CHAPTER 2: DESIGN PATTERNS

2.1 GRASP Patterns – Designing objects with Responsibilities

Object design is sometimes described as: - “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.”

The GRASP patterns are a learning aid to help one understand essential object design. This approach to understanding and using design principles is based on patterns of assigning responsibilities.

Responsibilities:

The UML defines a **responsibility as** "a contract or obligation of a classifier". 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:

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

Knowing responsibilities of an object include:

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

Responsibilities are assigned to classes of objects during object design.
For example**, "a Sale is responsible for creating SalesLineltems" (a doing), or "a Sale is responsible for knowing its total" (a knowing).**

Granularity of a responsibility:
The translation of problem domain responsibilities into classes and methods is influenced by the granularity of the responsibility. In other words, how much of the responsibility is done in a single interaction with the service.

Responsibilities and 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.

Responsibilities and Interaction Diagrams:

The purpose of GRASP patterns is to methodically apply fundamental principles for assigning responsibilities to objects.

Within the UML artifacts, responsibilities (implemented as methods) are considered during the creation of **interaction diagrams**

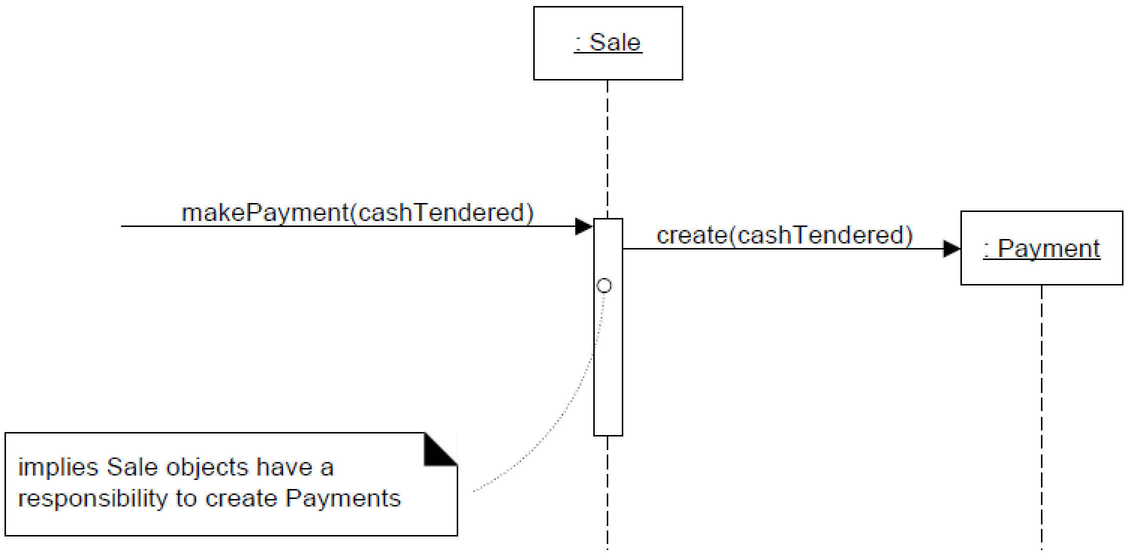

: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
implies Sale objects have a responsibility to create Payments

Figure 16.1 Responsibilities and methods are related.

Sale objects have been given a responsibility to create Payments, which is invoked with a makePayment message and handled with a corresponding makePayment method.

In summary, interaction diagrams show choices in assigning responsibilities to objects.

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.**

Definition of a pattern:

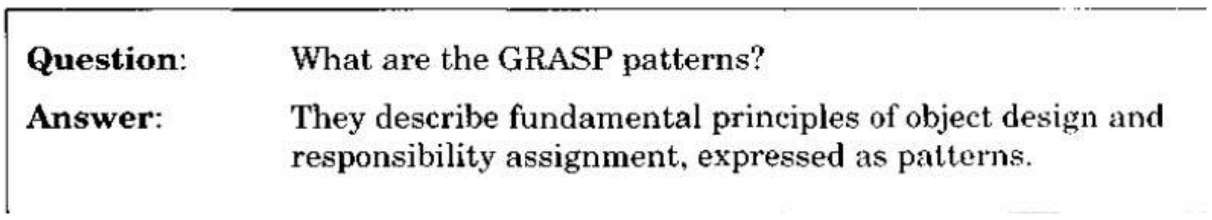
A **pattern** is a named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs.

Naming a pattern:

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.

GRASP patterns:



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

GRASP is an acronym that stands for **General Responsibility Assignment Software Patterns**

The following are the first five GRASP patterns:
. Information Expert
. Creator
. High Cohesion
. Low Coupling
. Controller

2.1.1 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?

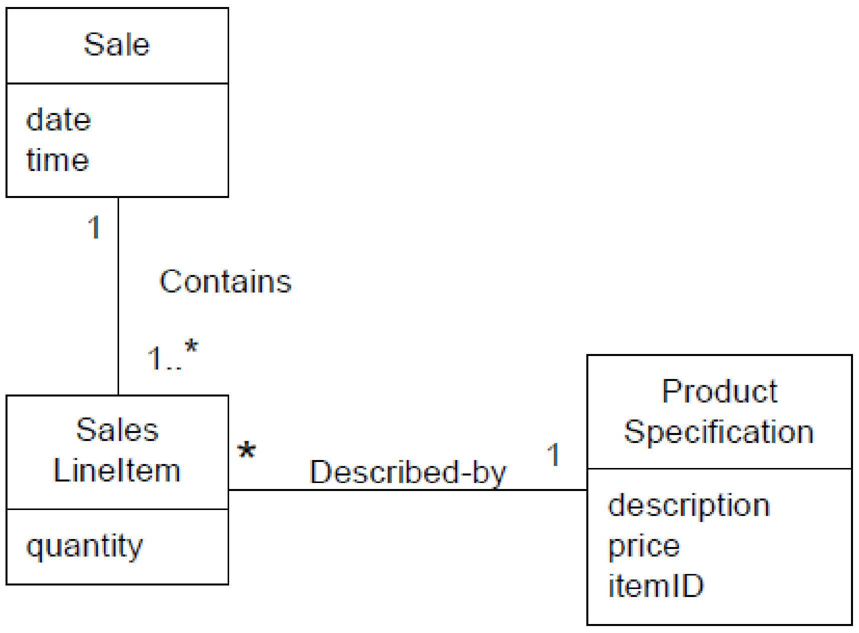
- ❒ A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled.
- ❒ During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes.

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.

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.
Do we look in the Domain Model or the Design Model to analyse the classes that have the information needed?
The Domain Model illustrates conceptual classes of the real-world domain;
The Design Model illustrates software 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.

To examine in detail, consider the partial Domain Model



16.3 Associations of Sale.

What information is needed to determine the grand total?
It is necessary to know about all the SalesLineItem instances of a sale and the sum of their subtotals.

A Sale instance contains these; therefore, Sale is a suitable class of object for this responsibility; it is an information expert for the work.

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.

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.
So the design is as follows:

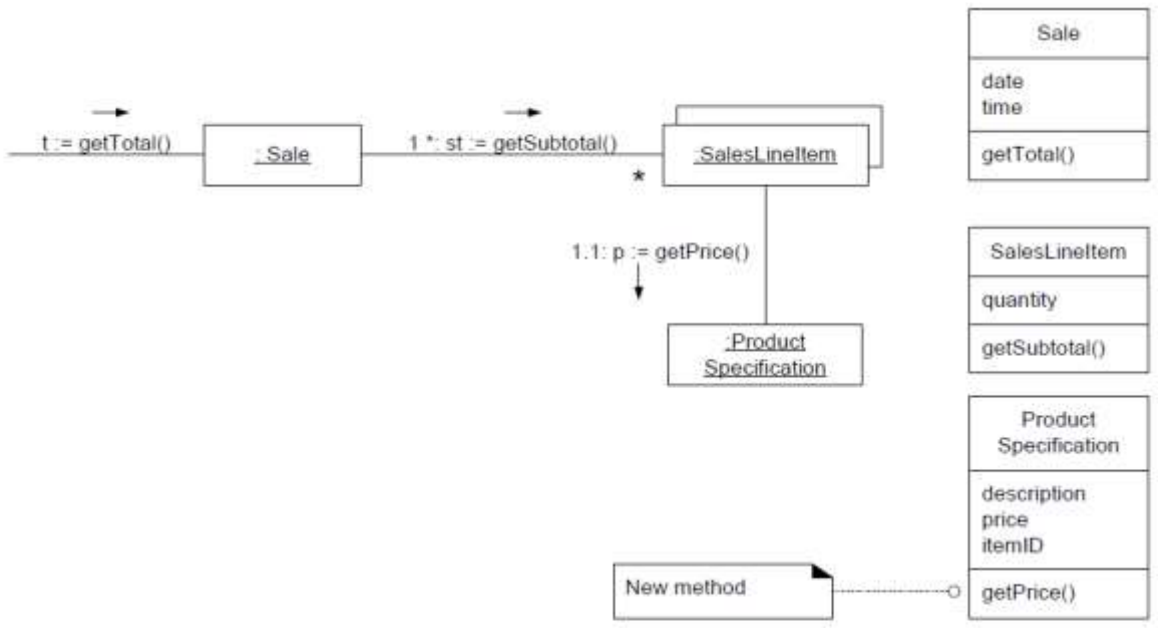


Figure 16.6 Calculating the *Sale* total.

Thus the set of information experts in NextGen POS system are:

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

Benefits:

- Information encapsulation is maintained, since objects use their own information to fulfill tasks.
- This usually supports low coupling, which leads to more robust and maintainable systems
- Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain.
- High cohesion is usually supported

Also Known AS: "That which knows, does," "Do It Myself,"

Contraindications: In some situations a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion.

For example, who should be responsible for saving a Sale in a database?

- € If Sale is responsible, and then each class has its own services to save itself in a database. The Sale class must now contain logic related to database handling, such as related to SQL and JDBC.

This will raises its coupling and duplicate the logic. The design would violate a separation of concerns – a basic architectural design goal.

2.1.2 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?

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

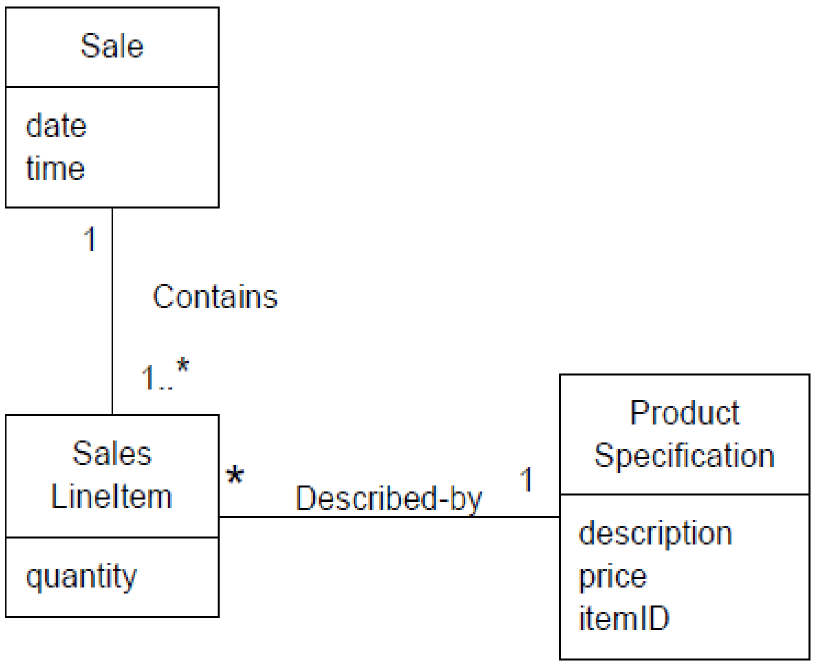


Figure 16.7 Partial domain model.

Since a Sate contains many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.

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

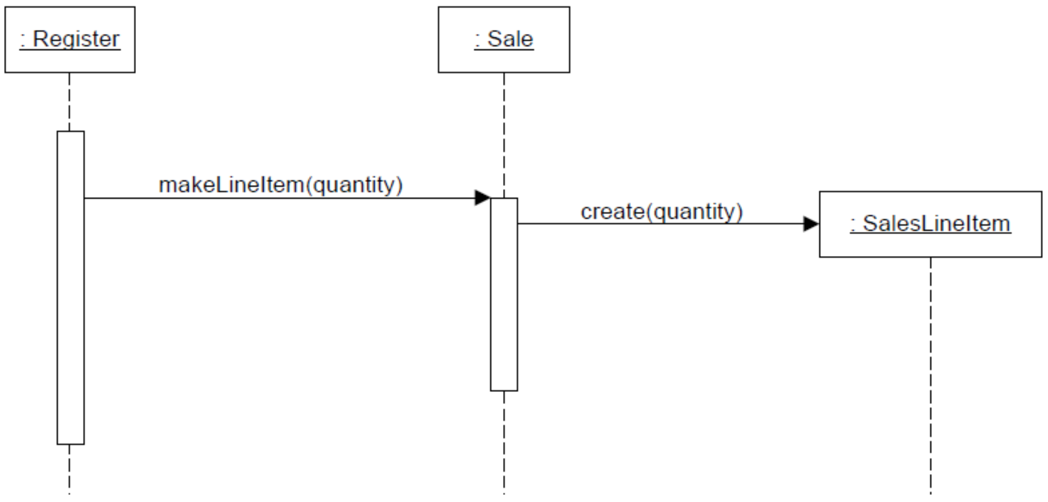
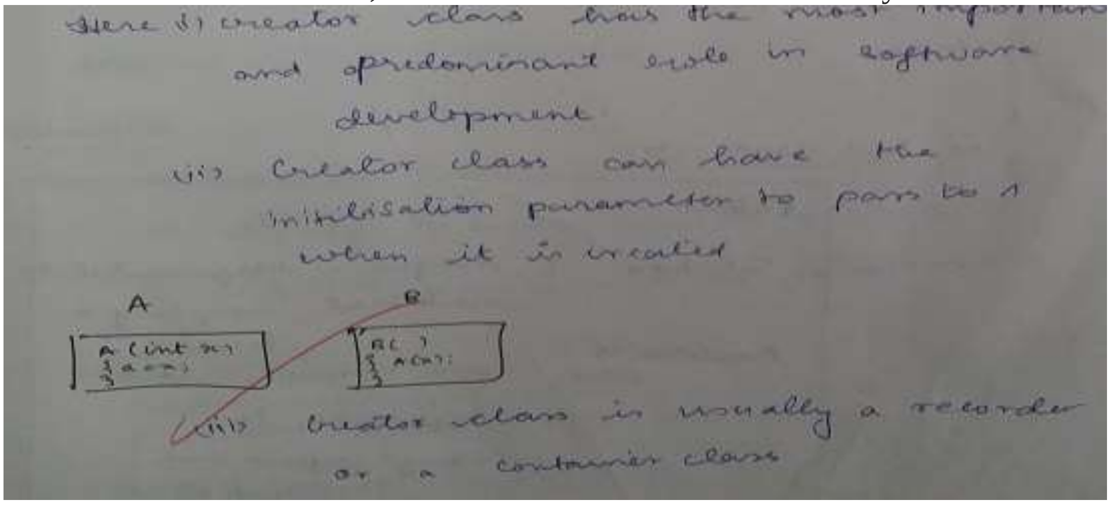


Figure 16.8 Creating a SalesLineItem.

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

Note:

Creator suggests that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded.
We identify a creator by looking for the class that has the initialization data that will be passed to constructor during creation.
For example, 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.



Here i) creator class has the most important and predominant role in software development

(ii) Creator class can have the initialisation parameter to pass to A when it is created

A

```
A(int x)
{ a=x;
}
```

B

```
B()
{ A(x);
}
```

(iii) Creator class is usually a recorder or a container class

Contraindications:

a) Creation requires significant complexity,
b) It is advisable to delegate creation to a helper class called a Factory rather than use the class suggested by Creator.

Benefits:

Low coupling (described next) is supported, which implies lower maintenance dependencies and higher opportunities for reuse.
Related patterns: "low Coupling" "factory"

2.1.3 Low Coupling

Solution	Assign a responsibility so that coupling remains low.
Problem	How to support low dependency, low change impact, and increased reuse?

Explanation:

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements

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

Payment
Register
Sale

Design 1:

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.

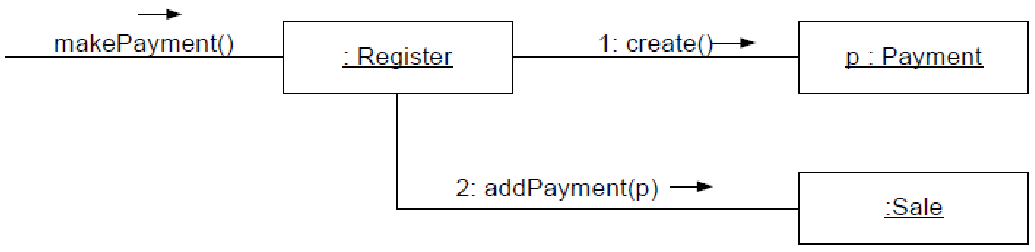


Figure 16.9 Register creates Payment.

Design 2:

An alternative solution to creating the Payment and associating it with the Sale is shown in diagram below

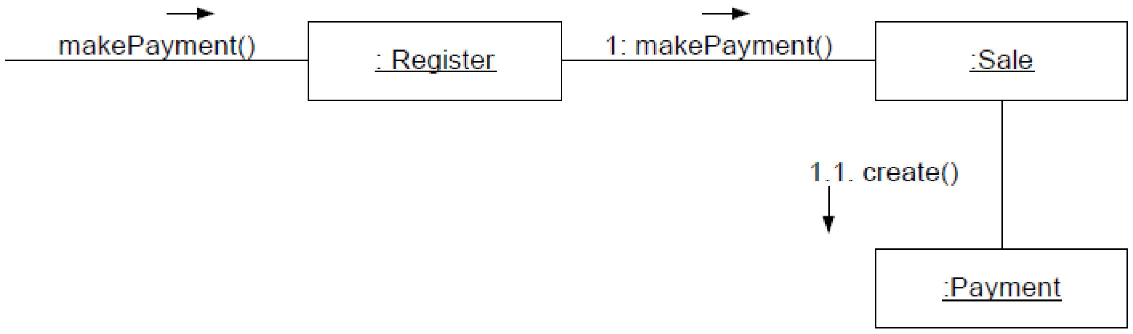


Figure 16.10 Sale creates Payment.

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.**
Discussion:

- ► 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:
  - o TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
  - o A TypeX object calls on services of a TypeY object.
  - o 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.
  - o TypeX is a direct or indirect subclass of TypeY.
- ► Low Coupling supports the design of classes that are more independent, which reduces the impact of change.

Benefits

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

Contraindications:

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.

Related principles: “protected Variation”

2.1.4 High Cohesion

Solution	Assign a responsibility so that cohesion remains high.
Problem	How to keep complexity manageable?

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

Example:

Consider the two designs for the NextGEN POS system

Design 1:

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,

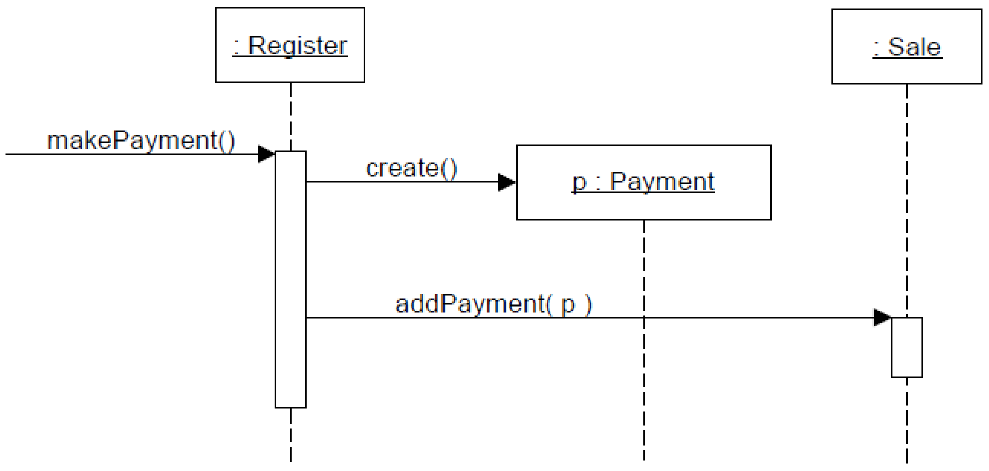


Figure 16.11 Register creates Payment.

Design 2:

The second design delegates the payment creation responsibility to the Sale, which supports higher cohesion.

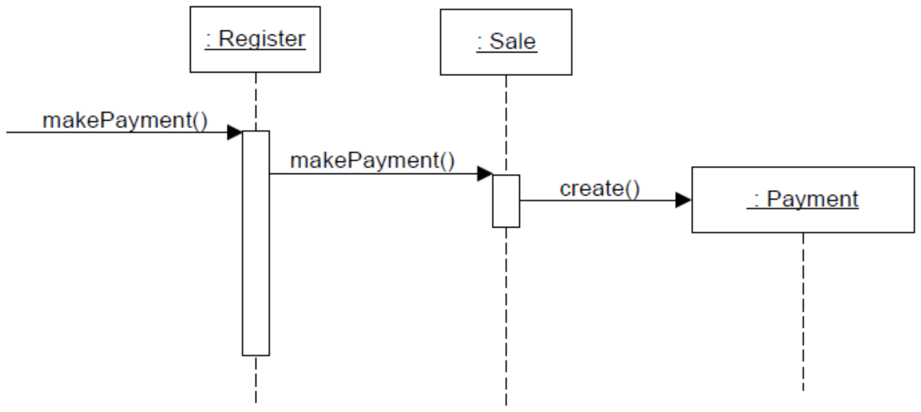


Figure 16.12 Sale creates Payment

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

Benefits:

A class with high cohesion is advantageous because

- ★ It is relatively easy to maintain, understand, and reuse.
- ★ The high degree of related functionality

★ The fine grain of highly related functionality also supports increased reuse potential.

Related principle – Modularity:
Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Contraindications:
There are a few cases where low cohesion is required:

1. 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.
2. 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
3. Few OO programmers may have strong SQL skills.
4. Suppose the SQL expert is not even a comfortable OO programmer.
5. 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.

2.1.5 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).*

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

Definition of Controller:
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.

Explanation:
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.

Example:

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."

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).

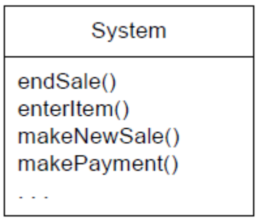
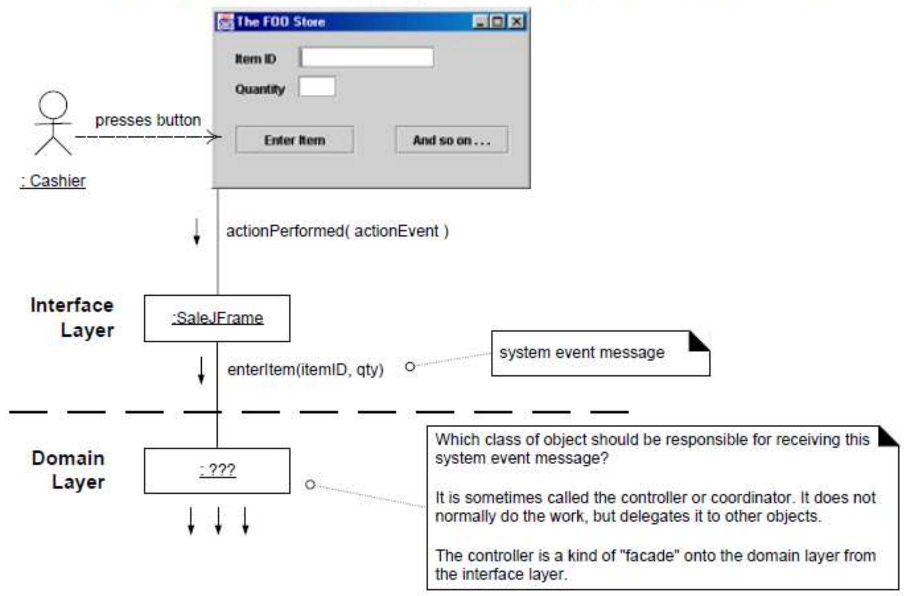


Figure 16.13 System operations associated with the system events.

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*

Discussion:

- ★ 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.
- ★ The first category of controller is a facade controller representing the overall system, device, or a subsystem.
- ★ **Boundary objects are** abstractions of the interfaces.

- ★ **Entity objects** are the application-independent domain software objects
- ★ C**ontrol objects are** use case handlers as described in this Controller pattern.
- ★ 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),

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.
- **Reason about the state of the use case -** It is sometimes necessary to ensure that system operations occur in a legal sequence.
    - For example, it may be necessary to guarantee that the makePay-ment operation cannot occur until the endSale operation has occurred.
