the *information expert.*

In terms of an interaction diagram, this means that the *Sale* needs to send *get-Subtotal* messages to each of the *SalesLineltems* and sum the results; this design is shown in Figure 16.5.

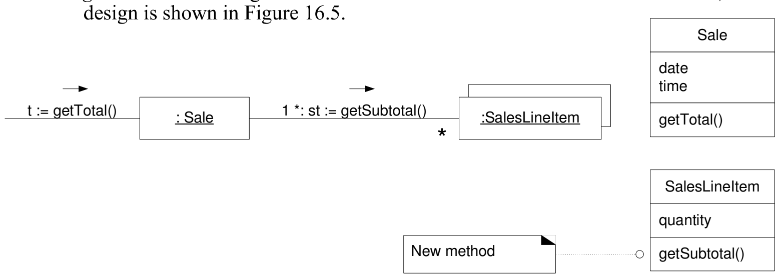


Figure 16.5 Calculating the Sale total

To fulfill the responsibility of knowing and answering its subtotal, a *Sales-Lineltem* needs to know the product price.

The *ProductSpecification* is an information expert on answering its price; therefore, a message must be sent to it asking for its price.

The design is shown in Figure 16.6.

In conclusion, to fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects as follows.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

The context in which these responsibilities were considered and decided upon was while drawing an interaction diagram. The method section of a class diagram can then summarize the methods.

The principle by which each responsibility was assigned was Information Expert—placing it with the object that had the information needed to fulfill it.

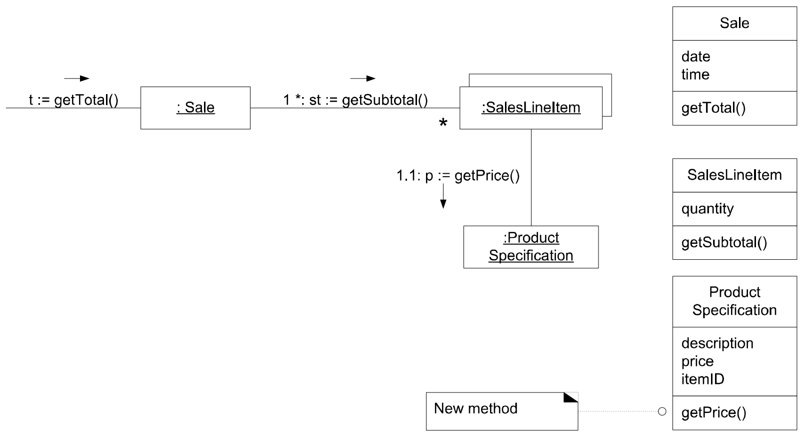


Figure 16.6 Calculating the *Sale* total.

Discussion Information Expert is frequently used in the assignment of responsibilities; it is a basic guiding principle used continuously in object design. Expert is not meant to be an obscure or fancy idea; it expresses the common "intuition" that objects do things related to the information they have.

Notice that the fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many "partial" information experts who will collaborate in the task. For example, the sales total problem ultimately required the collaboration of three classes of objects.

Whenever information is spread across different objects, they will need to interact via messages to share the work.

Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents; Peter Goad calls this the "Do It Myself" strategy [Coad95]. For example, in the real world, without the use of electro-mechanical aids, a sale does not tell you its total; it is an inanimate thing. Someone calculates the total of the sale. But in object-oriented software land, all software objects are "alive" or "animated," and they can take on responsibilities and do things. Fundamentally, they do things related to the information they know. I call this the "animation" principle in object design; it is like being in a cartoon where everything is alive.

The Information Expert pattern—like many things in object technology—has a real-world analogy. We commonly give responsibility to individuals who have the information necessary to fulfill a task. For example, in a business, who should be responsible for creating a profit-and-loss statement? The person who has access to all the information necessary to create it—perhaps the chief financial officer. And just as software objects collaborate because the information is spread around, so it is with people. The company's chief financial officer may ask accountants to generate reports on credits and debits.

Contraindications There are situations where a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion (these principles are discussed later in this chapter).

For example, who should be responsible for saving a *Sale* in a database? Certainly, much of the information to be saved is in the *Sale* object, and thus by Expert an argument could be made to put the responsibility in the *Sale* class. And the logical extension of this decision is that each class has its own services to save itself in a database. But this leads to problems in cohesion, coupling, and duplication. For example, the *Sale* class must now contain logic related to database handling, such as related to SQL and JDBC (Java Database Connectivity). The class is no longer focused on just the pure application logic of "being a sale;" it now has other kinds of responsibilities, which lowers its cohesion. The class must be coupled to the technical database services of another subsystem, such as JDBC services, rather than just being coupled to other objects in the domain layer of software objects, which raises its coupling. And it is likely that similar database logic would be duplicated in many persistent classes.

All these problems indicate violation of a basic architectural principle: design for a separation of major system concerns. Keep application logic in one place (such as the domain software objects), keep database logic in another place (such as a separate persistence services subsystem), and so forth, rather than intermingling different system concerns in the same component.[3]

3. See Chapter 32 for a discussion of separation of concerns.

Supporting a separation of major concerns improves coupling and cohesion in a design. Thus, even though by Expert there could be some justification to put the responsibility for database services in the *Sale* class, for other reasons (usually cohesion and coupling), it is a poor design.

Benefits

- Information encapsulation is maintained, since objects use their own infor mation to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems. (Low Coupling is also a GRASP pat tern that is discussed in a following section).
- Behavior is distributed across the classes that have the required informa tion, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain. High cohesion is usually supported (another pattern discussed later).

Related Patterns or Principles

- Low Coupling
- High Cohesion

Also Known As; Similar To "Place responsibilities with data," "That which knows, does," "Do It Myself," "Put Services with the Attributes They Work On."

16.7 Creator

Solution Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B *aggregates* A objects.
- B *contains* A objects.
- B *records* instances of A objects.
- B *closely uses* A objects.
- B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).

B is a *creator* of A objects.

If more than one option applies, prefer a class B which *aggregates* or *contains* class A.

Problem Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.

Example In the POS application, who should be responsible for creating a *SalesLineltem* instance? By Creator, we should look for a class that aggregates, contains, and so on, *SalesLineltem* instances. Consider the partial domain model in Figure 16.7.

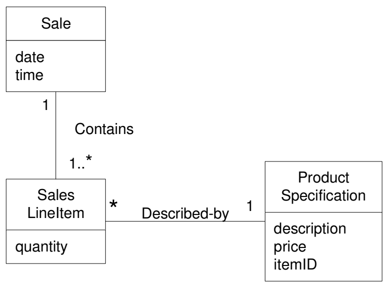


Figure 16.7 Partial domain model.

Since a Sate contains (in fact, aggregates) many *SalesLineltem* objects, the Creator pattern suggests that *Sale* is a good candidate to have the responsibility of creating *SalesLineltem* instances.

This leads to a design of object interactions as shown in Figure 16.8.

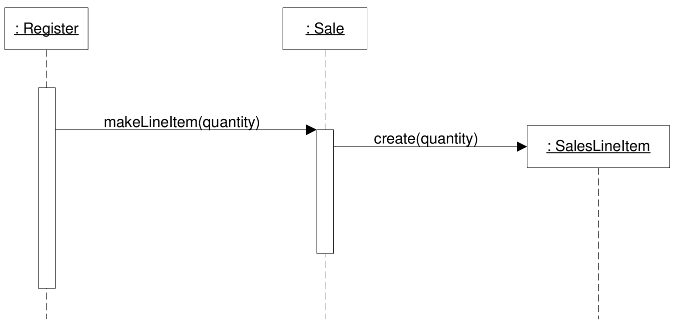


Figure 16.8 Creating a SalesLineltem.

This assignment of responsibilities requires that a *makeLineltem* method be defined in Sate.

Once again, the context in which these responsibilities were considered and decided upon was while drawing an interaction diagram. The method section of a class diagram can then summarize the responsibility assignment results, concretely realized as methods.

Discussion Creator guides assigning responsibilities related to the creation of objects, a very common task. The basic intent of the Creator pattern is to find a creator that needs to be connected to the created object in any event. Choosing it as the creator supports low coupling.

Aggregate *aggregates* Part, Container *contains* Content, and Recorder *records* Recorded are all very common relationships between classes in a class diagram. Creator suggests that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded. Of course, this is only a guideline.

Note that the concept of **aggregation** has been used in considering the Creator pattern. Aggregation is discussed in Chapter 27; a brief definition is that aggregation involves things that are in a strong Whole-Part or Assembly-Part relationship, such as Body aggregates Leg or Paragraph aggregates Sentence.

Sometimes a creator is found by looking for the class that has the initializing data that will be passed in during creation. This is actually an example of the Expert pattern. Initializing data is passed in during creation via some kind of initialization method, such as a Java constructor that has parameters. For example, assume that a *Payment* instance needs to be initialized, when created, with the *Sale* total. Since *Sale* knows the total, *Sale* is a candidate creator of the *Payment.*

Contraindications Often, creation requires significant complexity, such as using recycled instances for performance reasons, conditionally creating an instance from one of a family of similar classes based upon some external property value, and so forth. In these cases, it is advisable to delegate creation to a helper class called a *Factory* [GHJV95] rather than use the class suggested by *Creator.* Factories are discussed in Chapter 23.

Benefits Low coupling (described next) is supported, which implies lower maintenance dependencies and higher opportunities for reuse. Coupling is probably not increased because the *created* class is likely already visible to the *creator* class, due to the existing associations that motivated its choice as creator.

Related Patterns or Principles

Low Coupling

Factory

Whole-Part [BMRSS96] describes a pattern to define aggregate objects that support encapsulation of components.

16.8 Low Coupling

Solution Assign a responsibility so that coupling remains low.

Problem How to support low dependency, low change impact, and increased reuse?

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements; "too many" is context-dependent, but will be examined. These elements include classes, subsystems, systems, and so on.

A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:

- Changes in related classes force local changes.
- Harder to understand in isolation.
- Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Example Consider the following partial class diagram from a NextGen case study:

Payment
Register
Sale

Assume we have a need to create a *Payment* instance and associate it with the *Sale.* What class should be responsible for this? Since a *Register* "records" a *Payment* in the real-world domain, the Creator pattern suggests *Register* as a candidate for creating the *Payment.* The *Register* instance could then send an *addPayment* message to the *Sale,* passing along the new *Payment* as a parameter. A possible partial interaction diagram reflecting this is shown in Figure 16.9.

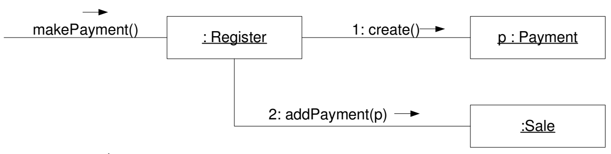


Figure 16.9 Register creates Payment.

This assignment of responsibilities couples the *Register* class to knowledge of the *Payment* class.

UML notation: Note that the *Payment* instance is explicitly named *p* so that in message 2 it can be referenced as a parameter.

An alternative solution to creating the *Payment* and associating it with the *Sale* is shown in Figure 16.10.

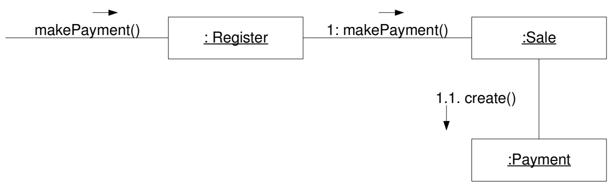


Figure 16.10 Sale creates Payment.

Which design, based on assignment of responsibilities, supports Low Coupling? In both cases we will assume the *Sale* must eventually be coupled to knowledge of a *Payment.* Design 1, in which the *Register* creates the *Payment,* adds coupling *of Register* to *Payment,* while Design 2, in which the *Sale* does the creation of a *Payment,* does not increase the coupling. Purely from the point of view of coupling, Design Two is preferable because overall lower coupling is maintained. This an example where two patterns—Low Coupling and Creator—may suggest different solutions.

In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and High Cohesion. Nevertheless, it is one factor to consider in improving a design.

Discussion Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an **evaluative principle** that a designer applies while evaluating all design decisions.

In object-oriented languages such as C++, Java, and C#, common forms of coupling from *TypeX* to *TypeY* include:

- *TypeX* has an attribute (data member or instance variable) that refers to a *TypeY* instance, or *TypeY* itself.
- A *TypeX* object calls on services of a *TypeY* object.
- *TypeX* has a method that references an instance of *TypeY,* or *TypeY* itself, by any means. These typically include a parameter or local variable of type *TypeY,* or the object returned from a message being an instance of *TypeY.*
- *TypeX* is a direct or indirect subclass of *TypeY.*

- *TypeY* is an interface, and *TypeX* implements that interface.

Low Coupling encourages assigning a responsibility so that its placement does not increase the coupling to such a level that it leads to the negative results that high coupling can produce.

Low Coupling supports the design of classes that are more independent, which reduces the impact of change. It can't be considered in isolation from other patterns such as Expert and High Cohesion, but rather needs to be included as one of several design principles that influence a choice in assigning a responsibility.

A subclass is strongly coupled to its superclass. The decision to derive from a superclass needs to be carefully considered since it is such a strong form of coupling. For example, suppose that objects need to be stored persistently in a relational or object database. In this case it is a relatively common design to create an abstract superclass called *PersistentObject* from which other classes derive. The disadvantage of this subclassing is that it highly couples domain objects to a particular technical service and mixes different architectural concerns, whereas the advantage is automatic inheritance of persistence behavior.

There is no absolute measure of when coupling is too high. What is important is that a developer can gauge the current degree of coupling, and assess if increasing it will lead to problems. In general, classes that are inherently very generic in nature, and with a high probability for reuse, should have especially low coupling.

The extreme case of Low Coupling is when there is no coupling between classes. This is not desirable because a central metaphor of object technology is a system of connected objects that communicate via messages. If Low Coupling is taken to excess, it yields a poor design because it leads to a few incohesive, bloated, and complex active objects that do all the work, with many very passive zero-coupled objects that act as simple data repositories. Some moderate degree of coupling between classes is normal and necessary to create an object-oriented system in which tasks are fulfilled by a collaboration between connected objects.

Contraindications High coupling to stable elements and to pervasive elements is seldom a problem. For example, a Java J2EE application can safely couple itself to the Java libraries *(java.util,* and so on), because they are stable and widespread.

Pick Your Battles

It is not high coupling per se that is the problem; it is high coupling to elements that are unstable in some dimension, such as their interface, implementation, or mere presence.

This is an important point: As designers, we can add flexibility, encapsulate details and implementations, and in general design for lower coupling in many areas of the system. But, if we put effort into "future proofing" or lowering the coupling at some point where in fact there is no realistic motivation, this is not time well spent.

Designers have to pick their battles in lowering coupling and encapsulating things. Focus on the points of realistic high instability or evolution. For example, in the NextGen project, it is known that different third-party tax calculators (with unique interfaces) need to be connected to the system. Therefore, designing for low coupling at this variation point is practical.

Benefits

- not affected by changes in other components
- simple to understand in isolation
- convenient to reuse

Background Coupling and cohesion (described next) are truly fundamental principles in design, and should be appreciated and applied as such by all software developers. Larry Constantine, also a founder of structured design in the 1970s and a current advocate of more attention to usability engineering [CL99], was primarily responsible in the 1960s for identifying and communicating coupling and cohesion as critical principles [ConstantineGS, CMS74].

Related Patterns ■ Protected Variation

16.9 High Cohesion

Solution Assign a responsibility so that cohesion remains high.

Problem How to keep complexity manageable?

In terms of object design, **cohesion** (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.

A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly effected by change

Low cohesion classes often represent a very "large grain" of abstraction, or have taken on responsibilities that should have been delegated to other objects.

Example The same example problem used in the Low Coupling pattern can be analyzed for High Cohesion.

Assume we have a need to create a (cash) *Payment* instance and associate it with the *Sale.* What class should be responsible for this? Since *Register* records a *Payment* in the real-world domain, the Creator pattern suggests *Register* as a candidate for creating the *Payment.* The *Register* instance could then send an *addPayrnent* message to the *Sale,* passing along the new *Payment* as a parameter, as shown in Figure 16.11.

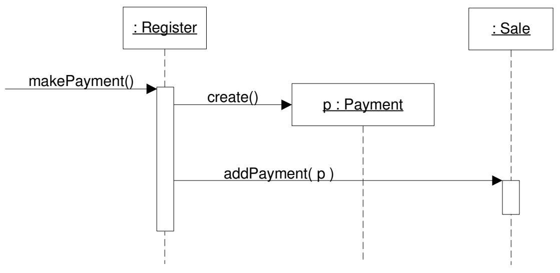


Figure 16.11 Register creates Payment.

This assignment of responsibilities places the responsibility for making a payment in the *Register.* The *Register* is taking on part of the responsibility for fulfilling the *makePayment* system operation.

In this isolated example, this is acceptable; but if we continue to make the *Register* class responsible for doing some or most of the work related to more and more system operations, it will become increasingly burdened with tasks and become incohesive.

Imagine that there were fifty system operations, all received by *Register.* If it did the work related to each, it would become a "bloated" incohesive object. The point is not that this single *Payment* creation task in itself makes the *Register* incohesive, but as part of a larger picture of overall responsibility assignment, it may suggest a trend toward low cohesion.

And most important in terms of developing skills as an object designer, regardless of the final design choice, the valuable thing is that at least a developer knows to consider the impact on cohesion.

By contrast, as shown in Figure 16.12, the second design delegates the payment creation responsibility to the *Sale,* which supports higher cohesion in the

Since the second design supports both high cohesion and low coupling, it is desirable.

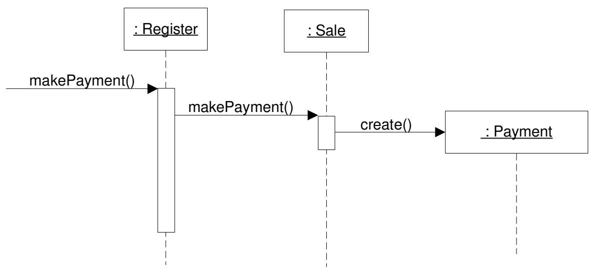


Figure 16.12 Sale creates Payment

In practice, the level of cohesion alone can't be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.

Discussion Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an evaluative principle that a designer applies while evaluating all design decisions.

Grady Booch describes high functional cohesion as existing when the elements of a component (such as a class) "all work together to provide some well-bounded behavior" [Booch94].

Here are some scenarios that illustrate varying degrees of functional cohesion:

1. *Very low cohesion*—A class is solely responsible for many things in very dif ferent functional areas.
   - o Assume a class exists called *RDB-RPC-Interface* which is completely responsible for interacting with relational databases and for handling remote procedure calls. These are two vastly different functional areas, and each requires lots of supporting code. The responsibilities should be split into a family of classes related to RDB access and a family related to RFC support.
2. *Low cohesion*—A class has sole responsibility for a complex task in one func tional area.
   - o Assume a class exists called *RDBInterface* which is completely responsible for interacting with relational databases. The methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds or thousands of methods. The class should split into a family of lightweight classes sharing the work to provide RDB access.

3. *High cohesion*—A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
   - o Assume a class exists called *RDBInterface* which is only partially responsible for interacting with relational databases. It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.
4. *Moderate cohesion*—A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept, but not to each other.
   - o Assume a class exists called *Company* which is completely responsible for (a) knowing its employees and (b) knowing its financial information. These two areas are not strongly related to each other, although both are logically related to the concept of a company. In addition, the total number of public methods is small, as is the amount of supporting code.

As a rule of thumb, a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.

A class with high cohesion is advantageous because it is relatively easy to maintain, understand, and reuse. The high degree of related functionality, combined with a small number of operations, also simplifies maintenance and enhancements. The fine grain of highly related functionality also supports increased reuse potential.

The High Cohesion pattern—like many things in object technology—has a real-world analogy. It is a common observation that if a person takes on too many unrelated responsibilities—especially ones that should properly be delegated to others—then the person is not effective. This is observed in some managers who have not learned how to delegate. These people suffer from low cohesion; they are ready to become "unglued."

Another Classic Principle: Modular Design

Coupling and cohesion are old principles in software design; designing with objects does not imply ignoring well-established fundamentals. Another of these—which is strongly related to coupling and cohesion—is to promote **modular design.** To quote:

> Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules [Booch94].

We promote a modular design by creating methods and classes with high cohesion. At the basic object level, modularity is achieved by designing each method with a clear, single purpose, and grouping a related set of concerns into a class.

Cohesion and Coupling; Yin and Yang

Bad cohesion usually begets bad coupling, and vice versa. 1 call cohesion and coupling *the yin and yang of software engineering* because of their interdependent influence. For example, consider a GUI widget class that represents and paints a widget, saves data to a database, and invokes remote object services. Not only is it profoundly incohesive, but it is coupled to many (and disparate) elements.

Contraindications There are a few cases in which accepting lower cohesion is justified.

One case is the grouping of responsibilities or code into one class or component to simplify maintenance by one person—although be warned that such grouping may also make maintenance worse. But for example, suppose an application contains embedded SQL statements that by other good design principles should be distributed across ten classes, such as ten "database mapper" classes. Now, it is common that only one or two SQL experts know how to best define and maintain this SQL, even if there are dozens of object-oriented (OO) programmers on the project; few OO programmers may have strong SQL skills. Suppose the SQL expert is not even a comfortable OO programmer. The software architect may decide to group all the SQL statements into one class, *RDBOperations,* so that it is easy for the SQL expert to work on the SQL in one location.

Another case for components with lower cohesion is with distributed server objects. Because of overhead and performance implications associated with remote objects and remote communication, it is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations. This is also related to the pattern called **Coarse-Grained Remote Interface,** in which the remote operations are made more coarse-grained in order to do or request more work in remote operation call, because of the performance penalty of remote calls over a network. As a simple example, instead of a remote object with three fine-grained operations *setName, setSalary,* and *setHi-reDate,* there is one remote operation *setData* which receives a set of data. This results in less remote calls, and better performance.

Benefits

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- The fine grain of highly related functionality supports increased reuse because a cohesive class can be used for a very specific purpose.

16.10 Controller

Solution Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system, device, or subsystem *(facade controller).*
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session *(use-case or session controller).*
    - o Use the same controller class for all system events in the same use case scenario.
    - o Informally, a session is an instance of a conversation with an actor. Sessions can be of any length, but are often organized in terms of use cases (use case sessions).

Corollary: Note that "window," "applet," "widget," "view," and "document" classes are not on this list. Such classes should *not* fulfill the tasks associated with system events, they typically receive these events and delegate them to a controller.

Problem Who should be responsible for handling an input system event?

An input **system event** is an event generated by an external actor. They are associated with **system operations**—operations of the system in response to system events, just as messages and methods are related.

For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended." Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

Example In the NextGen application, there are several system operations, as illustrated in Figure 16.13, showing the system itself as a class or component (which is legal in the UML).

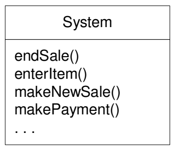
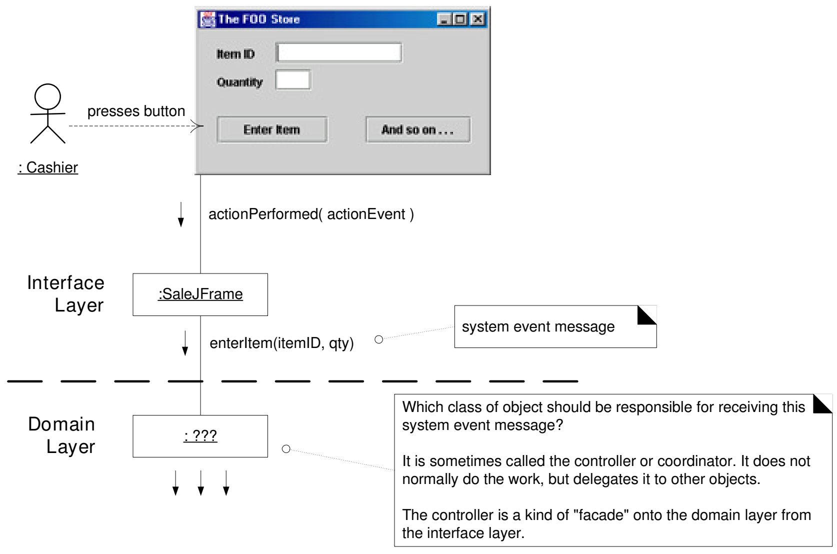


Figure 16.13 System operations associated with the system events.

> During analysis, system operations may be assigned to the class *System*, to indicate they are system operations. However, this does *not* mean that a software class named *System* fulfills them during design. Rather, during design, a Controller class is assigned the responsibility for system operations (see Figure 16.14).

Who should be the controller for system events such as *enterItem* and *endSale*!



Which class of object should be responsible for receiving this system event message?

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

Figure 16.14 Controller for enterItem?

By the Controller pattern, here are some choices:

represents the overall "system," device, or subsystem	*Register, POSSystem*
represents a receiver or handler of all system events of a use case scenario	*ProcessSaleHandler, ProcessSaleSestsion*

In terms of interaction diagrams, it means that one of the examples in Figure 16.15 may be useful.

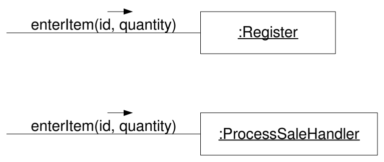


Figure 16.15 Controller choices.

The choice of which of these classes is the most appropriate controller is influenced by other factors, which the following section explores.

During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such *as Register,* as shown in Figure 16.16.

Discussion Systems receive external input events, typically involving a GUI operated by a person. Other mediums of input include external messages such as in a call processing telecommunications switch, or signals from sensors such as in process control systems.

In all cases, if an object design is used, some handler for these events must be chosen. The Controller pattern provides guidance for generally accepted, suitable choices. As illustrated in Figure 16.14, the controller is a kind of facade into the domain layer from the interface layer.

It is often desirable to use the same controller class for all the system events of one use case so that it is possible to maintain information about the state of the use case in the controller. Such information is useful, for example, to identify out-of-sequence system events (for example, a *makePayment* operation before an *endSale* operation). Different controllers may be used for different use cases.

A common defect in the design of controllers is to give them too much responsibility.

Normally, a controller should *delegate* to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.

Please see the "Issues and Solutions" section later for elaboration.

The first category of controller is a facade controller representing the overall system, device, or a subsystem. The idea is to choose some class name that suggests a cover, or facade, over the other layers of the application, and that provides the main point of service calls from the UI layer down to other layers. It could be an abstraction of the overall physical unit, such as a *Register*[4], *TelecommSwitch, Phone,* or *Robot; a* class representing the entire software system, such as *POSSystem,* or any other concept which the designer chooses to represent the overall system or a subsystem, even, for example, *ChessGame* if it was game software.

Facade controllers are suitable when there are not "too many" system events, or it is not possible for the user interface (UI) to redirect system event messages to alternating controllers, such as in a message processing system.

If a use-case controller is chosen, then there is a different controller for each use case. Note that this is not a domain object; it is an artificial construct to support the system (a *Pure Fabrication* in terms of the GRASP patterns). For example, if the NextGen application contains use cases such as *Process Sale* and *Handle Returns,* then there may be a *ProcessSaleHandler* class and so forth.

When should you choose a use-case controller? It is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling, typically when the facade controller is becoming "bloated" with excessive responsibilities. A use-case controller is a good choice when there are many system events across different processes; it factors their handling into manageable separate classes, and also provides a basis for knowing and reasoning about the state of the current scenario in progress.

In the UP and Jacobson's older Objectory method [Jacobson92], there are the (optional) concepts of boundary, control, and entity classes. **Boundary objects** are abstractions of the interfaces, **entity objects** are the application-independent (and typically persistent) domain software objects, and **control objects** are use case handlers as described in this Controller pattern.

A important corollary of the Controller pattern is that interface objects (for example, window objects or widgets) and the presentation layer should not have responsibility for fulfilling system events. In other words, system operations should be handled in the application logic or domain layers of objects rather than in the interface layer of a system. See the "Issues and Solutions" section for an example.

The Controller object is typically a client-side object within the same process as the UI (for example, an application with a Java Swing GUI), and so is not exactly applicable when the UI is a Web client in a browser, and there is server-side software involved. In the latter case, there are various common patterns of handling the system events that are strongly influenced by the chosen server-side technical framework, such as Java servlets. Nevertheless, it is a common idiom to create server-side use-case controllers with either a servlet for each use case or an Enterprise JavaBeans (EJB) session bean for each use case. The

4. Various terms are used for a physical POS unit, including register, point-of-sale terminal (POST), and so forth. Over time, "register" has come to embody the notion of both a physical unit, and the logical abstraction of the thing that registers sales and payments.

server-side session object represents a "session" of interaction with an external actor.

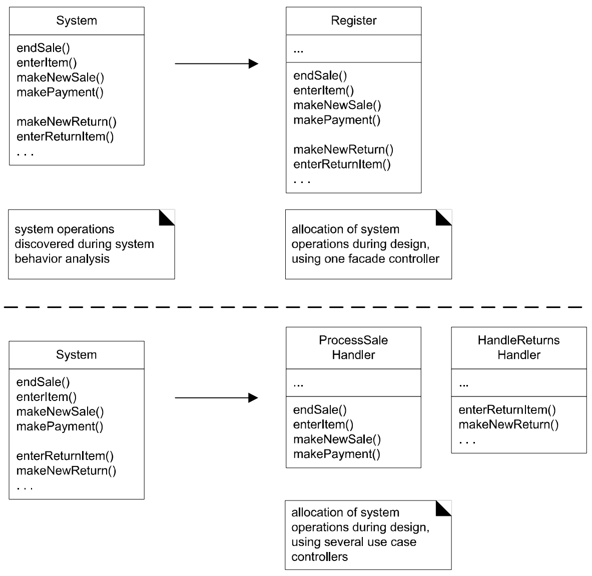


Figure 16.16 Allocation of system operations.

If the UI is not a web client (for example, it is a Swing or Windows GUI), but the application calls on remote services, it is still common to use the Controller pattern. The UI forwards the request to the local client-side Controller, and the Controller may forward all or part of the request handling on to remote services. This design lowers the coupling of the UI to remote services, and makes it easier, for example, to provide the services either locally or remotely, through the indirection of the client-side Controller.

To summarize, the Controller receives the service requests from the UI layer and coordinates their fulfillment, usually by delegation to other objects.

Benefits

- *Increased potential for reuse,* and *pluggable interfaces*—It ensures that application logic is *not* handled in the interface layer. The responsibilities of a controller could technically be handled in an interface object, but the implication of such a design is that program code and logic related the fulfillment of application logic would be embedded in interface or window objects. An interface-as-controller design reduces the opportunity to reuse logic in future applications, since it is bound to a particular interface (for example, window-like objects) that is seldom applicable in other applications. By contrast, delegating a system operation responsibility to a controller supports the reuse of the logic in future applications. And since the application logic is not bound to the interface layer, it can be replaced with a different interface.
- *Reason about the state of the use case*—It is sometimes necessary to ensure that system operations occur in a legal sequence, or to be able to reason about the current state of activity and operations within the use case that is underway. For example, it may be necessary to guarantee that the *makePay-ment* operation can not occur until the *endSale* operation has occurred. If so, this state information needs to be captured somewhere; the controller is one reasonable choice, especially if the same controller is used throughout the use case (which is recommended).

Issues and Solutions

Bloated Controllers

Poorly designed, a controller class will have low cohesion—unfocused and handling too many areas of responsibility; this is called a bloated controller. Signs of bloating include:

- There is only a *single* controller class receiving *all* system events in the sys tem, and there are many of them. This sometimes happens if a facade con troller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the sys tem event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.
- A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.

There are several cures to a bloated controller, including:

1. Add more controllers—a system does not have to have only one. Instead of facade controllers, use use-case controllers. For example, consider an application with many system events, such as an airline reservation system.

It may contain the following controllers:

Use-case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

2. Design the controller so that it primarily delegates the fulfillment of each system operation responsibility on to other objects.

Interface Layer Does Not Handle System Events

To reiterate: an important corollary of the Controller pattern is that interface objects (for example, window objects) and the interface layer should not have responsibility for handling system events. As an example, consider a design in Java that uses a *JFrame* to display the information.

Assume the NextGen application has a window that displays sale information and captures cashier operations. Using the Controller pattern, Figure 16.17 illustrates an acceptable relationship between the *JFrame* and Controller and other objects in a portion of the POS system (with simplifications).

Notice that the *SaleJFrame* class—part of the interface layer—passes the *enter-Item* message to the *Register* object. It did not get involved in processing the operation or deciding how to handle it; the window only delegated it to another layer.

Assigning the responsibility for system operations to objects in the application or domain layer—using the Controller pattern rather than the interface layer supports increased reuse potential. If an interface layer object (like the *SaleJ-Frame)* handles a system operation—which represents part of a business pro-cess—then business process logic would be contained in an interface (for example, window-like) object, which has low opportunity for reuse because of its coupling to a particular interface and application.

Consequently, the design in Figure 16.18 is undesirable.

Placing system operation responsibility in a domain object controller makes it easier to reuse the program logic supporting the associated business process in future applications. It also makes it easier to unplug the interface layer and use a different interface framework or technology, or to run the system in an offline "batch" mode.

Message Handling Systems and the Command Pattern

Some applications are message-handling systems or servers that receive requests from other processes. A telecommunications switch is a common example. In such systems, the design of the interface and controller is somewhat different. The details are explored in a later chapter, but in essence, a common solution is to use the Command pattern [GHJV95] and Command Processor pattern [BMRSS96], introduced in Chapter 34.

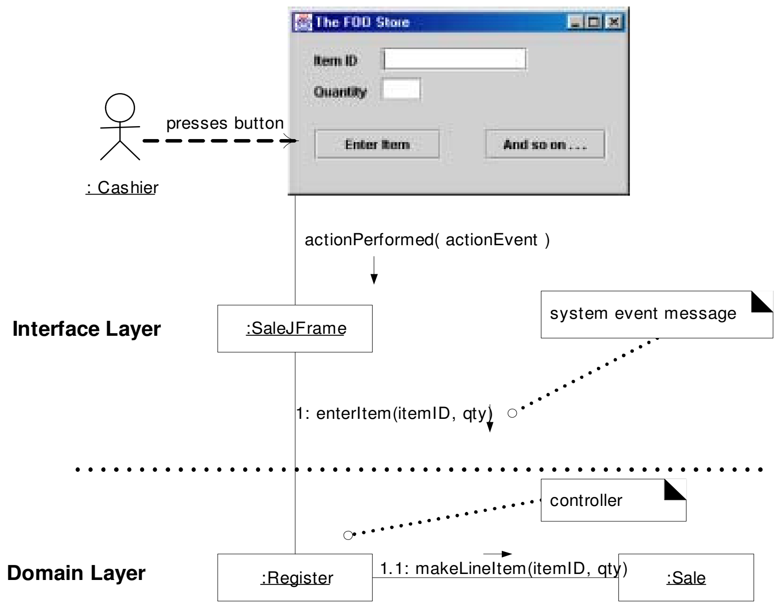


Figure 16.17 Desirable coupling of interface layer to domain layer.

Related Patterns

- **Command**—In a message-handling system, each message may be represented and handled by a separate Command object [GHJV95].
- **Facade**—A facade controller is a kind of Facade [GHJV95].
- **Layers**—This is a POSA pattern [BMRSS96]. Placing domain logic in the domain layer rather than the presentation layer is part of the Layers pattern.

Pure Fabrication—This is another GRASP pattern. A Pure Fabrication is an arbitrary creation of the designer, not a software class whose name is inspired by the Domain Model. A use-case controller is a kind of Pure Fabrication.

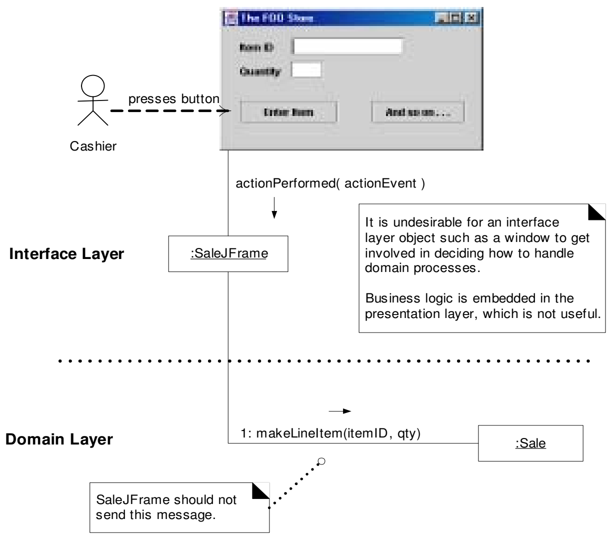

The FOO Store
Item ID
Quantity
Enter Item
And so on...
presses button
Cashier
actionPerformed(actionEvent)
Interface Layer
:SaleJFrame
It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.
Business logic is embedded in the presentation layer, which is not useful.
Domain Layer
1: makeLineItem(itemID, qty)
:Sale
SaleJFrame should not send this message.

Figure 16.18 Less desirable coupling of interface layer to domain layer.

16.11 Object Design and CRC Cards

Although not formally part of the UML, another device sometimes used to help assign responsibilities and indicate collaboration with other objects are **CRC cards** (Class-Responsibility-Collaborator cards) [BC89]. These were pioneered by Kent Beck and Ward Cunningham, who are largely responsible for encouraging objects designers to think more abstractly in terms of responsibility assignment and collaborations, and also for the use of patterns.

CRC cards are index cards, one for each class, upon which the responsibilities of the class are briefly written, and a list of collaborator objects to fulfill those responsibilities. They are usually developed in a small group session. The GRASP patterns may be applied when considering the design while using CRC cards.

CRC cards are one approach to recording the results of responsibility assignment and collaborations. The recording can be enhanced with the use of interaction and class diagrams. The real value is not the cards or the diagrams, but the consideration of responsibility assignment.

16.12 Further Readings

The metaphor of collaborating objects with responsibilities, or **Responsibility-Driven Design,** especially emerged from the influential object work in Smalltalk at Tektronix in Portland, from Kent Beck, Ward Cunningham, Rebecca Wirfs-Brock, and others. *Designing Object-Oriented Software* [WWW90] is the landmark text, and as relevant today as when it was written.

Two other recommended texts emphasizing fundamental object design principles are *Object-Oriented Design Heuristics* by Riel, and *Object Models* by Coad.