Chapter 17

DESIGN MODEL: USE-CASE REALIZATIONS WITH GRASP PATTERNS

To invent, you need a good imagination and a pile of junk.

—Thomas Edison

Objectives

- Design use-case realizations.
- Apply the GRASP patterns to assign responsibilities to classes.
- Use the UML interaction diagram notation to illustrate the design of objects.

Introduction

This chapter explores how to create a design of collaborating objects with responsibilities. Particular attention is given to the application of the GRASP patterns to develop a well-designed solution. Please note that the GRASP patterns as such or by name are not the important thing; they are just a learning aid to help talk about and methodically do fundamental object design.

This chapter communicates the principles, using the NextGen POS example, by which an object-oriented designer assigns responsibilities and establishes object interactions—a core skill in object-oriented development.

Note:

> The assignment of responsibilities and design of collaborations are very important and creative steps during design, either while diagraming or while programming.

The material is intentionally detailed; it attempts to exhaustively illustrate that there is no "magic" or unjustifiable decisions in object design—assignment of responsibilities and the choice of object interactions can be rationally explained and learned.

17.1 Use-Case Realizations

To quote, "A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects" [RUP]. More precisely, a designer can describe the design of one or more *scenarios* of a use case; each of these is called a use-case realization. Use-case realization is a UP term or concept used to remind us of the connection between the requirements expressed as use cases, and the object design that satisfies the requirements.

UML interaction diagrams are a common language to illustrate use-case realizations. And as was explored in the prior chapter, there are principles and patterns of object design, such as Information Expert and Low Coupling, that can be applied during this design work.

To review, Figure 17.20 (near the end of this chapter) illustrates the relationship between some UP artifacts:

- The use case suggests the system events that are explicitly shown in system sequence diagrams.
- Details of the effect of the system events in terms of changes to domain objects may optionally be described in system operation contracts.
- The system events represent messages that initiate interaction diagrams, which illustrate how objects interact to fulfill the required tasks—the use case realization.
- The interaction diagrams involve message interaction between software objects whose names are sometimes inspired by the names of conceptual classes in the Domain Model, plus other classes of objects.

17.2 Artifact Comments

Interaction Diagrams and Use-Case Realizations

In the current iteration we are considering various scenarios and system events such as:

- *Process Sale: makeNewSale, enterItem, endSale, makePayment*

If collaboration diagrams are used to illustrate the use-case realizations, a different collaboration diagram will be required to show the handling of each system event message. For example (Figure 17.1):

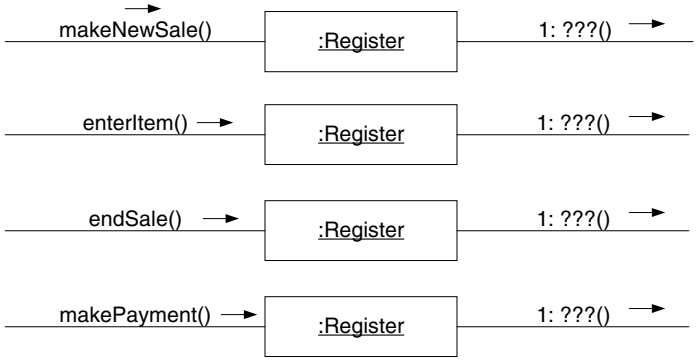

Figure 17.1 Collaboration diagrams and system event message handling.

On the other hand, if sequence diagrams are used, it *may* be possible to fit all system event messages on the same diagram, as in Figure 17.2.

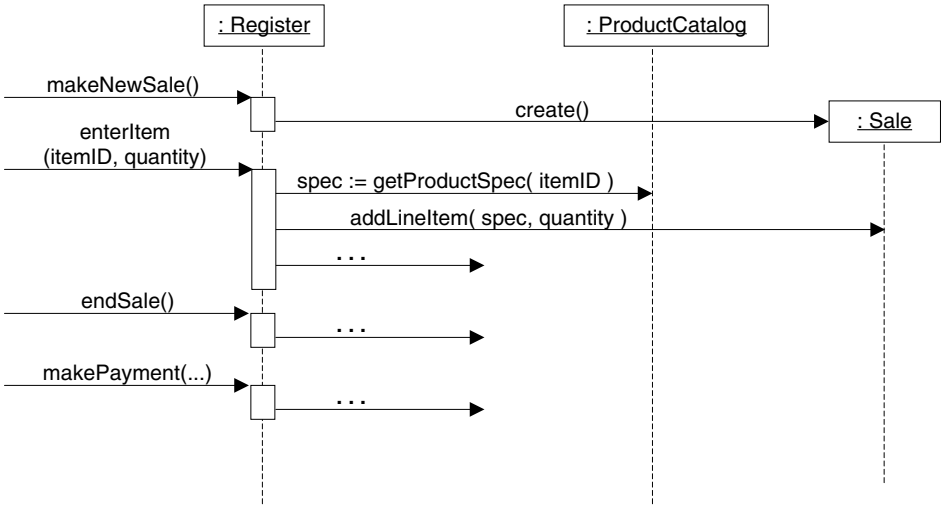

Figure 17.2 One sequence diagram and system event message handling.

However, it is often the case that the sequence diagram is then too complex or long. It is legal, as with interaction diagrams, to use a sequence diagram for each system event message, as in Figure 17.3.

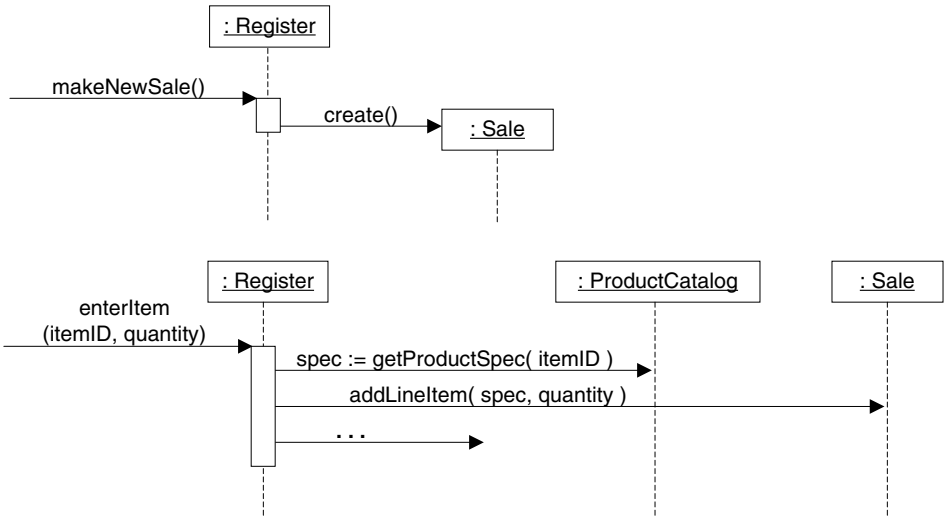

Figure 17.3 Multiple sequence diagrams and system event message handling.

Contracts and Use-Case Realizations

To reiterate, it may be possible to design use-case realizations directly from the use case text. In addition, for some system operations, contracts may have been written that add greater detail or specificity. For example:

Contract CO2: enterItem

Operation:	enterItem(itemID : ItemID, quantity : integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	– A SalesLineItem instance sli was created (instance creation). – ...

In conjunction with contemplating the use case text, for each contract, we work through the postcondition state changes and design message interactions to satisfy the requirements. For example, given this partial *enterItem* system opera-

tion, a partial interaction diagram is shown in Figure 17.4 that satisfies the state change of *SalesLineItem* instance creation.

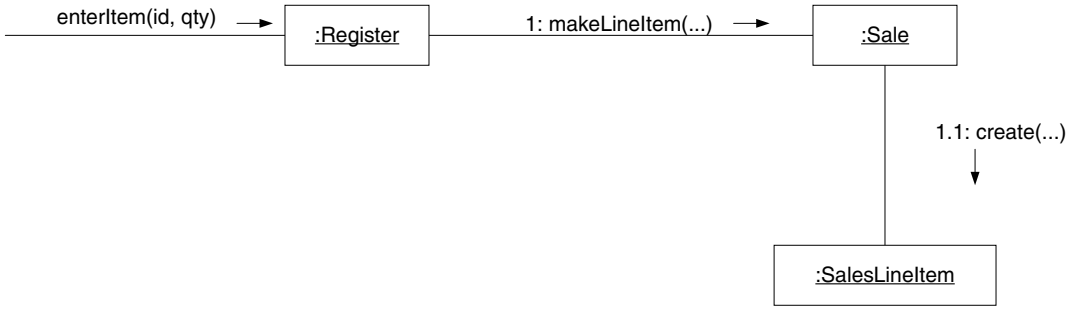

Figure 17.4 Partial interaction diagram.

Caution: The Requirements Are Not Perfect

It is useful to bear in mind that previously written use cases and contracts are only a guess of what must be achieved. The history of software development is one of invariably discovering that the requirements are not perfect, or have changed. This is not an excuse to ignore trying to do a good requirements job, but a recognition of the need to continuously engage customers and subject matter experts in review and feedback on the growing system's behavior.

An advantage of iterative development is that it naturally supports the discovery of new analysis and design results during design and implementation work. The spirit of iterative development is to capture a "reasonable" degree of information during requirements analysis, filling in details during design and implementation.

The Domain Model and Use-Case Realizations

Some of the software objects that interact via messages in the interaction diagrams are inspired from the Domain Model, such as a *Sale* conceptual class and *Sale* design class. The choice of appropriate responsibility placement using the GRASP patterns relies, in part, upon information in the Domain Model. As mentioned, the existing Domain Model is not likely to be perfect; errors and omissions are to be expected. You will discover new concepts that were previously missed, ignore concepts that were previously identified, and do likewise with associations and attributes.

Conceptual vs. Design Classes

Recall that the UP Domain Model does not illustrate software classes, but may be used to inspire the presence and names of some software classes in the

Design Model. During interaction diagramming or programming, the developers may look to the Domain Model to name some design classes, thus creating a design with lower representational gap between the software design and our concepts of the real domain to which the software is related (see Figure 17.5).

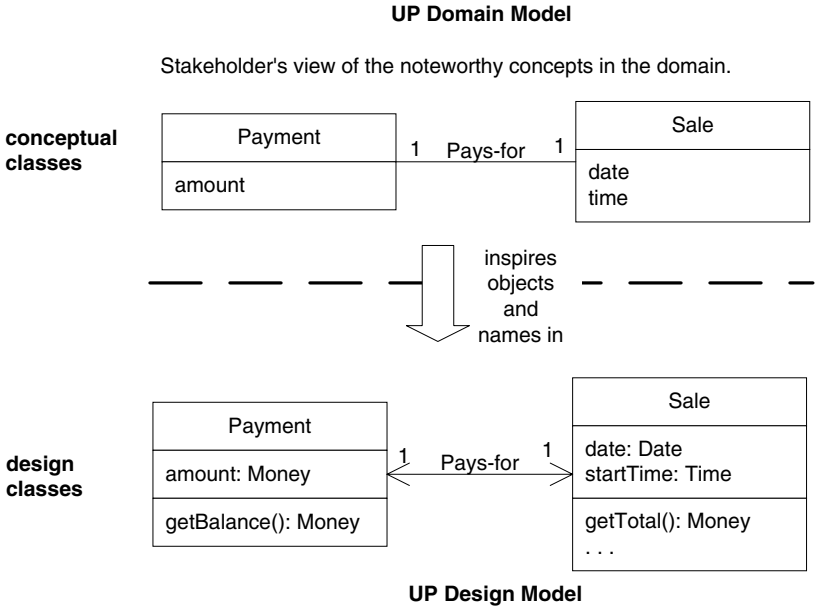

Figure 17.5 Lowering representational gap with design classes named from conceptual classes.

Must the design classes in the Design Model be limited to classes with names inspired from the Domain Model? Not at all; it is appropriate to discover new conceptual classes during this design work that were missed during earlier domain analysis, and also to make up software classes whose names and purpose is completely unrelated to the Domain Model.

17.3 Use-Case Realizations for the NextGen Iteration

The following sections explore the choices and decisions made while designing a use-case realization with objects based on the GRASP patterns. The explanations are intentionally detailed, in an attempt to illustrate that there does not have be any "hand waving" in the creation of well-designed interaction diagrams; their construction is based on justifiable principles.

Notationally, the design of objects for each system event message will be shown in a separate diagram, to focus on the design issues of each. However, they could have been grouped together on one sequence diagram.

17.4 Object Design: makeNewSale

The *makeNewSale* system operation occurs when a cashier requests to start a new sale, after a customer has arrived with things to buy. The use case may have been sufficient to decide what was necessary, but for this case study we wrote contracts for all the system events, for explanation and completeness.

Contract CO1: makeNewSale

Operation:	makeNewSale()
Cross References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	– A Sale instance s was created (instance creation). – s was associated with the Register (association formed). – Attributes of s were initialized.

Choosing the Controller Class

Our first design choice involves choosing the controller for the system operation message *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, ProcessSaleSession*

Choosing a facade controller like *Register* is satisfactory if there are only a few system operations and the facade controller is not taking on too many responsibilities (in other words, if it is becoming incohesive). Choosing a use-case controller is suitable when there are many system operations and we wish to distribute responsibilities in order to keep each controller class lightweight and focused (in other words, cohesive). In this case, *Register* will suffice, since there are only a few system operations.

This *Register* is a software object in the Design Model. It is not a real physical register but a software abstraction whose name was chosen to lower the representational gap between our concept of the domain and the software.

Thus, the interaction diagram shown in Figure 17.6 begins by sending the *makeNewSale* message to a *Register* software object.

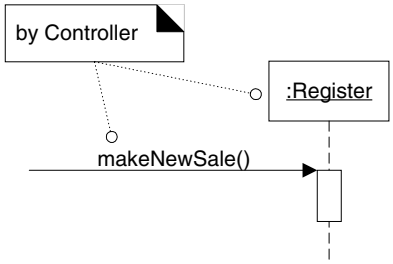

Figure 17.6 Applying the GRASP Controller pattern.

Creating a New Sale

A software *Sale* object must be created, and the GRASP Creator pattern suggests assigning the responsibility for creation to a class that aggregates, contains, or records the object to be created.

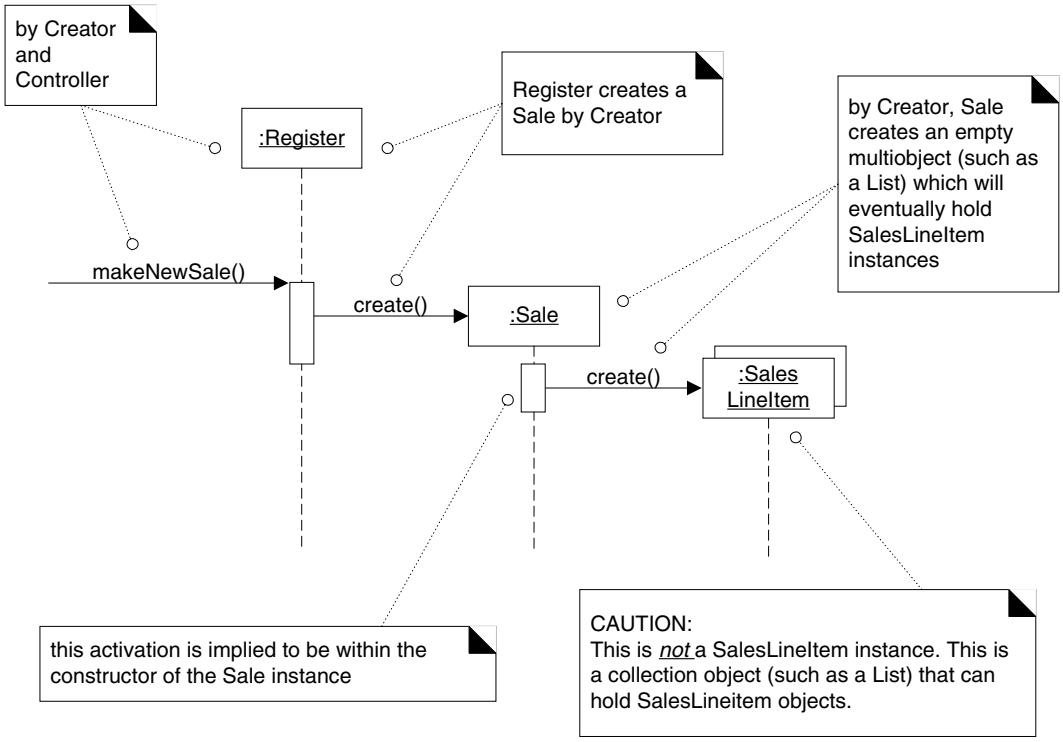

Figure 17.7 Sale and multiobject creation.

Analyzing the Domain Model reveals that a *Register* may be thought of as recording a *Sale*; indeed, the word "register" in business has for many years meant the thing that recorded (or registered) account transactions, such as sales.

Thus, *Register* is a reasonable candidate for creating a *Sale*. And by having the *Register* create the *Sale*, the *Register* can easily be associated with it over time, so that during future operations within the session, the *Register* will have a reference to the current *Sale* instance.

In addition to the above, when the *Sale* is created, it must create an empty collection (container, such as a Java *List*) to record all the future *SalesLineItem* instances that will be added. This collection will be contained within and maintained by the *Sale* instance, which implies by Creator that the *Sale* is a good candidate for creating it.

Therefore, the *Register* creates the *Sale*, and the *Sale* creates an empty collection, represented by a multiobject in the interaction diagram.

Hence, the interaction diagram in Figure 17.7 illustrates the design.

Conclusion

The design was not difficult, but the point of its careful explanation in terms of Controller and Creator was to illustrate that the details of a design can be rationally and methodically decided and explained in terms of principles and patterns, such as GRASP.

17.5 Object Design: enterItem

The *enterItem* system operation occurs when a cashier enters the *itemID* and (optionally) the quantity of something to be purchased. Here is the complete contract:

Contract CO2: enterItem

Operation:	enterItem(itemID : ItemID, quantity : integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	– A SalesLineItem instance sli was created (instance creation). – sli was associated with the current Sale (association formed). – sli.quantity became quantity (attribute modification). – sli was associated with a ProductSpecification, based on itemID match (association formed).

An interaction diagram will be constructed to satisfy the postconditions of *enterItem*, using the GRASP patterns to help with the design decisions.

Choosing the Controller Class

Our first choice involves handling the responsibility for the system operation message *enterItem*. Based on the Controller pattern, as for *makeNewSale*, we will continue to use *Register* as a controller.

Display Item Description and Price?

Because of a design principle called **Model-View Separation**, it is not the responsibility of non-GUI objects (such as a *Register* or *Sale*) to get involved in output tasks. Therefore, although the use case states that the description and price are displayed after this operation, the design will be ignored at this time.

All that is required with respect to responsibilities for the display of information is that the information is known, which it is in this case.

Creating a New SalesLineItem

The *enterItem* contract postconditions indicate the creation, initialization, and association of a *SalesLineItem*. Analyzing the Domain Model reveals that a *Sale* contains *SalesLineItem* objects. Taking inspiration from the domain, a software *Sale* may similarly contain software *SalesLineItem*. Hence, by Creator, a software *Sale* is an appropriate candidate to create a *SalesLineItem*.

The *Sale* can be associated with the newly created *SalesLineItem* by storing the new instance in its collection of line items. The postconditions indicate that the new *SalesLineItem* needs a quantity, when created; therefore, the *Register* must pass it along to the *Sale*, which must pass it along as a parameter in the *create* message (in Java, that would be implemented as a constructor call with a parameter).

Therefore, by Creator, a *makeLineItem* message is sent to a *Sale* for it to create a *SalesLineItem*. The *Sale* creates a *SalesLineItem*, and then stores the new instance in its permanent collection.

The parameters to the *makeLineItem* message include the *quantity*, so that the *SalesLineItem* can record it, and likewise the *ProductSpecification* which matches the itemID.

Finding a ProductSpecification

The *SalesLineItem* needs to be associated with the *ProductSpecification* that matches the incoming *itemID*. This implies it is necessary to retrieve a *ProductSpecification,* based on an *itemID* match.

Before considering *how* to achieve the lookup, it is useful to consider *who* should be responsible for it. Thus, a first step is:

> Start assigning responsibilities by clearly stating the responsibility.

To restate the problem:

> Who should be responsible for knowing a *ProductSpecification*, based on an *itemID* match?

This is neither a creation problem nor one of choosing a controller for a system event. Now we see our first application of Information Expert in the design.

In many cases, the Expert pattern is the principal one to apply. Information Expert suggests that the object that has the information required to fulfill the responsibility should do it. Who knows about all the *ProductSpecification* objects?

Analyzing the Domain Model reveals that the *ProductCatalog* logically contains all the *ProductSpecifications*. Once again, taking inspiration from the domain, we design software classes with similar organization: a software *ProductCatalog* will contain software *ProductSpecifications*.

With that decided, then by Information Expert *ProductCatalog* is a good candidate for this lookup responsibility since it knows all the *ProductSpecification* objects.

This may be implemented, for example, with a method called *getSpecification.*[1]

Visibility to a ProductCatalog

Who should send the *getSpecification* message to the *ProductCatalog* to ask for a *ProductSpecification*?

It is reasonable to assume that a *Register* and *ProductCatalog* instance were created during the initial *Start Up* use case, and that there is a permanent connection from the *Register* object to the *ProductCatalog* object. With that assump-

1. The naming of accessing methods is of course idiomatic to each language. Java always uses the *object.getFoo()* form, C++ tends to use *object.foo()*, and C# uses *object.Foo,* which hides (like Eiffel and Ada) if it is a method call or direct access of a public attribute. The Java style is used in the examples.

tion (which we might record on a task list of things to ensure in the design when we get to designing the initialization), then it is possible for the *Register* to send the *getSpecification* message to the *ProductCatalog*.

This implies another concept in object design: visibility. **Visibility** is the ability of one object to "see" or have a reference to another object.

> For an object to send a message to another object it must have visibility to it.

Since we will assume that the *Register* has a permanent connection—or reference—to the *ProductCatalog*, it has visibility to it, and hence can send it messages such as *getSpecification*.

The following chapter will explore the question of visibility more closely.

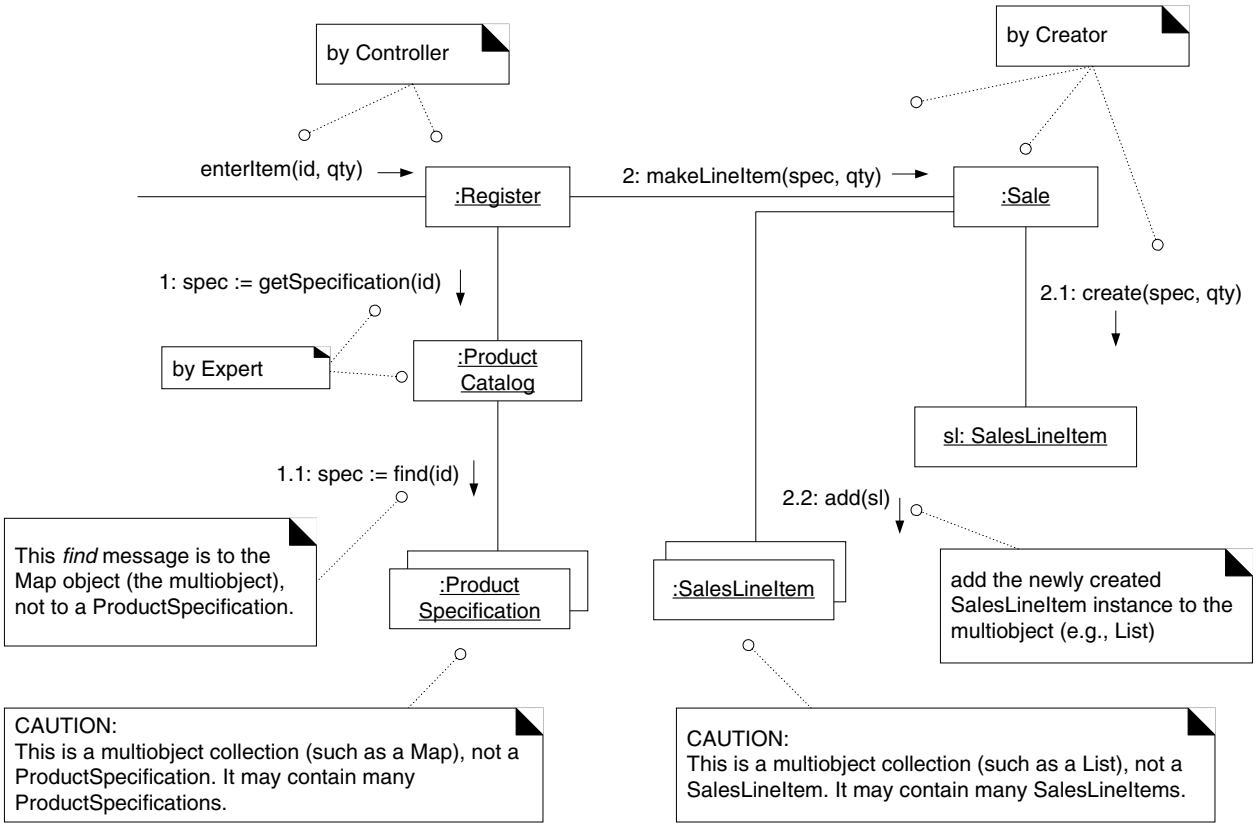

Figure 17.8 The enterItem interaction diagram.

Retrieving ProductSpecifications from a Database

In the final version of the NextGen POS application, it is unlikely that all the *ProductSpecifications* will actually be in memory. They will most likely be stored in a relational or object database and retrieved on demand; some may be cached in the client process for performance or fault-tolerance reasons. However, the issues surrounding retrieval from a database will be deferred for now in the interest of simplicity. It will be assumed that all the *ProductSpecifications* are in memory.

Chapter 34 explores the topic of database access of persistent objects, which is a large topic usually influenced by the choice of technologies, such as J2EE, .NET, and so forth.

The enterItem Object Design

Given the above discussion, the interaction diagram in Figure 17.8 reflects the decisions regarding the assignment of responsibilities and how objects should interact. Observe that considerable reflection was done to arrive at this design, based on the GRASP patterns; the design of object interactions and responsibility assignment require some deliberation.

Messages to Multiobjects

Notice that the interpretation of a message sent to a multiobject in the UML is that it is a message to the collection object itself, rather than an implicit broadcast to the collection's members. This is especially obvious for generic collection operations such as *find* and *add*.

For example, in the *enterItem* interaction diagram:

- The *find* message sent to the *ProductSpecification* multiobject is a message being sent once to the collection data structure represented by the multiobject (such as a Java *Map*).
    - The language-independent and generic *find* message will, during programming, be translated for a specific language and library. Perhaps it will actually be *Map.get* in Java. The message *get* could have been used in the diagram; *find* was used to make the point that design diagrams may require some mapping to different languages and libraries.
- The *add* message sent to the *SalesLineItem* multiobject is to add an element to the collection data structure represented by the multiobject (such as a Java *List*).

17.6 Object Design: endSale

The *endSale* system operation occurs when a cashier presses a button indicating the end of a sale. Here is the contract:

Contract CO3: endSale

Operation:	endSale()
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	Sale.isComplete became true (attribute modification).

Choosing the Controller Class

Our first choice involves handling the responsibility for the system operation message *endSale*. Based on the Controller GRASP pattern, as for *enterItem*, we will continue to use *Register* as a controller.

Setting the Sale.isComplete Attribute

The contract postconditions state:

- *Sale.isComplete* became *true* (attribute modification).

As always, Expert should be the first pattern considered unless it is a controller or creation problem (which it is not).

Who should be responsible for setting the *isComplete* attribute of the *Sale* to true?

By Expert, it should be the *Sale* itself, since it owns and maintains the *isComplete* attribute. Thus the *Register* will send a *becomeComplete* message to the *Sale* to set it to *true*.

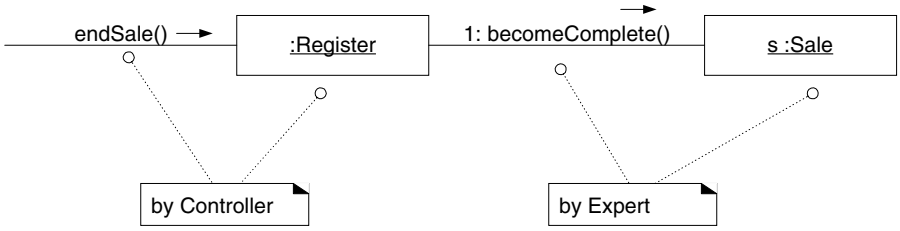

Figure 17.9 Completion of item entry.

UML Notation to Show Constraints, Notes, and Algorithms

Figure 17.9 shows the *becomeComplete* message, but does not communicate the details of what happens in the *becomeComplete* method (although it is admittedly trivial in this case). Sometimes in the UML we wish to use text to describe the algorithm of a method, or specify some constraint.

For these needs, the UML provides both **constraints** and **notes**. A UML constraint is some semantically meaningfully information attached to a model element. UML constraints are text enclosed in { } braces; for example, { x > 20 }. Any informal or formal language can be used for the constraint, and the UML especially includes the **OCL** (object constraint language) [WK99] if one desires to use that.

A UML note is a comment that has no semantic impact, such as date of creation or author.

A note is always shown in a **note box** (a dog-eared text box).

A constraint may be shown as simple text with braces, which is suitable for short statements. However, long constraints may be also placed within a "note box," in which case the so-called note box actually holds a constraint rather than a note. The text in the box is within braces, to indicate it is a constraint.

In Figure 17.10 both styles are used. Note that the simple constraint style (in braces but not in a box) just shows a statement which must hold true (the classic meaning of a constraint in logic). On the other hand, the "constraint" in the note box shows a Java method implementation of the constraint. Both styles are legal in the UML for a constraint.

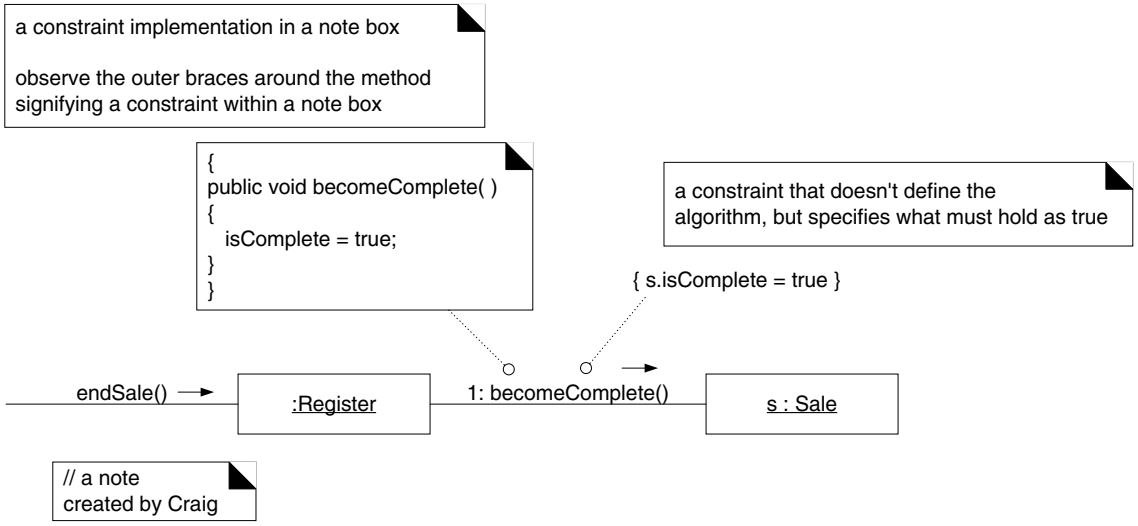

Figure 17.10 Constraints and notes.

Calculating the Sale Total

Consider this fragment of the *Process Sale* use case:

> **Main Success Scenario:**
> 1. Customer arrives ...
> 2. Cashier tells System to create a new sale.
> 3. Cashier enters item identifier.
> 4. System records sale line item and ...
> *Cashier repeats steps 3-4 until indicates done.*
> 5. System presents total with taxes calculated.

In step 5, a total is presented (or displayed). Because of the Model-View Separation principle, we should not concern ourselves with the design of how the sale total will be displayed, but it is necessary to ensure that the total is known. Note that no design class currently knows the sale total, so we need to create a design of object interactions that satisfies this requirement.

As always, Information Expert should be a pattern to consider unless it is a controller or creation problem (which it is not).

It is probably obvious the *Sale* itself should be responsible for knowing its total, but just to make the reasoning process to find an Expert crystal clear—with a simple example—please consider the following analysis.

1. State the responsibility:
    - Who should be responsible for knowing the sale total?
2. Summarize the information required:
    - The sale total is the sum of the subtotals of all the sales line-items.
    - sales line-item subtotal := line-item quantity * product description price
3. List the information required to fulfill this responsibility and the classes that know this information.

Information Required for Sale Total	Information Expert
ProductSpecification.price	*ProductSpecification*
SalesLineItem.quantity	*SalesLineItem*
all the *SalesLineItems* in the current Sale	*Sale*

A detailed analysis follows:

- Who should be responsible for calculating the *Sale* total? By Expert, it should be the *Sale* itself, since it knows about all the *SalesLineItem* instances whose subtotals must be summed to calculate the sale total. Therefore, *Sale* will have the responsibility of knowing its total, implemented as a *getTotal* method.
- For a *Sale* to calculate its total, it needs the subtotal for each *SalesLineItem*. Who should be responsible for calculating the *SalesLineItem* subtotal? By Expert, it should be the *SalesLineItem* itself, since it knows the quantity and the *ProductSpecification* it is associated with. Therefore, *SalesLineItem* will have the responsibility of knowing its subtotal, implemented as a *getSubtotal* method.
- For the *SalesLineItem* to calculate its subtotal, it needs the price of the *ProductSpecification*. Who should be responsible for providing the *ProductSpecification* price? By Expert, it should be the *ProductSpecification* itself, since it encapsulates the price as an attribute. Therefore, *ProductSpecification* will have the responsibility of knowing its price, implemented as a *getPrice* operation.

> Although the above analysis is trivial in this case, and the degree of excruciating elaboration presented is uncalled for in actual design practice, the same reasoning strategy to find an Expert can and should be applied in more difficult situations. You will find that once you learn these principles you can quickly perform this kind of reasoning mentally.

The Sale--getTotal Design

Given the above discussion, it is now desirable to construct an interaction diagram that illustrates what happens when a *Sale* is sent a *getTotal* message. The first message in this diagram is *getTotal*, but observe that the *getTotal* message is not a system event.

This leads to the following observation:

> Not every interaction diagram starts with a system event message; they can start with any message for which the designer wishes to show interactions.

The interaction diagram is shown in Figure 17.11. First, the *getTotal* message is sent to a *Sale* instance. The *Sale* will then send a *getSubtotal* message to each related *SalesLineItem* instance. The *SalesLineItem* will in turn send a *getPrice* message to its associated *ProductSpecifications*.

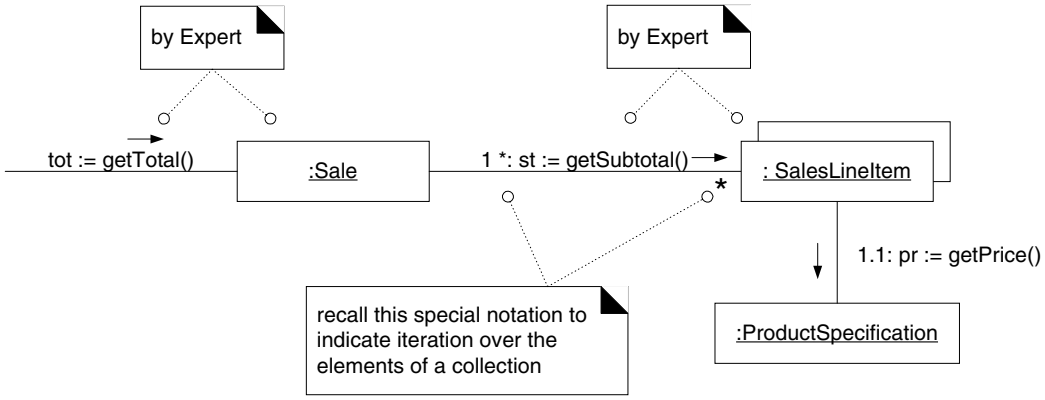

Figure 17.11 Sale--getTotal interaction diagram.

Since arithmetic is not (usually) illustrated via messages, the details of the calculations can be illustrated by attaching algorithms or constraints to the diagram that defines the calculations.

Who will send the *getTotal* message to the *Sale*? Most likely, it will be an object in the UI layer, such as a Java *JFrame*.

Observe in Figure 17.12 the use of algorithm notes and constraints, to communicate details of *getTotal* and *getSubtotal*.

17.7 Object Design: makePayment

The *makePayment* system operation occurs when a cashier enters the amount of cash tendered for payment. Here is the complete contract:

Contract CO4: makePayment

Operation:	makePayment(amount: Money)
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	– A Payment instance p was created (instance creation). – p.amountTendered became amount (attribute modification). – p was associated with the current Sale (association formed). – The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales).

A design will be constructed to satisfy the postconditions of *makePayment*.

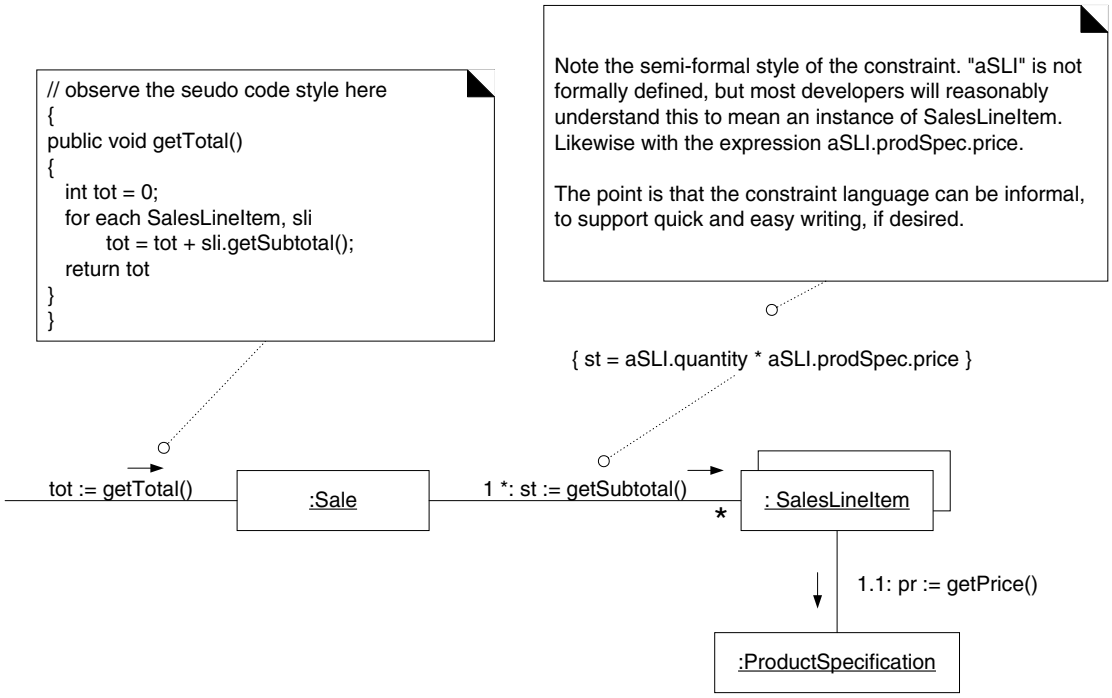

Figure 17.12 Algorithm notes and constraints.

Creating the Payment

One of the contract postconditions states:

- A *Payment* instance p was created (instance creation).

This is a creation responsibility, so the Creator GRASP pattern should be applied.

Who records, aggregates, most closely uses, or contains a *Payment*? There is some appeal in stating that a *Register* logically records a *Payment*, because in the real domain a "register" records account information, so it is a candidate by the goal of reducing the representational gap in the software design. Additionally, it is reasonable to expect that a *Sale* software will closely use a *Payment*; thus, it may be a candidate.

Another way to find a creator is to use the Expert pattern in terms of who is the Information Expert with respect to initializing data—the amount tendered in this case. The *Register* is the controller which receives the system operation *makePayment* message, so it will initially have the amount tendered. Consequently the *Register* is again a candidate.

In summary, there are two candidates:

- *Register*
- *Sale*

Now, this leads a key design idea:

> When there are alternative design choices, take a closer look at the cohesion and coupling implications of the alternatives, and possibly at the future evolution pressures on the alternatives. Choose an alternative with good cohesion, coupling, and stability in the presence of likely future changes.

Consider some of the implications of these choices in terms of the High Cohesion and Low Coupling GRASP patterns. If the *Sale* is chosen to create the *Payment*, the work (or responsibilities) of the *Register* is lighter—leading to a simpler *Register* definition. Also, the *Register* does not need to know about the existence of a *Payment* instance because it can be recorded indirectly via the *Sale*—leading to lower coupling in the *Register*. This leads to the design shown in Figure 17.13.

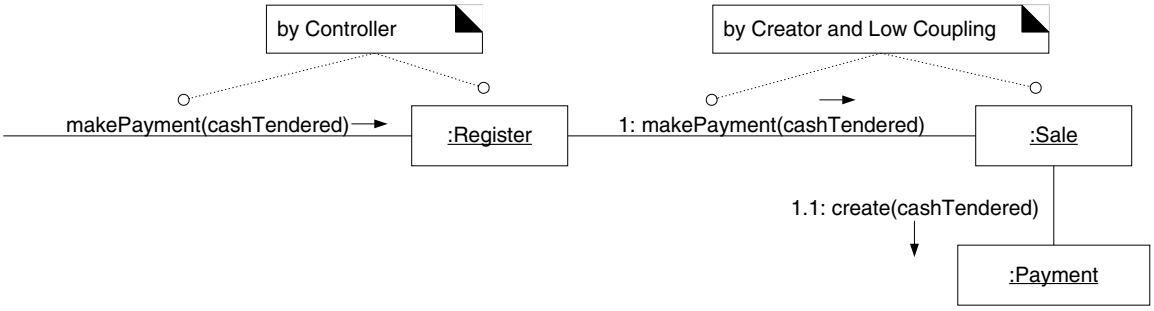

Figure 17.13 Register--makePayment interaction diagram.

This interaction diagram satisfies the postconditions of the contract: the *Payment* has been created, associated with the *Sale*, and its *amountTendered* has been set.

Logging a Sale

Once complete, the requirements state that the sale should be placed in an historical log. As always, Information Expert should be an early pattern considered unless it is a controller or creation problem (which it is not), and the responsibility should be stated:

> Who is responsible for knowing all the logged sales, and doing the logging?

By the goal of low representational gap in the software design (in relation to our concepts of the domain) it is reasonable for a *Store* to know all the logged sales, since they are strongly related to its finances. Other alternatives include classic accounting concepts, such as a *SalesLedger*. Using a *SalesLedger* object makes sense as the design grows and the *Store* becomes incohesive (see Figure 17.14).

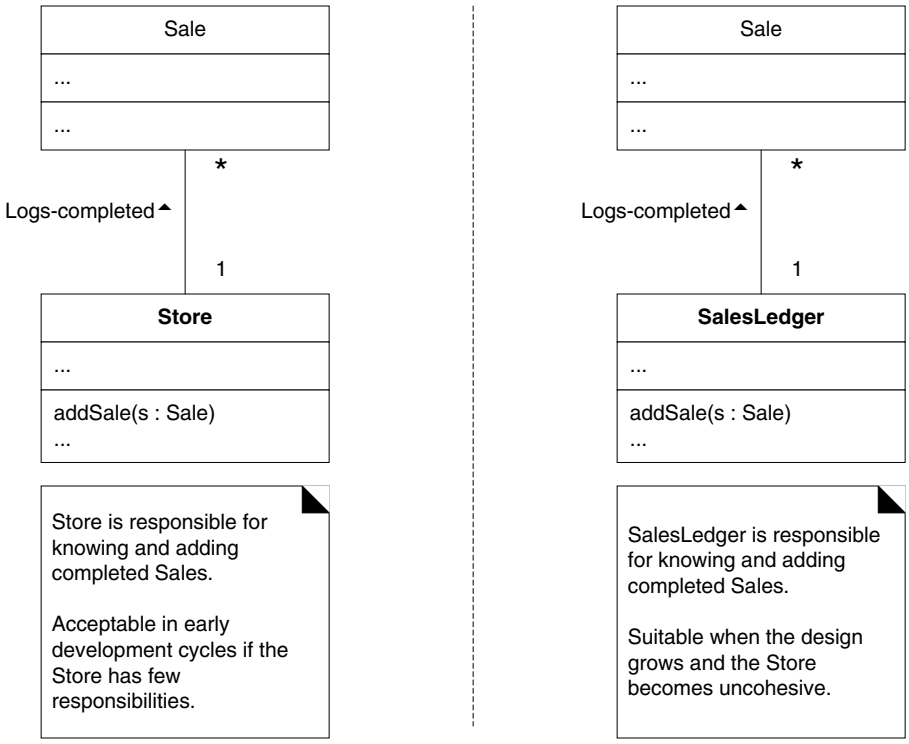

Figure 17.14 Who should be responsible for knowing the completed sales?

Note also that the postconditions of the contract indicate relating the *Sale* to the *Store*. This is an example where the postconditions may not be what we want to actually achieve in the design. Perhaps we didn't think of a *SalesLedger* earlier, but now that we have, we choose to use it instead of a *Store*. If this were the case, *SalesLedger* would ideally be added to the Domain Model as well, as it is a name of a concept in the real-world domain. This kind of discovery and change during design work is to be expected.

In this case, we will stick with the original plan of using the *Store* (see Figure 17.15).

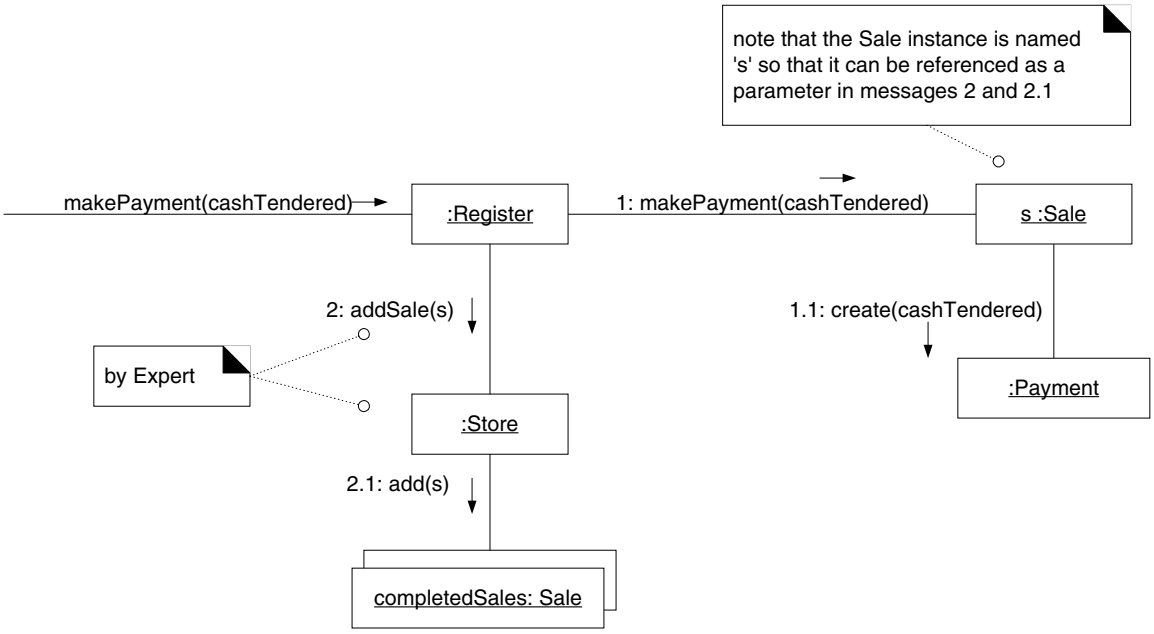

Figure 17.15 Logging a completed sale.

Calculating the Balance

The *Process Sale* use case implies that the balance due from a payment be printed on a receipt and displayed somehow.

Because of the Model-View Separation principle, we should not concern ourselves with how the balance will be displayed or printed, but it is necessary to ensure that it is known. Note that no class currently knows the balance, so we need to create a design of object interactions that satisfies this requirement.

As always, Information Expert should be considered unless it is a controller or creation problem (which it is not), and the responsibility should be stated:

> Who is responsible for knowing the balance?

To calculate the balance, the sale total and payment cash tendered are required. Therefore, *Sale* and *Payment* are partial Experts on solving this problem.

If the *Payment* is primarily responsible for knowing the balance, it would need visibility to the *Sale*, in order to ask the *Sale* for its total. Since it does not currently know about the *Sale*, this approach would increase the overall coupling in the design—it would not support the Low Coupling pattern.

In contrast, if the *Sale* is primarily responsible for knowing the balance, it needs visibility to the *Payment*, in order to ask it for its cash tendered. Since the *Sale*

already has visibility to the *Payment*—as its creator—this approach does not increase the overall coupling, and is therefore a preferable design.

Consequently, the interaction diagram in Figure 17.16 provides a solution for knowing the balance.

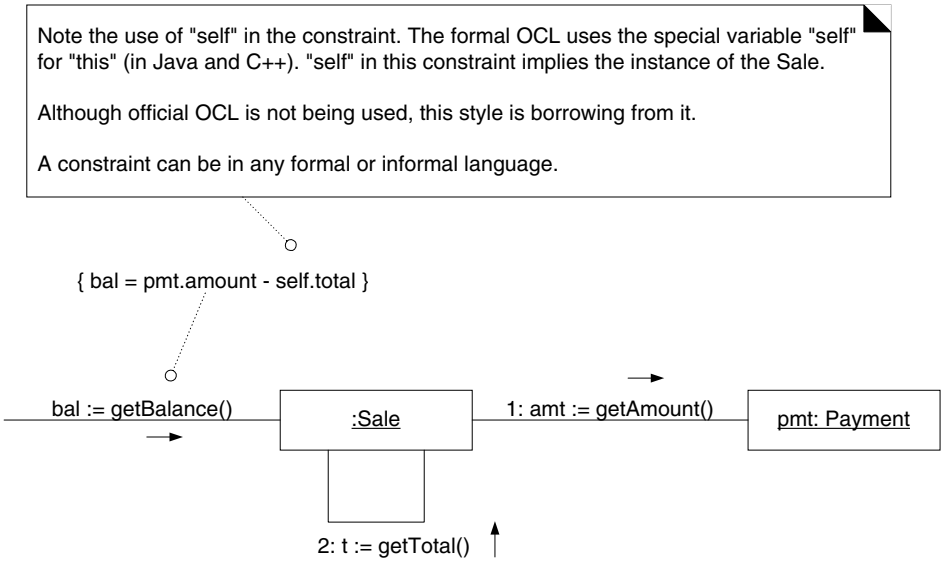

Figure 17.16 Sale--getBalance interaction diagram.

17.8 Object Design: startUp

When to Create the startUp Design?

Most, if not all, systems have a *Start Up* use case, and some initial system operation related to the starting up of the application. Although this *startUp* system operation is the earliest one to execute, delay the development of an interaction diagram for it until after all other system operations have been considered. This ensures that information has been discovered concerning the initialization activities required to support the later system operation interaction diagrams.

Do the initialization design last.

How Applications Start Up

The *startUp* operation abstractly represents the initialization phase of execution when an application is launched. To understand how to design an interaction diagram for this operation, it is helpful to understand the contexts in which initialization can occur. How an application starts and initializes is dependent on the programming language and operating system.

In all cases, a common design idiom is to ultimately create an **initial domain object**, which is the first software "domain" object created.

A note on terminology: As will be explored, applications are organized into logical layers that separate the major concerns of the application. These include a UI layer (for UI concerns) and a "domain" layer (for domain logic concerns). The domain layer of the Design Model is composed of software classes whose names are inspired from the domain vocabulary, and which contain application logic. Virtually all the design objects we have considered, such as *Sale* and *Register*, are domain objects in the domain layer of the Design Model.

The initial domain object, once created, is responsible for the creation of its direct child domain objects. For example, if a *Store* is chosen as the initial domain object, it may be responsible for the creation of a *Register* object.

The place where this initial domain object is created is dependent on the object technology chosen. For example, in a Java application, the *main* method may create it, or delegate the work to a *factory* object that creates it.

```
public class Main
{

public static void main( String[] args )
{
    // Store is the initial domain object.
    // The Store creates some other domain objects.

    Store store = new Store();

    Register register = store.getRegister();

    ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
    ...
}

}
```

Interpretation of the startUp System Operation

The preceding discussion illustrates that the *startUp* system operation is a language-independent abstraction. During design, there is variation in where the initial object is created, and whether or not it takes control of the process. The initial domain object does not usually take control if there is a GUI; otherwise, it often does.

The interaction diagrams for the *startUp* operation represent what happens when the initial problem domain object is created, and optionally what happens if it takes control. They do not include any prior or subsequent activity in the GUI layer of objects, if one exists.

Hence, the *startUp* operation may be reinterpreted as:

1. In one interaction diagram, send a *create()* message to create the initial domain object.
2. (optional) If the initial object is taking control of the process, in a second interaction diagram, send a *run* message (or something equivalent) to the initial object.

The POS Application startUp Operation

The *startUp* system operation occurs when a manager powers on the POS system and the software loads. Assume that the initial domain object is *not* responsible for taking control of the process; control will remain in the UI layer (such as a Java *JFrame*) after the initial domain object is created. Therefore, the interaction diagram for the *startUp* operation may be reinterpreted solely as a *create()* message sent to create the initial object.

Choosing the Initial Domain Object

What should the class of the initial domain object be?

Choose as an initial domain object a class at or near the root of the containment or aggregation hierarchy of domain objects. This may be a facade controller, such as *Register*, or some other object considered to contain all or most other objects, such as a *Store*.

Choosing between these alternatives may be influenced by High Cohesion and Low Coupling considerations. In this application, the *Store* is chosen as the initial object.

Persistent Objects: ProductSpecification

The *ProductSpecification* instances will reside in a persistent storage medium, such as relational or object database. During the *startUp* operation, if there are only a few of these objects, they may all be loaded into the computer's direct memory. However, if there are many, loading them all would consume too much

memory or time. Alternately—and more likely—individual instances will be loaded on demand into memory as they are required.

The design of how to dynamically on-demand load objects from a database into memory is simple if an object database is used, but difficult for a relational database. This problem is deferred for now and makes a simplifying assumption that all the *ProductSpecification* instances can be "magically" created in memory by the *ProductCatalog* object.

Chapter 34 explores the question of persistent objects and one way to load them into memory.

Store--create() Design

The tasks of creation and initialization derive from the needs of the prior design work, such as the design for handling *enterItem* and so on. By reflecting on the prior interaction designs, the following initialization work can be identified:

- A *Store, Register, ProductCatalog* and *ProductSpecifications* need to be created.
- The *ProductCatalog* needs to be associated with *ProductSpecifications*.
- *Store* needs to be associated with *ProductCatalog*.
- *Store* needs to be associated with *Register*.
- *Register* needs to be associated with *ProductCatalog*.

Figure 17.17 shows a design. The *Store* was chosen to create the *ProductCatalog* and *Register* by the Creator pattern. *ProductCatalog* was likewise chosen to create the *ProductSpecifications*. Recall that this approach to creating the specifications is temporary. In the final design, they will be materialized from a database, as needed.

UML notation: Observe that the creation of all the *ProductSpecification* instances and their addition to a container happens in a repeating section, indicated by the "*" following the sequence numbers.

An interesting deviation between modeling the real-world domain and the design is illustrated in the fact that the software *Store* object only creates *one Register* object. A real store may house *many* real registers or POS terminals. However, we are considering a software design, not real life. In our current requirements, our software *Store* only needs to create a single instance of a software *Register*.

> Multiplicity between classes of objects in the Domain Model and Design Model may not be the same.

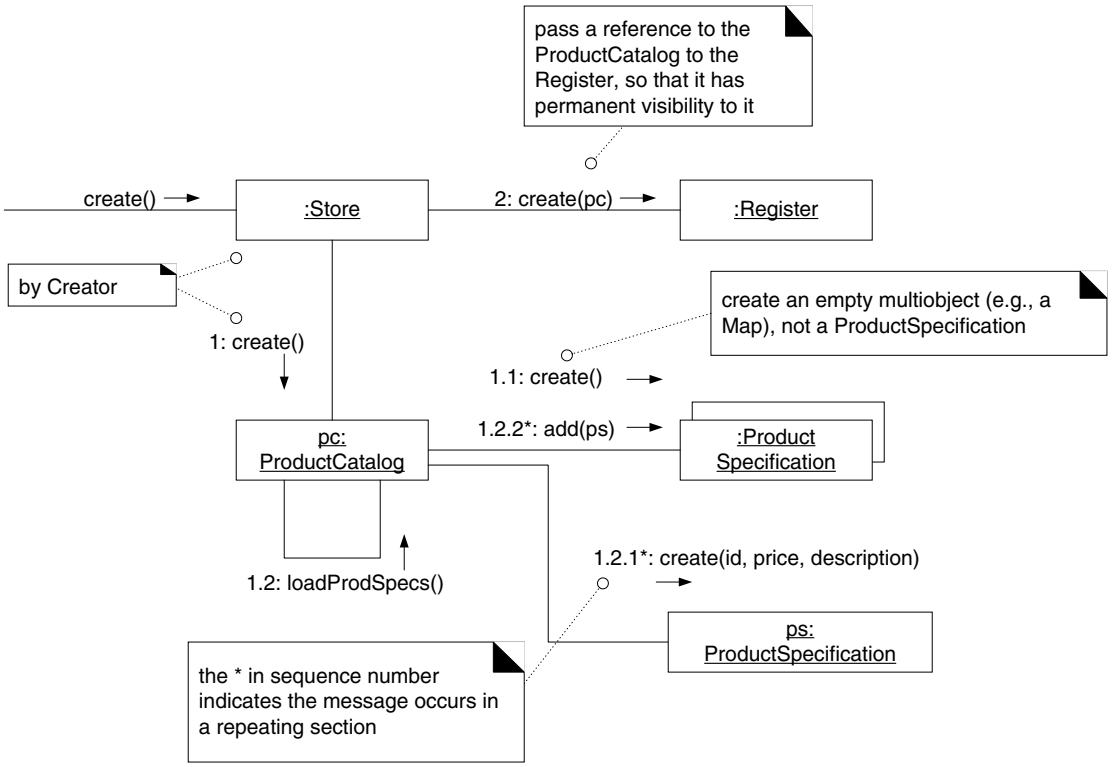

Figure 17.17 Creation of the initial domain object and subsequent objects.

17.9 Connecting the UI Layer to the Domain Layer

As has been briefly discussed, applications are organized into logical layers that separate the major concerns of the application, such as the UI layer (for UI concerns) and a "domain" layer (for domain logic concerns).

Common designs by which objects in the UI layer obtain visibility to objects in the domain layer include the following:

- An initializing routine (for example, a Java *main* method) creates both a UI and a domain object, and passes the domain object to the UI.
- A UI object retrieves the domain object from a well-known source, such as a factory object that is responsible for creating domain objects.

The sample code shown before is an example of the first approach:

```
public class Main
{
```

```
public static void main( String[] args )
{
    Store store = new Store();
    Register register = store.getRegister();
    ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
    ...
}

}
```

Once the UI object has a connection to the *Register* instance (the facade controller in this design), it can forward system event messages to it, such as the *enterItem* and *endSale* message (see Figure 17.18).

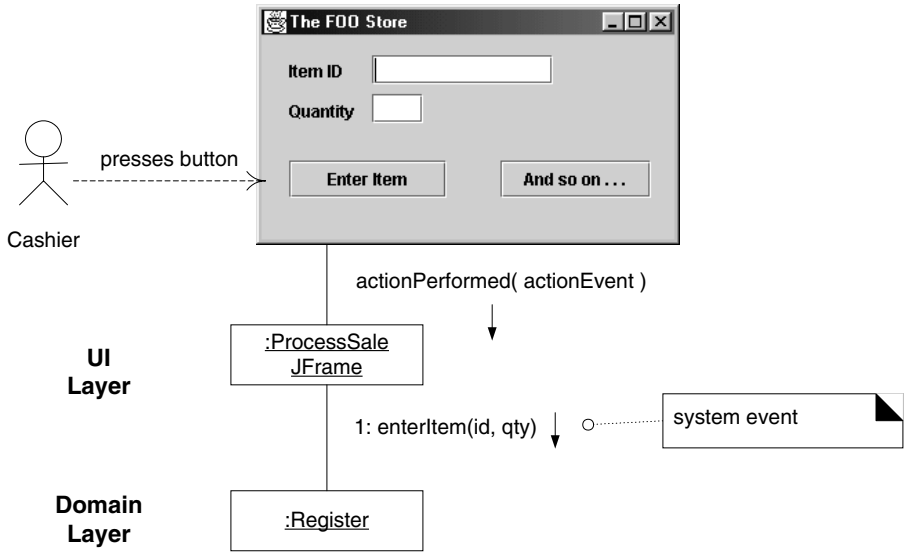

Figure 17.18 Connecting the UI and domain layers.

In the case of the *enterItem* message, the window needs to show the running total after each entry. There are several design solutions:

- Add a *getTotal* method to the *Register*. The UI sends the *getTotal* message to the *Register*, which forwards it to the *Sale*. This has the possible advantage of maintaining lower coupling from the UI to the domain layer—the UI only knows of the *Register* object. But it starts to expand the interface of the Register object, making it less cohesive.
- A UI asks for a reference to the current *Sale* object, and then when it needs the total (or any other information related to the sale), it directly sends messages to the *Sale*. This design increases the coupling from the UI to the domain layer. However, as was explored in the Low Coupling GRASP pattern discussion, higher coupling in and of itself is not a problem; rather, it is especially coupling to unstable things that is a problem. Assume we decide

the *Sale* is a stable object that will be an integral part of the design—which is very reasonable. Then, coupling to the *Sale* is not a problem.

As illustrated in Figure 17.19, this design follows the second approach.

Notice in these diagrams that the Java window (*ProcessSaleJFrame*), which is part of the UI layer, is not responsible for handling the logic of the application. It forwards requests for work (the system operations) to the domain layer, via the *Register*. This leads to the following design principle:

Interface and Domain Layer Responsibilities

The UI layer should not have any domain logic responsibilities. It should only be responsible for user interface tasks, such as updating widgets.

The UI layer should forward requests for all domain-oriented tasks on to the domain layer, which is responsible for handling them.

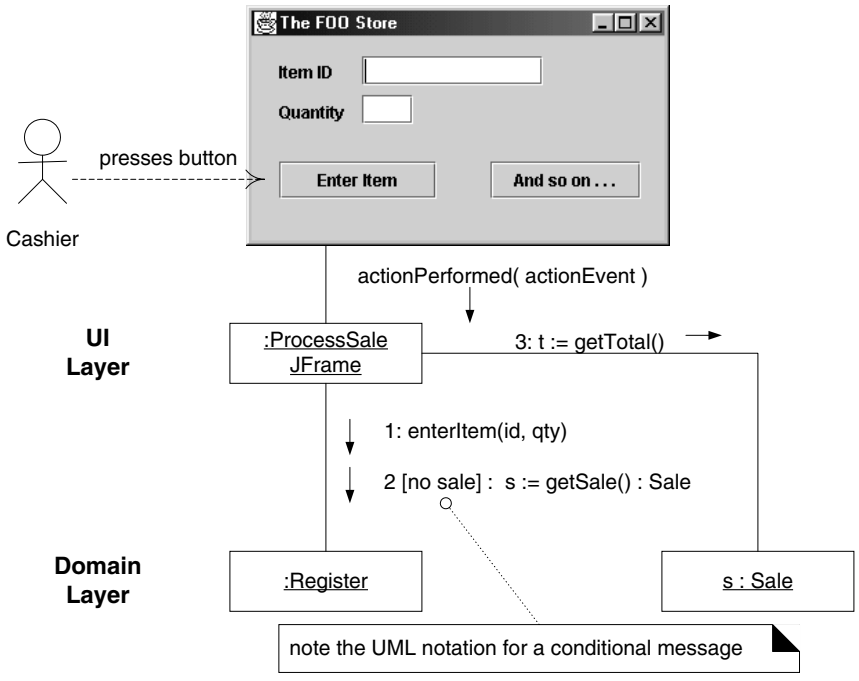

Figure 17.19 Connecting the UI and domain layers.

17.10 Use-Case Realizations Within the UP

Use-case realizations are part of the UP Design Model. This chapter has emphasized drawing interaction diagrams, but it is common and recommended to draw class diagrams in parallel. Class diagrams are examined in Chapter 19.

Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	***Design Model***		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 17.1 Sample UP artifacts and timing. s - start; r - refine

Phases

Inception—The Design Model and use-case realizations will not usually be started until elaboration because it involves detailed design decisions which are premature during inception.

Elaboration—During this phase, use-case realizations may be created for the most architecturally significant or risky scenarios of the design. However, UML diagramming will not be done for every scenario, and not necessarily in complete and fine-grained detail. The idea is to do interaction diagrams for the key use-case realizations that benefit from some forethought and exploration of alternatives, focusing on the major design decisions.

Construction—Use-case realizations are created for remaining design problems.

UP Artifacts and Process Context

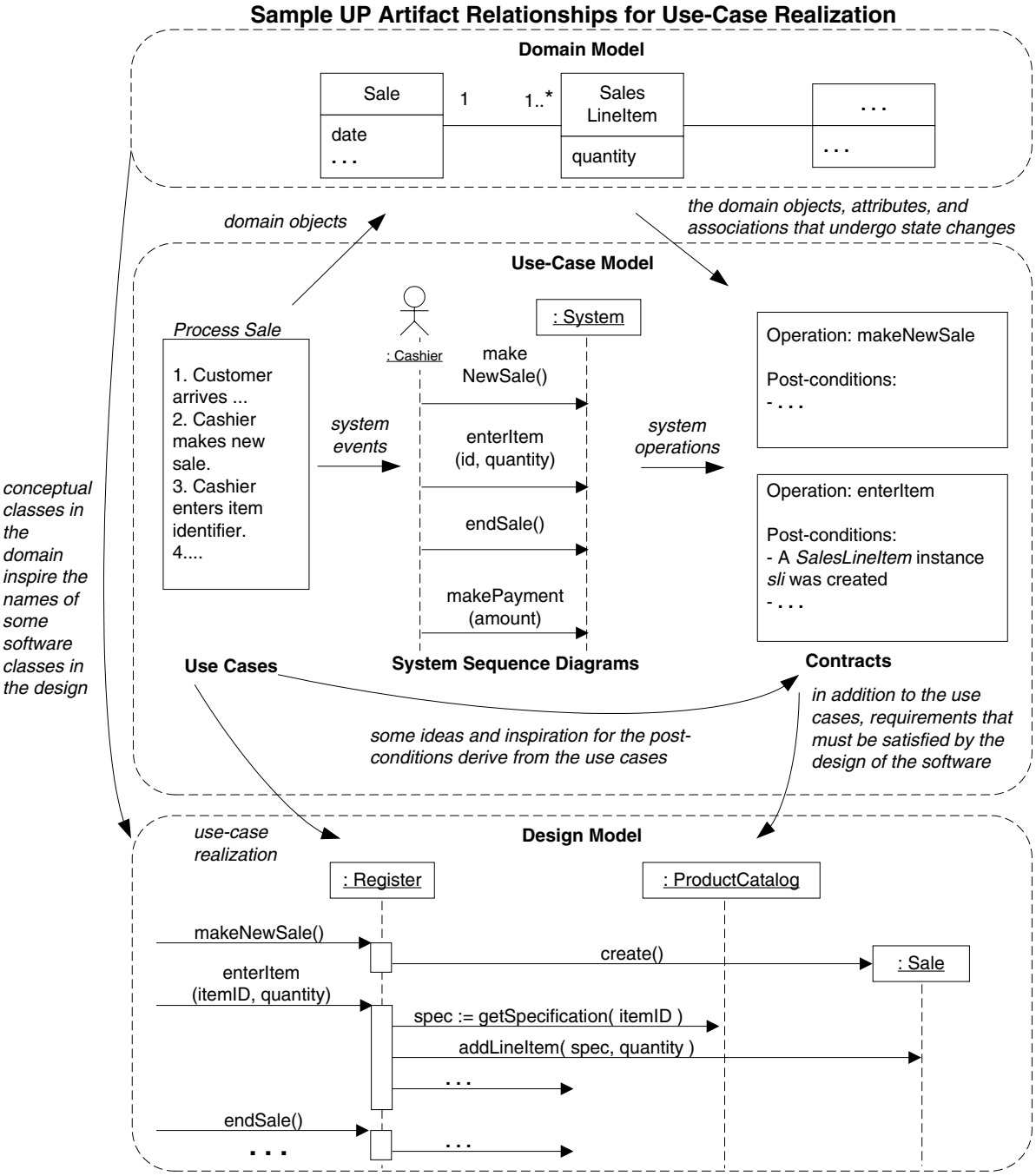

Figure 17.20 Sample UP artifact influence.

In the UP, use-case realization work is a design activity. Figure 17.21 offers suggestions on the time and space for doing this work.

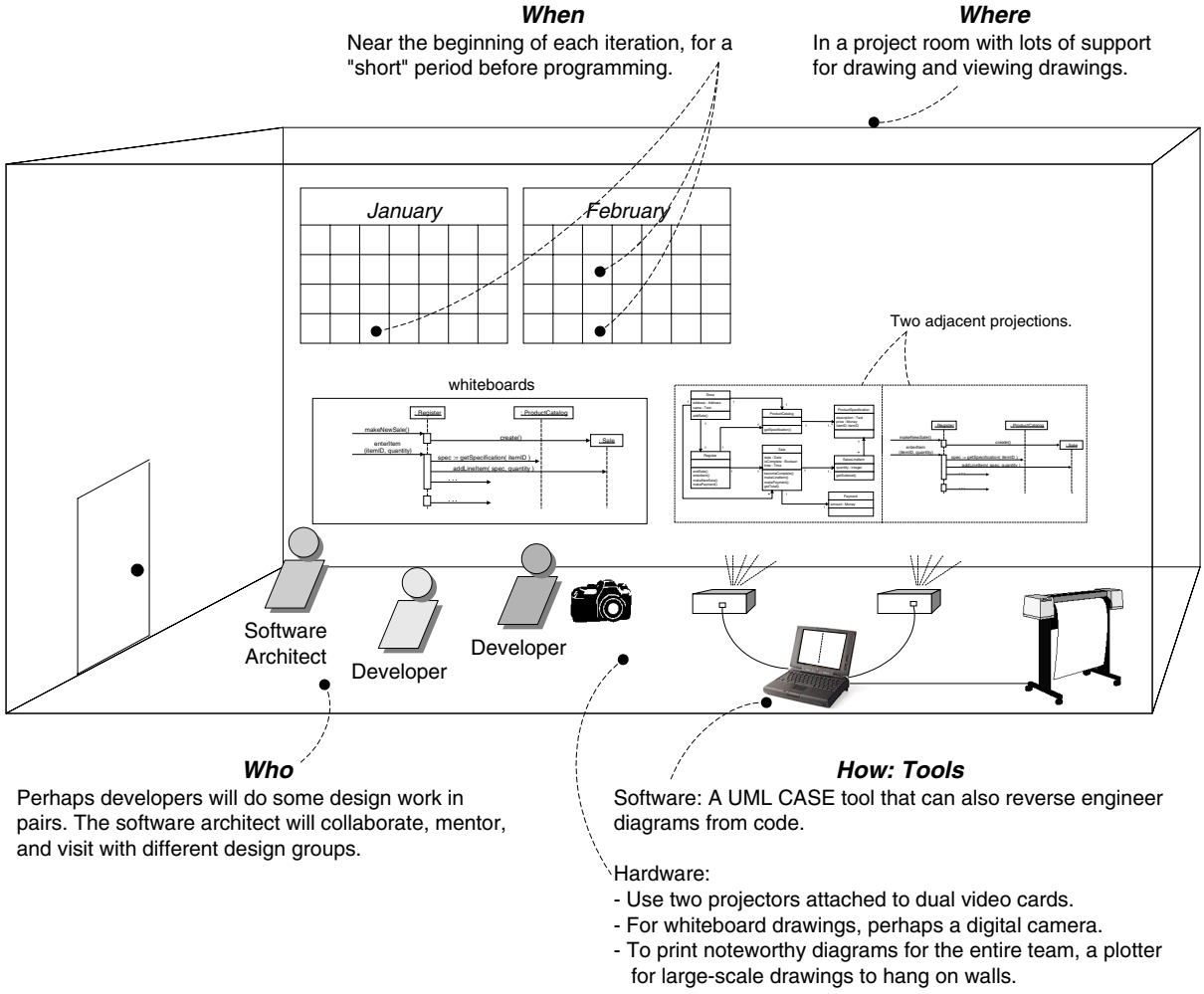

Figure 17.21 Sample process and setting context.

17.11 Summary

Designing object interactions and assigning responsibilities is at the heart of object design. These choices have can have a profound impact on the extensibility, clarity, and maintainability of an object software system, plus on the degree and quality of reusable components. There are principles by which the choices of responsibility assignment can be made; the GRASP patterns summarize some of the most general and common used by object-oriented designers.