GRASP: Designing objects with responsibilities

4.1 INTRODUCTION

- One way to describe object design "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"
- Not really- Answer the following questions:
  - What methods belong to where?
  - How the objects should interact?
- GRASP as Methodical Approach to Learning Basic Object Design

UML versus Design Principles

- The UML is simply a standard visual modeling language, knowing its details doesn't teach you how to think in objects – that is the theme of this course
- The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology

UML versus Design Principles

- The UML is simply a standard visual modeling language, knowing its details doesn't teach you how to think in objects – that is the theme of this course
- The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology

Object Design

After the requirements identification, add the methods to the classes and define the message between the objects. The designing of object starts with

- **Inputs**
- **Activities**
- **Outputs**

Inputs to object design

- Use case model
- Domain Model
- System Sequence Diagrams
- Operation Contracts

- Supplementary Specification

Activities of object design

- Dynamic and static modeling (draw both interaction and complementary class diagrams)
- Applying various OOD principles
    - GRASP- General Responsibility Assignment Software Patterns
    - GoF (Gang of Four)design patterns
    - Responsibility-Driven Design (RDD)

Outputs of object design

- Modeling for the difficult part of the design that we wished to explore before coding
- Specifically for object design,
    - UML Interaction diagrams
    - Class diagrams
    - Package diagrams
- UI sketches and prototypes
- Database models Report sketches and prototypes

4.2 RDD

- RDD is a general metaphor for thinking about OO software design.
- Thinking of software objects as having responsibilities an abstraction of what they do. Responsibility means a contract or obligation of a classifier.
- RDD is a general metaphor for thinking about object oriented design.
  Responsibilities are related to the obligation of an object in terms of its behavior
    - what an object should know?
    - what an object should do?

Responsibilities is of two types : Doing , Knowing

1. **Knowing responsibilities**:
    - knowing about private encapsulated data
    - knowing about related objects
    - knowing about things it can derive or calculate

2. **Doing responsibilities**:
    a. doing something itself, such as creating an object or doing a calculation
    b. initiating action in other objects
    c. controlling and coordinating activities in other objects.

assigning responsibilities to objects while coding or while modeling. Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities.
Responsibilities are implemented using methods

What are Patterns?

A pattern is a named description of a problem and solution that can be applied to new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-offs.
Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

2.1 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.

2.2 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 creation of interaction diagrams, and certainly during programming.

2.3 How to Apply the GRASP Patterns?

The following sections present the first five GRASP patterns:

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

4.3.1 Creator

Problem

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

One of the most common activities in object oriented system is creation of objects. General principle is applied for the assignment of creation responsibilities.

Design supports :

1) low coupling
2) increased clarity
3) encapsulation
4) reusability.

Solution

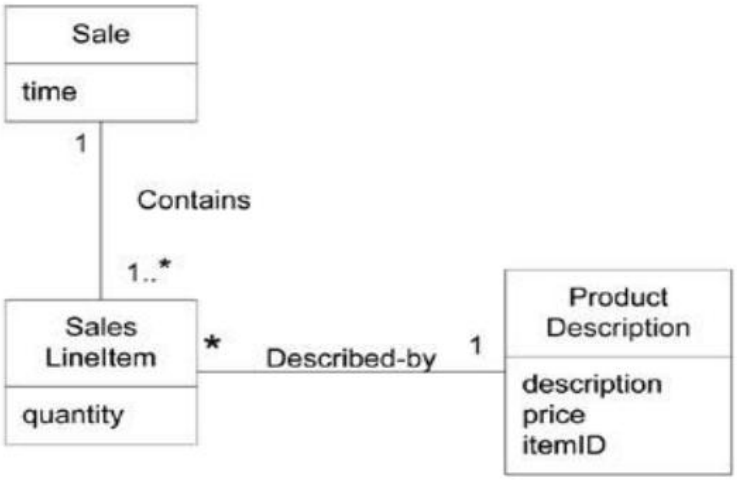
Assign class B the responsibility to create an instance of class A if one of these is true

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B is an expert while creating A (B passes the initializing data for A that is passed to A when created.)

B is a creator of A objects. If more than one option applies, usually prefer a class B which aggregates or contains class A.

Example:

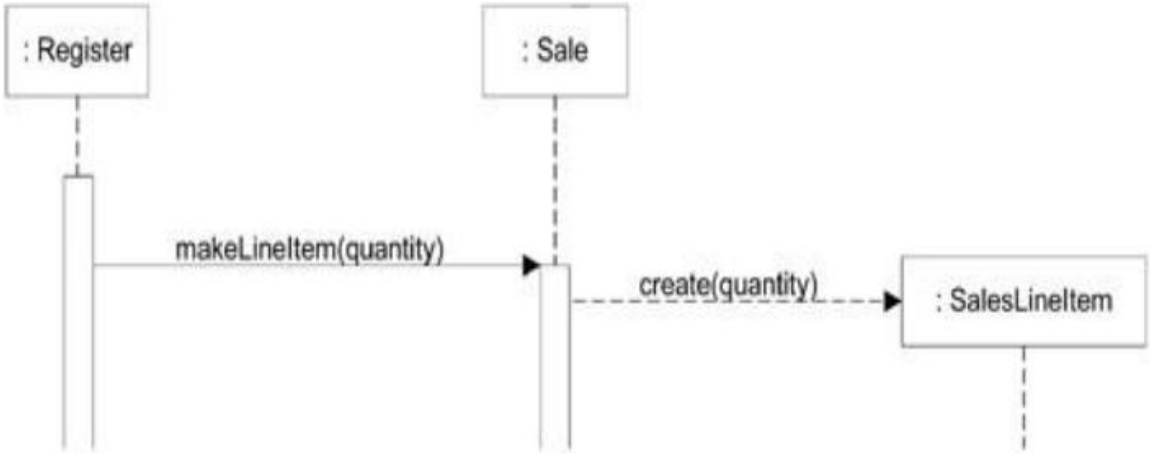
In the NextGen POS application, who should be responsible for creating a SalesLineItem instance? By Creator, we should look for a class that aggregates, contains, and so on, SalesLineItem instances. Consider the partial domain model in Figure



Partial Domain Model

Here "Sale " takes the responsibility of creating 'SalesLineItem' instance . Since sale contains many 'SalesLineItem' objects.

The assignment of responsibilities requires that a 'makeLineitem' must also be defined in 'Sale'.



Creating a SalesLineItem

Creator guides the assigning of 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.

All very common relationships between classes in a class diagram are,

- Composite aggregate part
- Container **contains** Content
- Recorder **records**

Creator suggests that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded.

Example:- 'Payment' instance while creation initialized with 'sale ' total. 'Sale' is a candidate creator of 'Payment'.

Contradictions:

Based upon some external property value, creation requires significant complexity like,

- Recycled instances for performances.
- Creating an instance from one of a family of similar classes based upon some external property value, etc.

In such cases we go for helper class called

- **Concrete Factory, and**
- **Abstract Factory**

Benefits of the creator

- Low coupling is supported which implies Lower maintenance and higher opportunities for reuse

4.3.2 INFORMATION EXPERT (OR EXPERT)

Problem

What is the general principle of assigning responsibilities to objects?

During Object Design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. This makes the software easier to

- Maintain
- Understand and
- Extend

Solution

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

Example:-

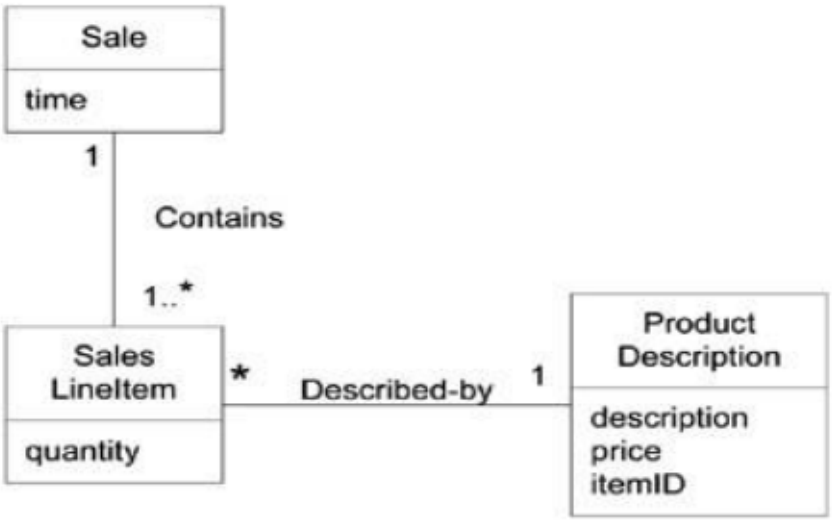
NextGEN POS Application

Some class needs to know the grand total of a sale. Start assigning responsibilities by clearly stating the responsibility.

1. Look at relevant classes in the Design Model if available,
2. Otherwise, look in the Domain Model

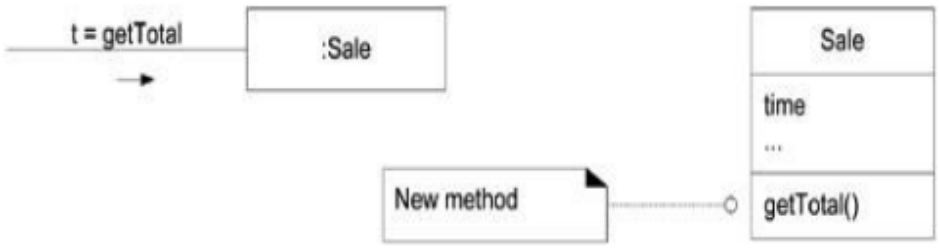
Example:-

If the design work has been just started, then look into the domain model, the real-world "sale". In design model, software class 'sale' is added with the responsibility for getting total with the method 'getTotal'.



Partial domain model for association of sale

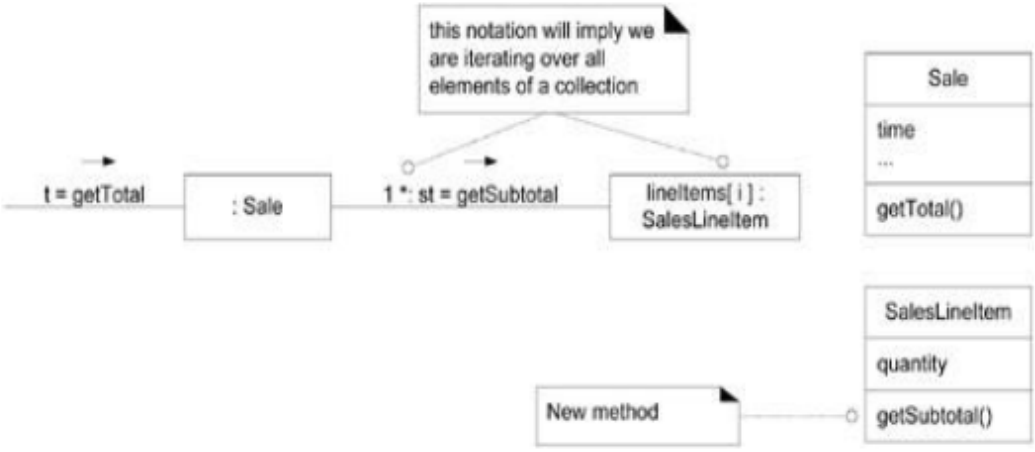
After adding the getTotal() , the partial interaction and class diagrams given as :



To determine expert using the above the SalesLineItem should determine subtotal.

- **SalesLineItem Quantity**
- **ProductDescription Price**

By information expert using the above the Sales LineItem should determine subtotal. This is done by Sale sending get Subtotal messages to each Sales LineItem and sum the results.

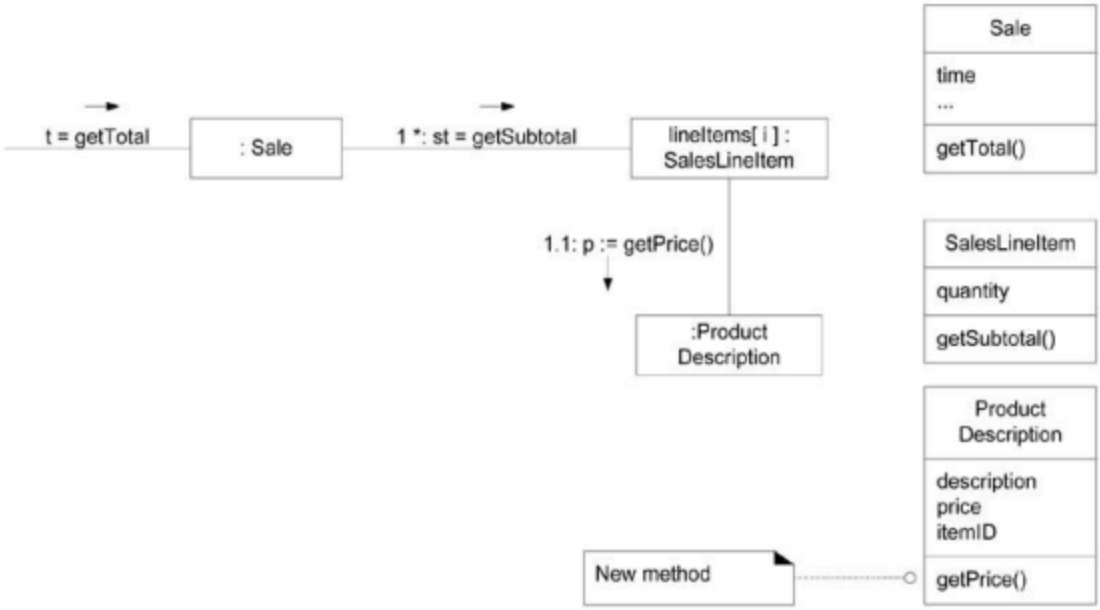


Calculating Sales Total

After knowing and answering subtotal , a SalesLineItem sends it a message asking for the product price. ProductDescription is an information expert on answering its price.

Calculating Sales Total

After knowing and answering subtotal , a SalesLineItem sends it a message asking for the product price. ProductDescription is an information expert on answering its price.



Calculating the sale total

Finally, we assigned three design classes of the objects with three responsibilities to find the sales total.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal

Design Class	Responsibility
ProductDescription	knows product price

- The Information Expert is frequently used in the assignment of responsibilities
- Experts express the common "intuition" that objects do things related to the information they have.
- Partial information experts will collaborate in the task.
- For example:- Sales total problem experts will collaborates in the task.
- Information expert thus has real world analogy.
- Information experts are basic guiding principle used continuously in object design.

Contradictions

Solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion.

To overcome this,

- Keep application logic in one place(like domain software objects)
- Keep database objects in another place(separate persistence services subsystem.
- Supporting a separation of major concerns improves coupling and cohesion in a design.

Benefits

- Information encapsulation is maintained since objects use their own information to fulfill tasks.
- High cohesion is usually supported

4.3.3 LOW COUPLING

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.

Types of coupling

Low coupling or weak coupling	High coupling or strong coupling
An element if does not depend on too many other elements like classes, subsystems and systems it is having low coupling.	A class with high coupling relies on many other classes. The Problem of high coupling are • Forced local changes because of changes in related classes. • Harder to understand in isolation. • Harder to reuse because its use requires the additional presence of the classes on which it is dependent

Problem :

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

Solution :

Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

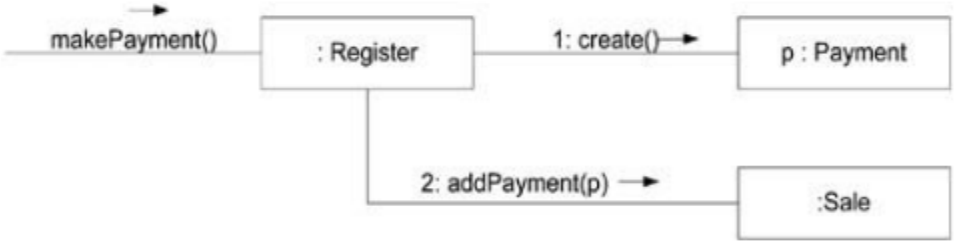
Example:-

NextGen case Study

We have to create payment instance and associate it with sale.

DESIGN 1: ***Suggested by creator***

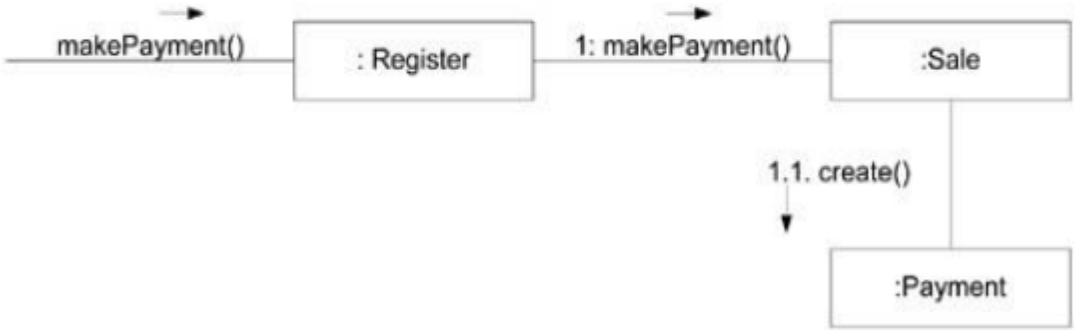
1) The Register creates the Payment and
2) It adds coupling of Register to Payment.



Register creates Payment

DESIGN 2: ***Suggested by low coupling***

1) The Sale does the creation of a Payment and
2) It does not increase the coupling.



Sales creates Payment

Low coupling is an evaluation principle for evaluating all designs decisions. In object-oriented languages such as C++, Java, and C#, common forms of coupling from TypeX to TypeY include the following:

- 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.

Contradictions

High coupling to stable elements and to pervasive elements is a problem. For example, a J2EE application can safely couple itself to the Java libraries

Benefits

- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse

4.3.4 CONTROLLER

Problem

What first object beyond the UI layer receives and coordinates ("controls") a system operation?

System operations were first explored during the analysis of SSD. These are the major input events upon our system.

Example:-

1) When a cashier using a POS terminal presses the "End Sale" button indicating "sale has ended".

2) When a writer using a word processor presses the "spell check" button to perform checking of spelling.

A **controller** is the first object beyond the User Interface (UI) layer that is responsible for receiving or handling a system operation message.

Solution

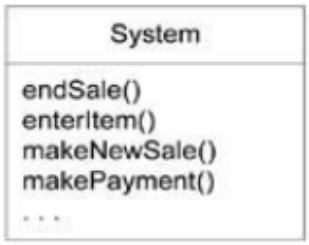
Assign the responsibility to one of the following

- Represents the overall "system," a "root object,"
  - These are all variations of a facade controller.

- Represents a use case scenario called
  - <UseCaseName>Handler,
  - <UseCaseName>Coordinator, or
  - <UseCaseName>Session

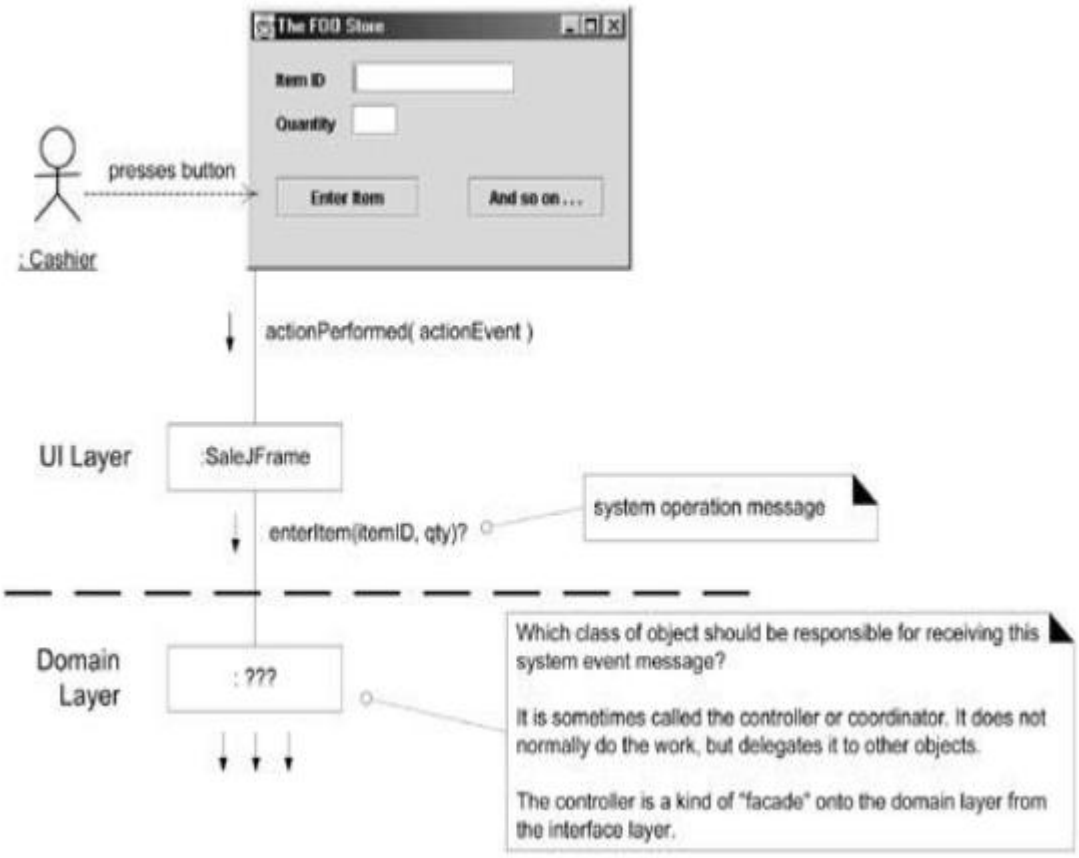
    - Use the same controller class for all system events in the same use case scenario.
    - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions)

Example: **NextGen application contains several system operations.**



Some system operations of NextGen POS Application.

During the design the responsibility of system operations Is done by the controller.

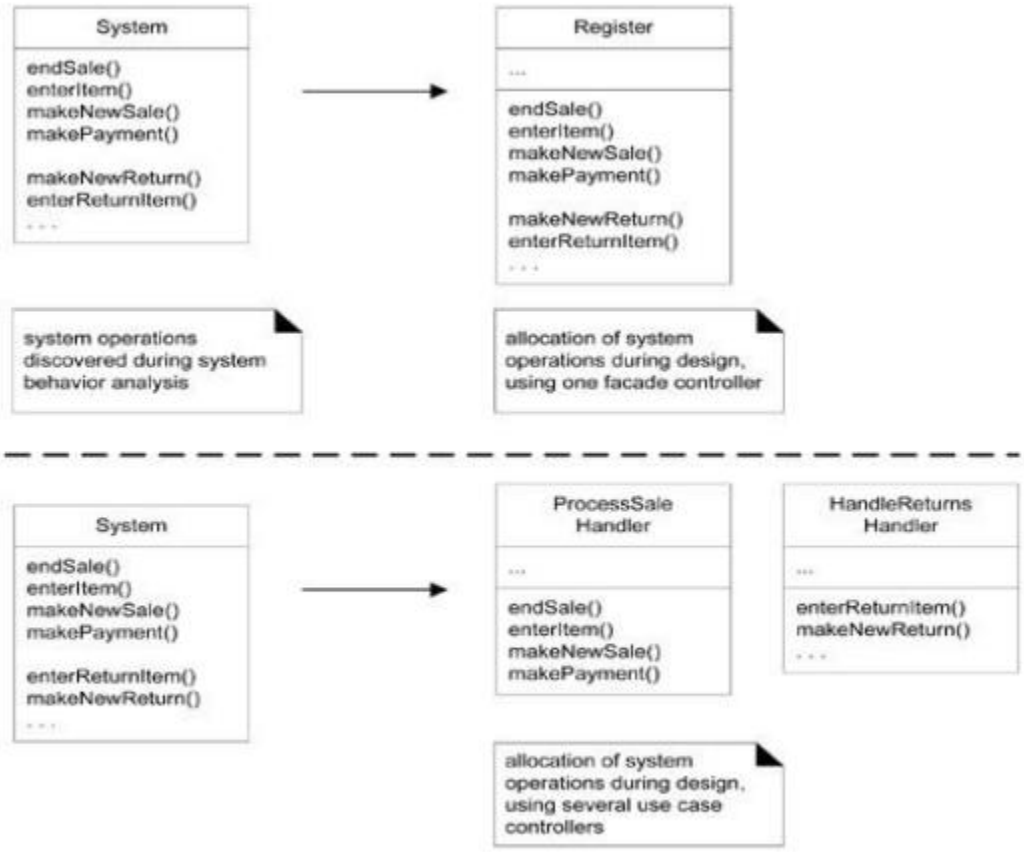


The controller pattern some choices are,

Represents the overall "system," "root object," device, or subsystem.	Register, POSSystem	enterItem(id, quantity) :Register
Represents a receiver or handler of all system events of a use case scenario.	ProcessSaleHandler, ProcessSaleSession	enterItem(id, quantity) :ProcessSaleHandler

A controller should assign other objects the work that needs to be done. It coordinates or controls the activity. Same controller class can be used for all system events to maintain information about the state of use case. A common defect in the design of controllers is it suffers from bad cohesion.

The system operations identified during system behavior analysis are assigned to one or more controller classes like,



Allocation of system operations

Controller

The Facade controller representing the overall system, device, or a subsystem. facade controller representing the overall system, device, or a subsystem.

Façade controllers

1) Facade controllers are suitable when there are not "too many" system events,
2) When the user interface (UI) cannot redirect system event messages to alternating controllers, such as in a message-processing system.

Use case controller

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.

Guideline

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.

Benefits

1. Increased potential for reuse and pluggable interfaces .
2. Opportunity to reason about the state of the use case.

Implementation

The code has

- Process JFrame window referring to domain controller object – Register.
- Define handler for button click.
- Show key message – sending enterItem message to the controller.

Code

```
public class ProcessSaleJFrame extends JFrame
{
   // the window has a reference to the 'controller' domain object
(1) private Register register;

   // the window is passed the register, on creation
public ProcessSaleJFrame(Register _register)
{
   register = _register;
}
   // this button is clicked to perform the
   // system operation "enterItem"
private JButton BTN_ENTER_ITEM;
   ----
   ----
   ----
(2) BTN_ENTER_ITEM.addActionListener(new ActionListener()
    {
    public void actionPerformed(ActionEvent e)
    {
           //  utility class

          ---
          ---
          ---
    (3)   register.enterItem(id, qty);
    }
    } ); // end of the addActionListener call
 return BTN_ENTER_ITEM;
 } // end of method
// …
} // end of class
```

Bloated Controllers

Issues and Solutions

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

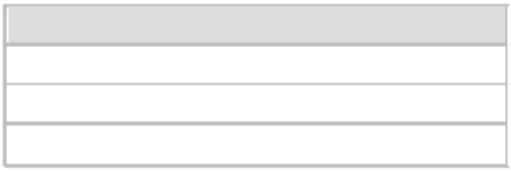
Signs of bloating are:

- There is only a single controller class receiving all system events in the system, and there are many of them. This sometimes happens if a facade controller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.

- A controller has many attributes, and it maintains significant information about the system or domain, which should have been distributed to other objects, or it duplicates information found elsewhere.

Among the **Cures for a bloated controller** are these two:

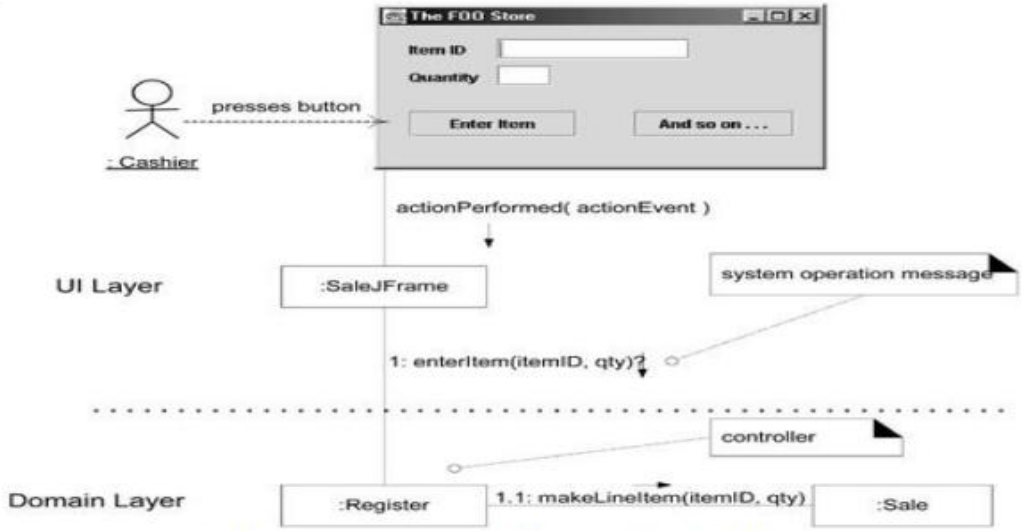
1. Add more controllers a system does not have to need only one. For example, consider an application with many system events, such as an airline reservation system.



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

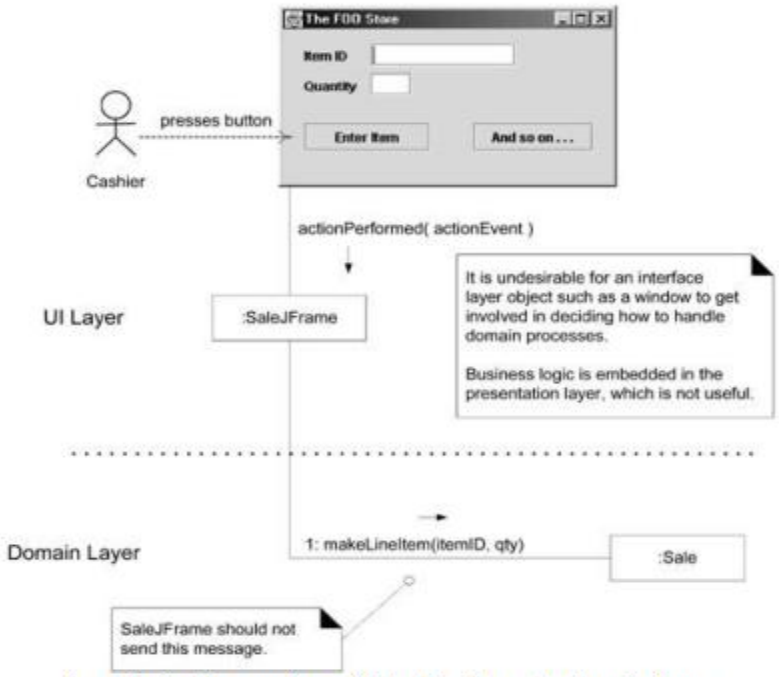
UI Layer Does Not Handle System Events

An important corollary of the Controller pattern is that UI objects (for example, window objects) and the UI layer should not have responsibility for handling system events. Assume the NextGen application has a window that displays sale information and captures cashier operations.



Desirable coupling of UI layer to domain layer

Assigning the responsibility for system operations to objects in the application or domain layer by using the Controller pattern rather than the UI layer can increase reuse potential. If a UI layer object (like the SaleJFrame) handles a system operation that represents part of a business process, then business process logic would be contained in an interface (for example, window-like) object; the opportunity for reuse of the business logic then diminishes because of its coupling to a particular interface and application.



Less desirable coupling of interface layer to domain layer

4.3.5 HIGH COHESION

Problem

How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

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 that does not do a tremendous amount of work has high cohesion. These elements include classes, subsystems, and so on.

Solution

Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

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 affected by change

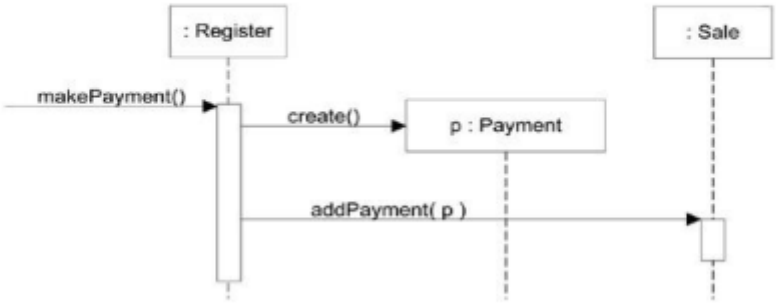
Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.

Example

Create a payment instance and associate it with sale

DESIGN 1

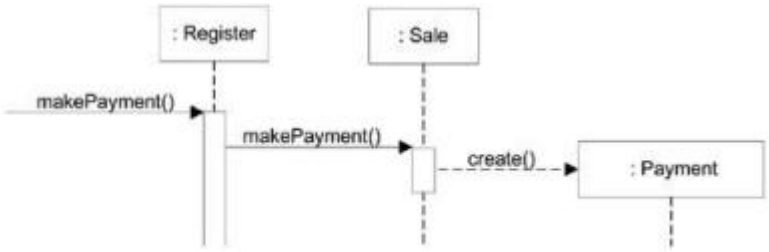
- Register records a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment.
- The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.



Register creates Payment

DESIGN 2

The second design delegates the payment creation responsibility to the Sale supports higher cohesion in the Register.



Sale creates Payment

- In the second design, payment creation in the responsibility of sale.
    a. It is highly desirable because it supports High Cohesion & Low Coupling

Scenarios of varying degrees of functional cohesion

1. **Very low cohesion**: A class is solely responsible for many things in very different functional areas.

   Ex : Assume the existence of a class 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.

2. **Low cohesion:** A class has sole responsibility for a complex task in one functional area.

   Ex: Assume the existence of a class 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.

3. **High cohesion**: A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

   Ex: Assume the existence of a class called RDBInterface that 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.

   Ex: Assume the existence of a class called Company that 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.

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.

- Easy to maintain
- Understand and
- Reuse

Modular Design

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Modular design creates methods and classes with single purpose, clarity and high cohesion.

Lower cohesion is had in

- Grouping responsibilities or code into one class or component.
- Distributed server objects.

Benefits

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