

UNIT - 4

DESIGN PATTERNS

4.1 GRASP DESIGNING OBJECTS WITH RESPONSIBILITIES

4.1.1 Introduction

After the requirements identification, add the methods to the classes and define the messages between the objects.

The designing of object starts with

- Inputs
 - Activities and
 - Outputs
- (i) GRASP → General Responsibility Assignment Software Patterns
- (ii) GOF → Gang of Four
- (iii) RDD → Responsibility Driven Design

GRASP

The GRASP is a tool that helps to master the basics of OOD and understanding responsibilities.

GRASP is a learning aid that helps to understand the essential object design and apply the design in a methodical, rotational and explainable way.

RDD

RDD is a general metaphor for thinking about object oriented design.

The responsibilities include

- Doing responsibility → creating action, initializing, controlling and coordinating activities
- Knowing responsibilities → includes knowing about private and related data.



4.1.2 Applying Grasp To Object Design

GRASP stands for -General Responsibility Assignment Software Patternsl.

It means low grasping of these principles to successfully design object oriented software.

While coding or drawing interaction and class diagrams, developers apply the ideas behind GRASP to master the principles of object oriented design.

There are nine different GRASP patterns:

- (i) Creator
- (ii) Information expert
- (iii) Low coupling
- (iv) Controller
- (v) High cohesion
- (vi) Polymorphism
- (vii) Pure fabrication
- (viii) Indirection
- (ix) Protected variations

Out of the nine the first five are dealt in detail.

4.2 CREATOR

Problem

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

- Low coupling
- Increased clarity
- Encapsulation
- Reusability

Solution

If B is a creator of A objects then atleast one of the following must be true.

- B contains A
- B 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).

If more than one option is true.

Class B aggregates or contains class A.

Example

NEXTGen POS Application

We have to find who is the creator of SalesLineItem instance.

Consider the partial domain model of SaleLineItem.

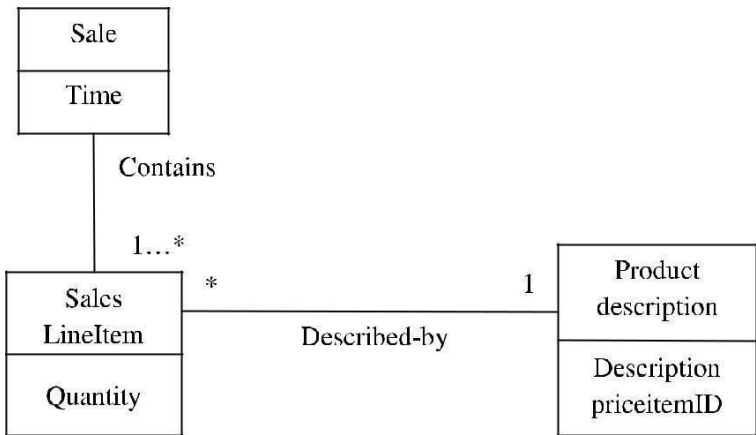


Figure 4.1 Partial domain model

Here `_Sale` takes the responsibility of creating `_SalesLineItem` instance. Since sale contains many `_SalesLineItem` objects.

While assigning responsibilities `_makeLineitem` must also be defined in `_Sale`.

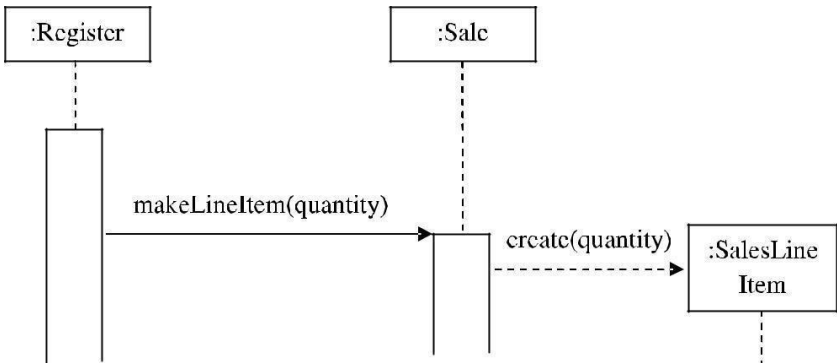


Figure 4.2 Creating a SalesLineItem

The creator pattern finds a creator that needs to be connected to the created event.

The common task of creator is to assign responsibilities related to the creation of objects. All common relationships between classes are

- Composite aggregates part



- Container contains content
- Recorder records

Enclosing container or recorder is good for creating the thing contained or recorded. Composition is also considered for creator.

Initialization during creation is done by some method like Java constructors.

Example: `_Payment` instance while creation initialized with `_Sale` total.

`_Sale` is a candidate creator of `_Payment`.

Contradictions:

Based on some external value, creation requires complexity like

- Recycled instances for performances.
- Creating an instance from one of a family of similar classes based an external property etc.
- Abstract factory or
- Concrete façade

Then use the class of creator.

Benefits

- Lower maintenance due to low coupling
- Higher opportunities for reuse

4.3 INFORMATION EXPERT (OR EXPERT)

Problem

When designing objects, interaction between objects are defined about assigning responsibilities to software classes.

This makes the software easier to

- Maintain
- Understand and
- Extend

Solution

Assign the responsibilities to information expert.

Example:

NextGEN POS Application

Some classes need to know the grand total of `_sale`.

Here we start assigning responsibilities by clearly starting the responsibility.



Information expert looks for the class of objects that has information to determine total.

- 1) Look at relevant classes in design model
- 2) Otherwise look in domain model.

For example:

We look at design model. It is minimal and hence go for domain model for information expert `_sale`.

Software class `_sale` is added with the responsibility of getting total with the method `_getTotal`.

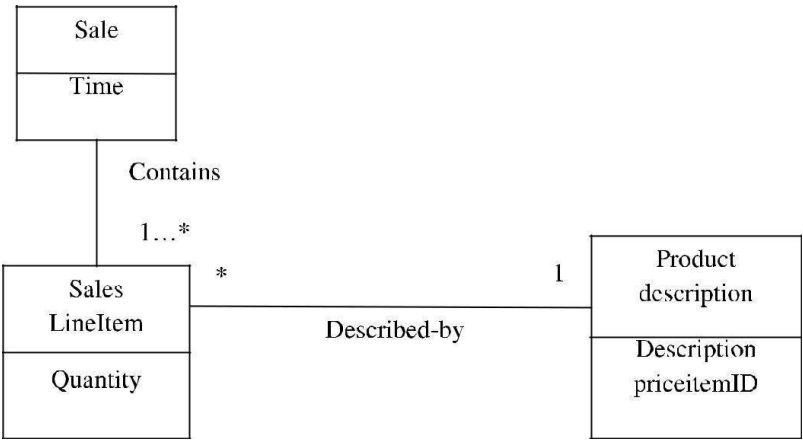


Figure 4.3 Partial domain model for association of sale

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

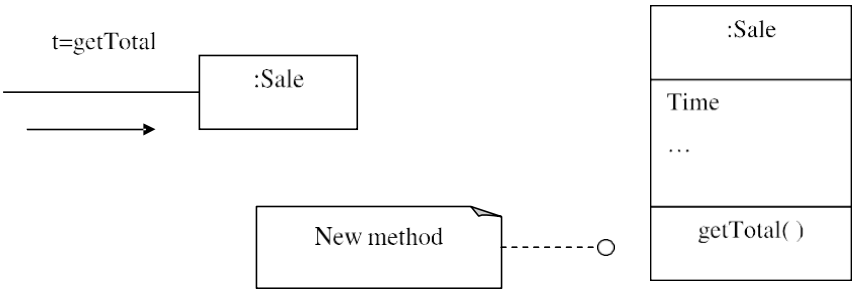


Figure 4.4 Partial interaction and class diagrams

To determine lineitem sub total we need,

SalesLineitem.quantity

ProductDescription.price

By information expert using the above the SalesLineItem should determine subtotal.

This is done by

- Sale sending `getSubtotal` messages to each SalesLineItem and sum the results.

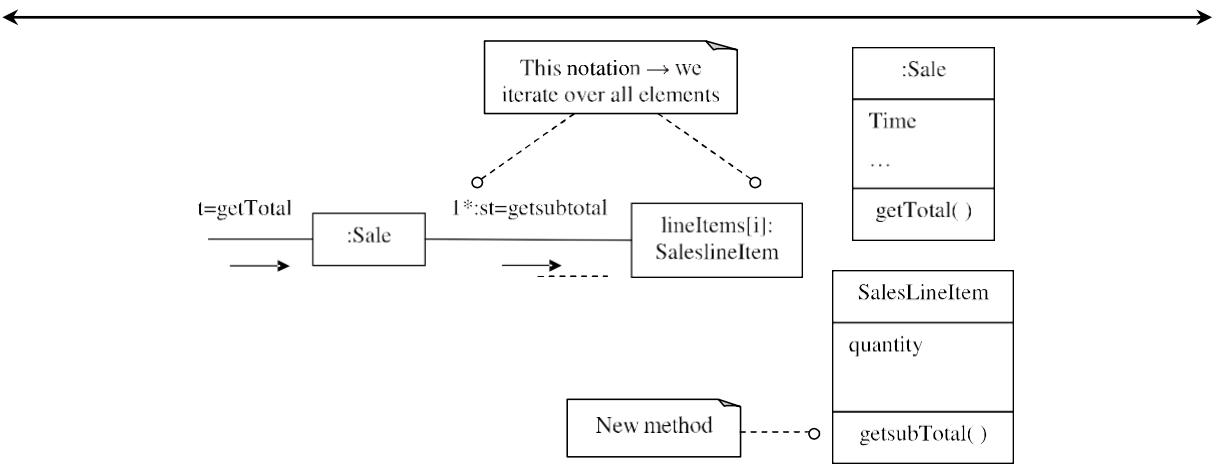


Figure 4.5 calculating Sales total

After knowing and answering subtotal, a SalesLineItem has to know product price.

ProductDescription is an information expert for answering price.

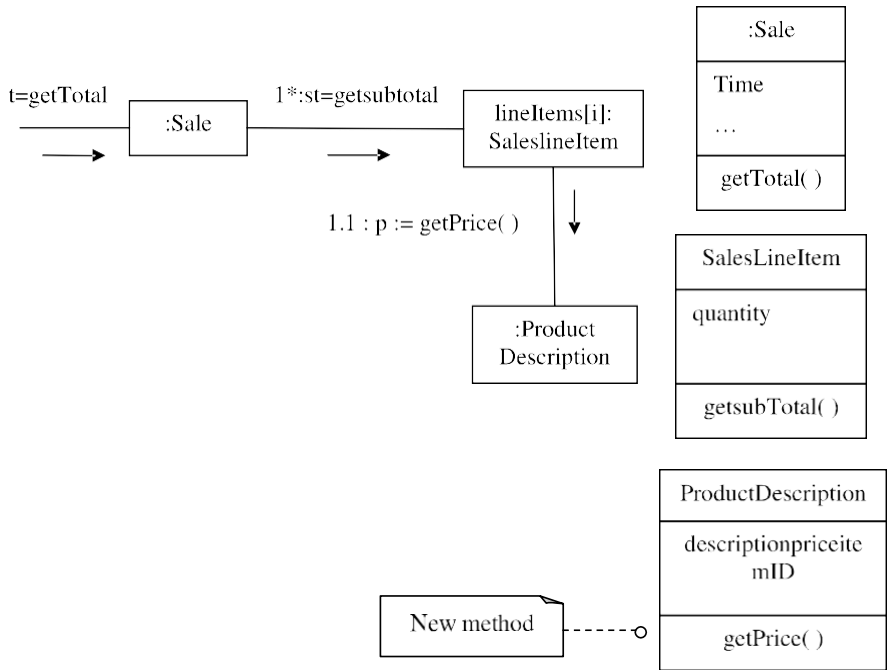


Figure 4.6 Calculating the sale total

Finally, three design classes, assigned with three responsibilities to find sales total.

Design class	Responsibility
Sale	Knows sale total
SalesLineItem	Knows line item subtotal
Productdescription	Knows product price



- Thus information expert is used in the assignment of responsibilities.
- Experts express the common ‘intuition’ that objects do things related to information they have.
- ‘Partial’ information experts will collaborate in the task.

Example: Sales total problem – collaboration of three classes of objects.

Profit and loss statement – collaboration of chief financial officer, accountants to generate reports on credits and debits.

- Information expert thus has real world analogy
- Information experts are basic guiding principle used continuously in object design.

Contradictions

Solution of expert is not desirable in some cases due to problems of 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]

Separation of major concerns improves coupling and cohesion in a design.

Benefits

- 1) Information encapsulation use their information to fulfill tasks.
- 2) High cohesion is supported

4.4 LOW COUPLING

Coupling is the measure of how strongly one element is connected to the other elements.

Types of coupling

There are 2 types of coupling

- 1) Low coupling or weak coupling
- 2) High coupling or strong coupling

Low coupling

An element if does not depend on too many other elements like classes, subsystems systems it is having low coupling.

High coupling

A class with high coupling relies on many other classes.

The problems of high coupling are

- Forced local changes
- Harder to understand
- Harder to reuse

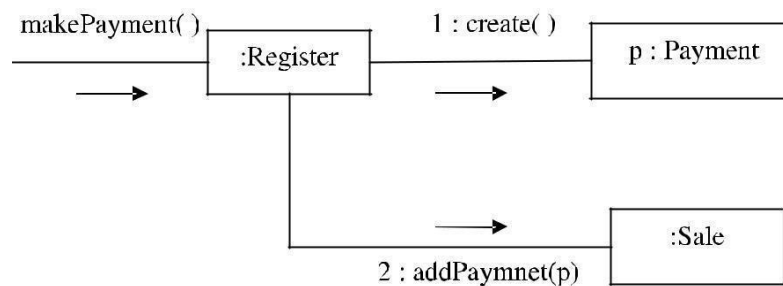
Example**Figure 4.7 NextGen Case study**

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

Design 1:

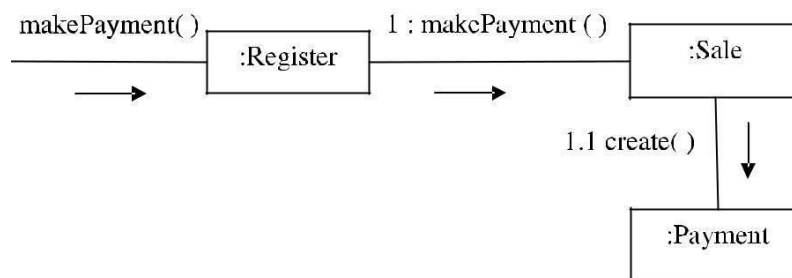
Suggested by creator

- 1) Register instance send addpayment message to sale, passing newPayment as a parameter.
- 2) Register class couples with payment class and creates payment.

**Figure 4.8 register creates Payment****Design 2:**

Suggested by low coupling

- 1) Here sale does the creation of payment.
- 2) It does not increase coupling and hence preferred.

**Figure 4.9 Sales creates Payment**

Low coupling is an evaluation principle for evaluating all designs.

Common forms of coupling from objects A to B include

- 1) A has an attribute referring to B or instance of B



- 2) A calls B's services
- 3) A has a method referring to B or B's instance
- 4) A is direct/indirect subclass of B
- 5) B is an interface and A implements B.

Low coupling

- 1) Assigns responsibility that will not yield negative results that cause high coupling.
- 2) Support design of independent classes.

Classes that are

- 1) Generic in nature
- 2) Having high probability of reuse will have low coupling. Extreme case of low coupling is no coupling between classes.

That is not desired because a moderate degree of coupling between the classes is normal and is necessary for creating an object oriented system.

High coupling to stable elements is seldom a problem.

Benefits

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

4.5 HIGH COHESION

Cohesion is a measure of how strongly related and focused responsibilities of an element are. An element has high cohesion if it,

- Does not do tremendous work
- Has high responsibilities

Difficulties of low cohesion are

- Hard to comprehend
- Hard to reuse
- Hard to maintain
- Delicate

Example

Create a payment instance and associate it with sale

Design 1

- Register records a payment in real world

- Then it sends addpayment message to sale, passing newpayment as a parameter.

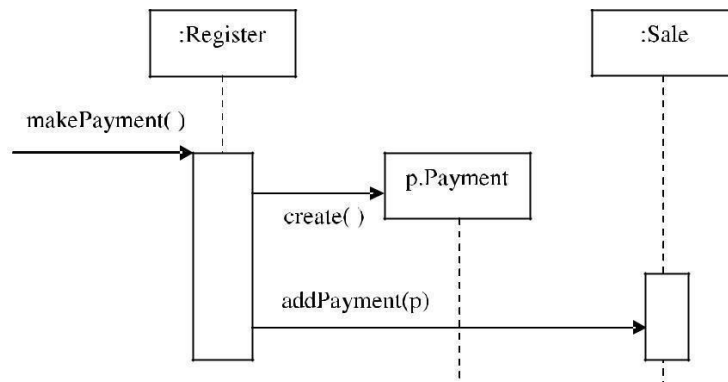


Figure 4.10 Register creates payment

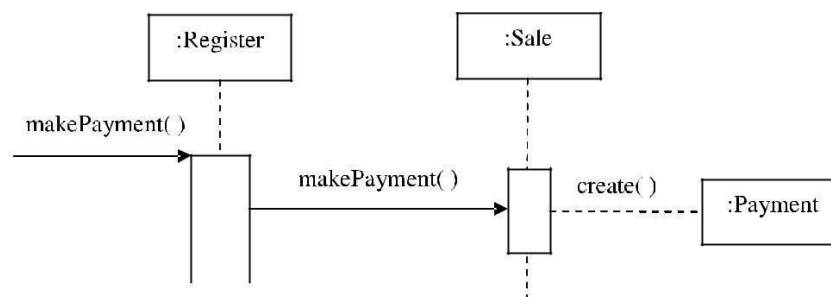


Figure 4.11 Sale creates payment

- In the second design, payment creation is the responsibility of sale.
- It is highly desirable because it supports
 - High cohesion and
 - Low coupling

Scenarios of varying degrees of functional cohesion

(1) *Very low cohesion*

A class is solely responsible for many things in different functional areas.

(2) *Low cohesion*

A class has sole responsibility for a complex task in one functional area.

(3) *High cohesion*

A class has moderate responsibilities in one functional area and collaborates with others.

(4) *Moderate cohesion*

A class has light weight. It is responsible for different areas logically related to the class concept but not to each other.



Rule of thumb

A class with high cohesion has small number of methods (with highly related functionality) and does not work too much.

High cohesion is

- Easy to maintain
- Understand and
- Reuse

Modular design

—Modularity is the property of 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 of high cohesion are

- 1) Clarity
- 2) Ease of comprehension
- 3) Reuse of fine-grained, highly related functionality.

4.6 CONTROLLER

A controller is defined as the first object beyond the user interface (UI) layer that is responsible for the receiving or handling a system operation message.


Example:

- 1) A cashier is POS terminal pressing -EndSale button indicating -sale has ended
- 2) Writer using word processor presses -spell check button to perform checking of spelling

Solution

Assign responsibility to one of the following

- Represent overall system, a root object
 - These are variations of façade controller.
- Represents a usecase scenario called
 - <usecaseName> handler

- 
- <usecaseName> coordinator or
 - <usecaseName> session

Here we use some controller class for all system events in same usecase scenario.
A session is an instance of a conversation with an actor.

Example:

NextGen application contains several system operations.

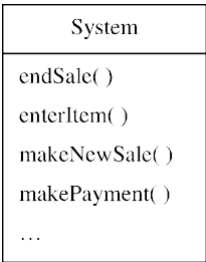


Figure 4.12 Some system operations of NextGen POS Application

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

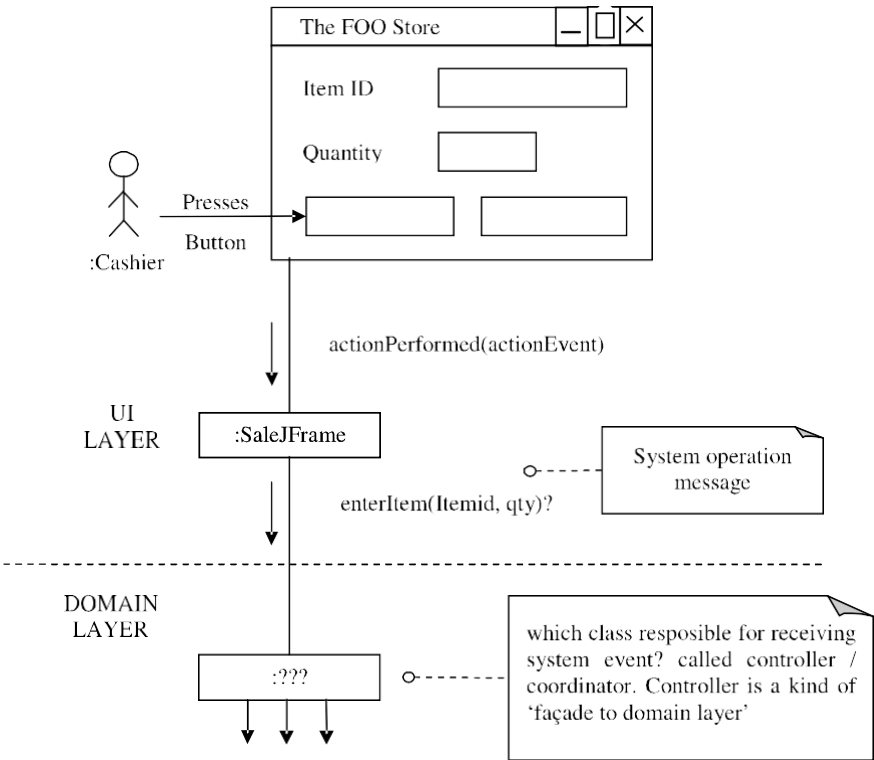


Figure 4.13 Controller for enterItem

The controller are

- 1) Register POS system – Represents overall `_system`, `_root` object

2) Process Sale handler – Represents receiver/handler of system events.
The choice of which is the most appropriate controller is influenced by other factors.

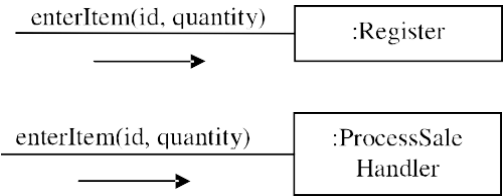


Figure 4.14 controller choices

The system operations are assigned to the controller classes like,

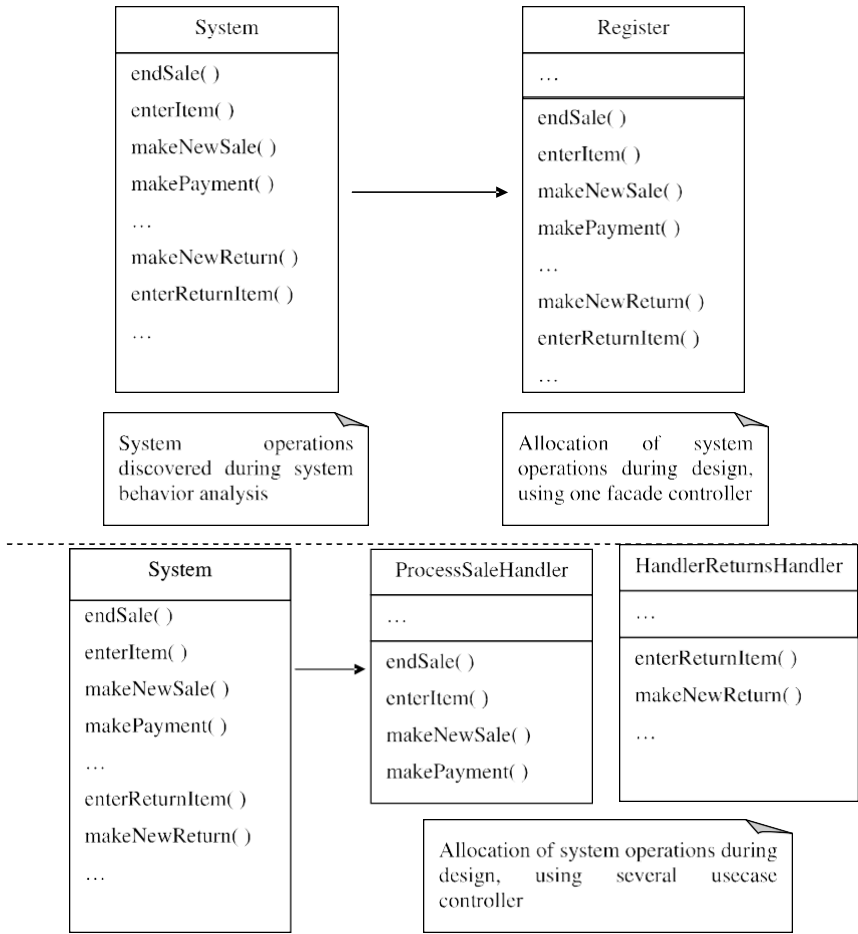


Figure 4.15 Allocation of system operations

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 usecase.
A common defect in the design of controllers is it suffers from bad cohesion.

←—————→ Controllers

The facade controller represents the overall system, device or a sub system.

Choose some class name that suggests a cover or faced over other layers of the application that provides main service calls from UI layer down to the other layers.

Facade controllers are used

- 1) Where there are not —too many system events
- 2) When the UI (User Interface) cannot redirect system event messages.

In used case controller there is a different controller for each use case. Facade controllers lead to low cohesion or high coupling design.

So usecase controllers are good when there are many system events across different processes.

- Boundary objects
- Entity objects
- Control objects

Boundary objects : Abstractions of the interfaces

Entity objects : Application independent domain software objects

Control objects : Use case handles

An important corollary of the controller pattern is UI objects.

System operations should be handled in application logic or domain layer of objects rather than UI layer of a system.

Web UIs and server side application of controller

An approach used is

ASP.NET and webforms:

- Developers insert application logic handling in the -code behind file, mixing application logic into the UI layer.
- Server side web UI frameworks embody the concept of web MVC (Model – view - controller) pattern.
- Choosing server technical frameworks strongly influence handling of serverside system operation.
- To lower the coupling of UI, the clientside UI forwards the request to the local client side controller.

Benefits

- 1) Increased potential for reuse and pluggable interfaces.

```

package.com.nextgen.ui.swing;
//import
public class ProcessSaleJFrame extends JFrame
{
    //window refers to _controller‘ domain object
    (1) private Register register
    public ProcessSaleJFrame (Register-register)
    {
        register = -register;
    }
    private JButton BTN_ENTER_ITEM /* this button is clicked to perform sys operation
    -enetrItem */
    _____
    _____
    _____
    (2) BTN_ENTER_ITEM.addActionListener (new ActionListener ( )
    {
        public void actionPerformed (ActionEvent e)
        {
            // utility class
            _____
            _____
            (3) register.enterItem(id, qty);
        }
    }); // end of ActionListener
    return BTN_ENTER_ITEM;
} // end of method
_____
}

```



Implementation with java statements: Client Browser and WebUI

- 1) Obtain a reference to the Register domain object.
- 2) Send -enterItem message to domain controller object.

Code:

```
package.com.nextgen.ui.web;
//import
public class EnterItemAction extends Action
{
    public ActionForward execute (ActionMapping mapping,
    ActionForm form, HttpServletRequest request, HttpServletResponse response)
    throws exception
    _____
    _____
    (1) Register register = repository.getRegister ( );
    _____
    _____
    (2) register.enterItem (id, qty);
} // end of method
} // end of class
```

A controller class has low cohesion i.e. unfocused and handling many responsibilities. It is called bloated controller.

- When facade controller is chosen, a single controller class receives all system events.
- Violating information expert and high cohesion controller itself performs many of the tasks.
- Controller has many attributes.

These are signs of bloated controller.

Cures for bloated controllers

- 1) Add more controllers

Example: Airline Reservation may have

Usecase controllers
Make reservation Handler
Manage Schedules Handler
Manage Fares Handler

- 2) Controller must delegate the fulfillment of each system operation.
UI layer does not handle system events.

The responsibility of system operations in application can be assigned by controller pattern rather than UI layer.

This is illustrated by diagrams.

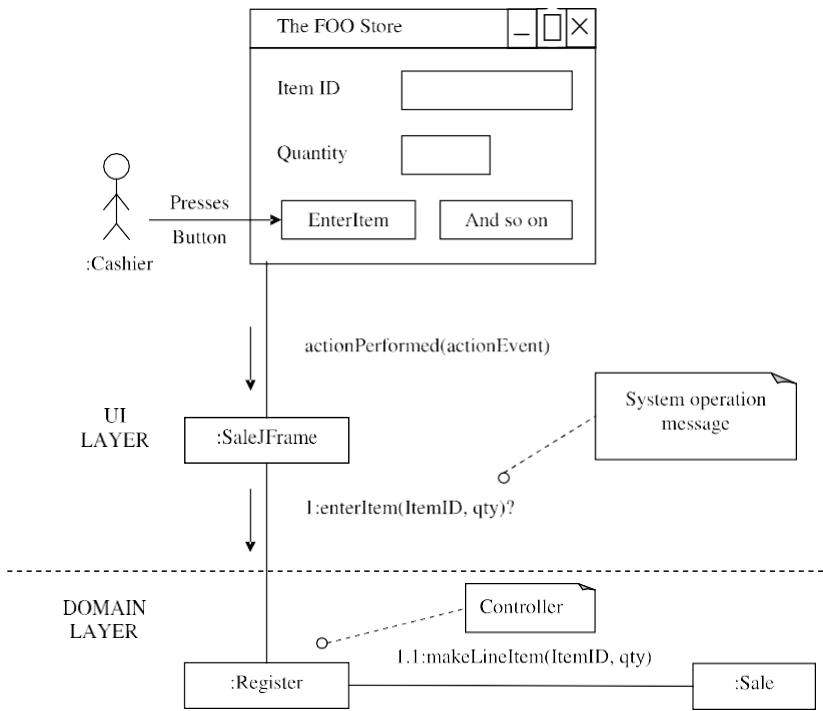


Figure 4.16 Coupling of UI layer to domain layer

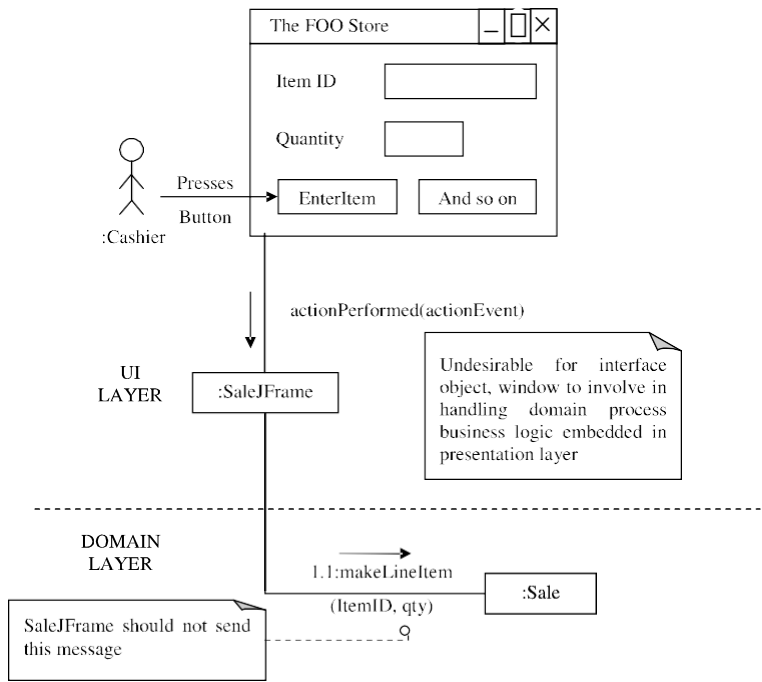


Figure 4.17 Less desirable coupling of interface layer to domain layer

Related patterns

- Command → Each message is a command object
- Facade Facade controller is kind of facade
- Layers → Domain logic in domain layer than presentation layer
- Pure fabrication → A use case controller

4.7 DESIGN PATTERNS

Designing for Visibility

The ability of one object to have reference of another is called visibility.

Example:

The getProductDescription message sent from a Register to a product catalog means that ProductCatalog instance is visible to register.

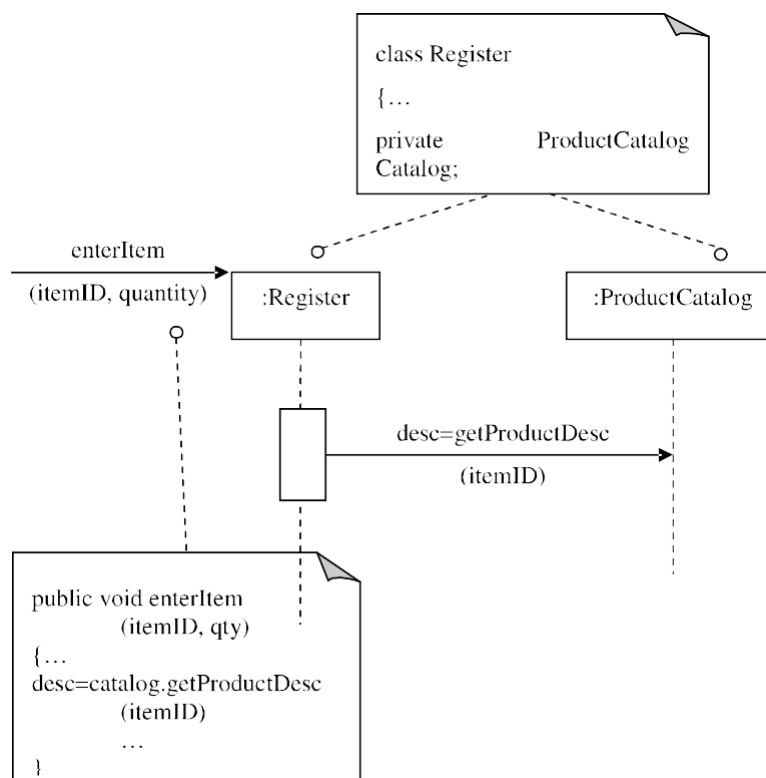


Figure 4.18 Visibility from register to Product catalog

Ways of visibility: from Object A to Object B

- 1) Attribute visibility – B is an attribute of A
- 2) Parameter visibility – B is a parameter of a method of A

- 3) Local visibility – B is a local object in a method of A
- 4) Global visibility – B is globally visible in some way

Motivation of visibility

For an object A to send a message to an object B, B must be visible to A.

Attribute visibility

- It is a permanent visibility.
- Attribute visibility from A to B exists if B is attribute of A.
- It persists as long as A and B exists

Example

Register has attribute visibility to ProductCatalog, because it is an attribute to Register.

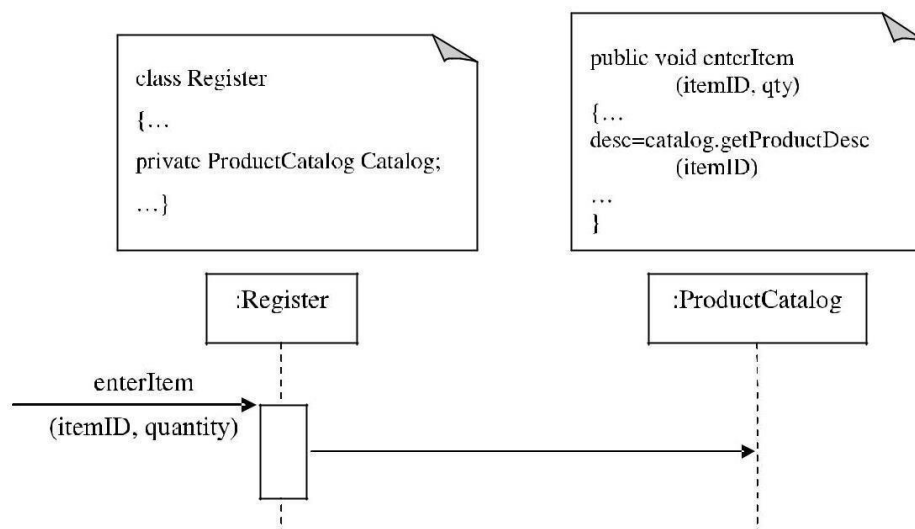


Figure 4.19 Attribute visibility

Parameter visibility

Here B passes as parameter to A when parameter visibility exists.

It persists only within the scope of the method.

So it is relatively temporary visibility.

Example:

When makeLineItem is sent to a sale instance, a ProductDescription instance is passed as a parameter.

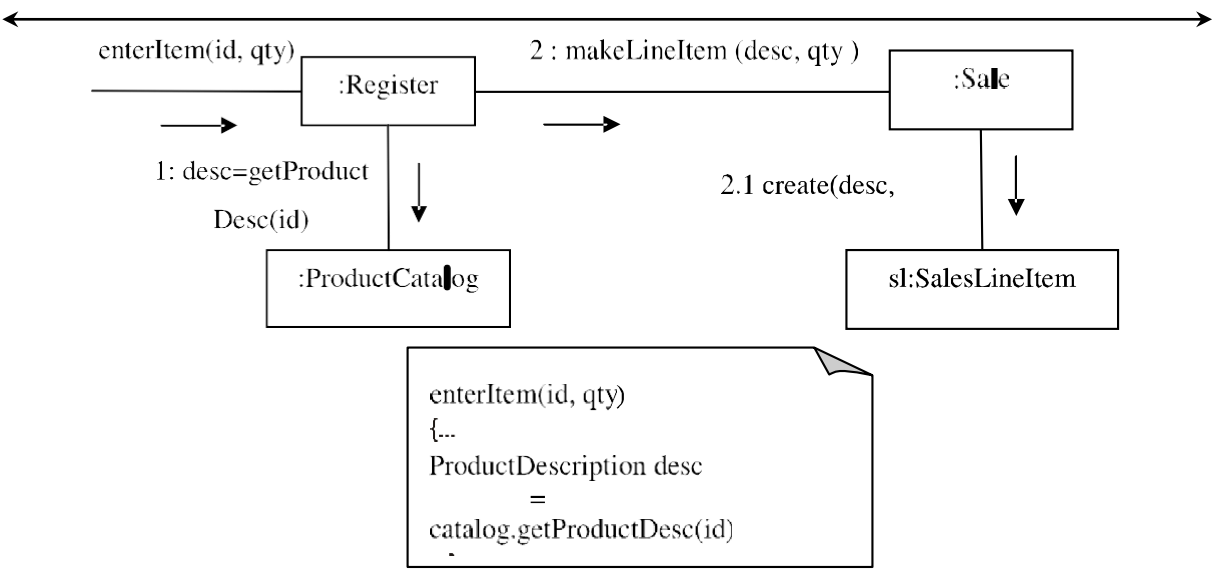


Figure 4.20 Parameter visibility

Local visibility

- It exists from A to B when B is declared as a local object within a method of A.
- It persists only within the scope of the method. So it is relatively temporarily visible.
- The local visibility is achieved by
- Creating a new local instance and assigning to a local variable
 - Assign the returning object from a method invocation to a local variable.

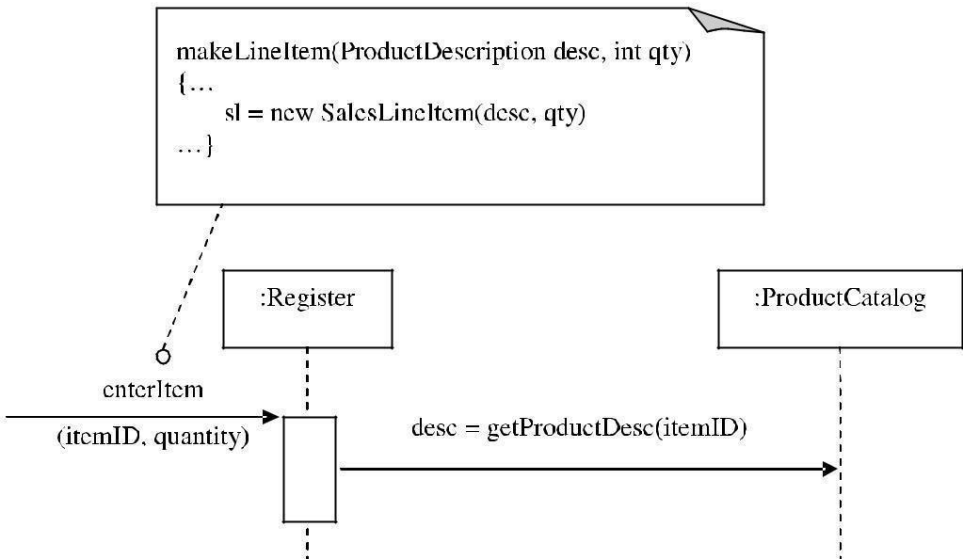
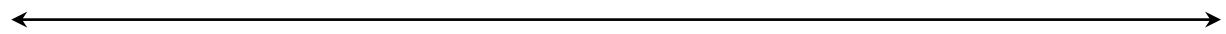


Figure 4.21 Local visibility



Global visibility

Here global visibility is achieved from A to B if B is global to A.

It persists as long as A and B exist. So it is permanent visibility.

To achieve global visibility,

- (i) Assign instance to global variable in languages like C++.
- (ii) To use `_Singleton` pattern.

4.8 FACTORY METHOD

It is called as

- Simple factory or
- Concrete factory

It is the simplification of GOF abstract factory.

Next adapter raises the problem,

- Who create adapters?
- How to create adapters?

When domain objects create the adapter, their responsibilities are beyond pure application logic and related to connectivity with other software components.

So, when a domain object creates adapters

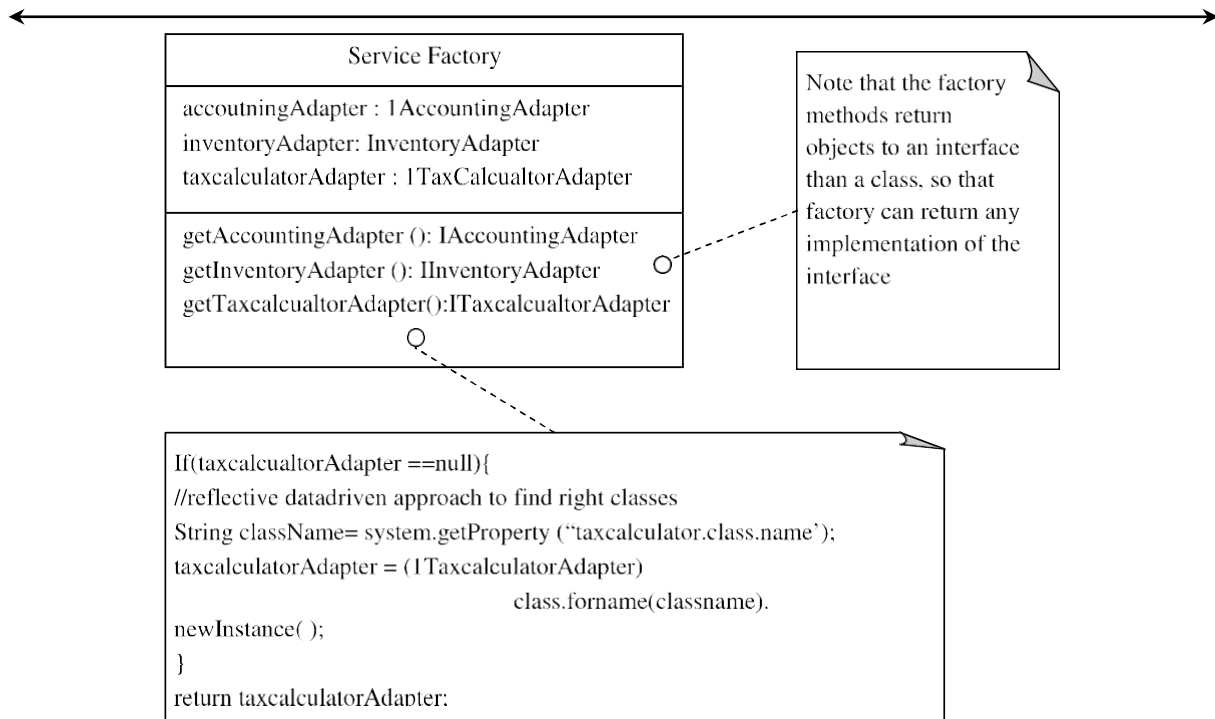
- (i) It does not support goal of separation of concerns.
- (ii) It lowers cohesion.

So we go in for `_factory` pattern, where pure fabrication -factory object is defined to create objects.

Advantages of factory

- (i) Separate the responsibility of complex creation into cohesive helper objects.
- (ii) Potential complex creation is hidden.
- (iii) Performance-enhancing memory management strategies such as object catching or recycling are allowed.

Strategies such as object catching or recycling is allowed.



Name : Factory

Problem : For complex creation logic, for better cohesion who is responsible for creating objects.

Solution : A pure fabrication object called `__factory` is created that handles creation. In `__Service Factory`, the logic to decide the class creation is resolved by reading in the class name from external source.

This is called -datadriven design||

4.9 ADAPTER

Name : Adapter

Problem : If similar components have different interfaces then how to resolve incompatible interfaces.

Solution : Using an intermediate adapter object, convert original interface of a component to another interface.

The NextGen POS system supports many third party services like,

- Tax calculators
- Credit authorization services
- Inventory systems
- Accounting systems etc

Add a level of indirection with objects to adapt to the solution.

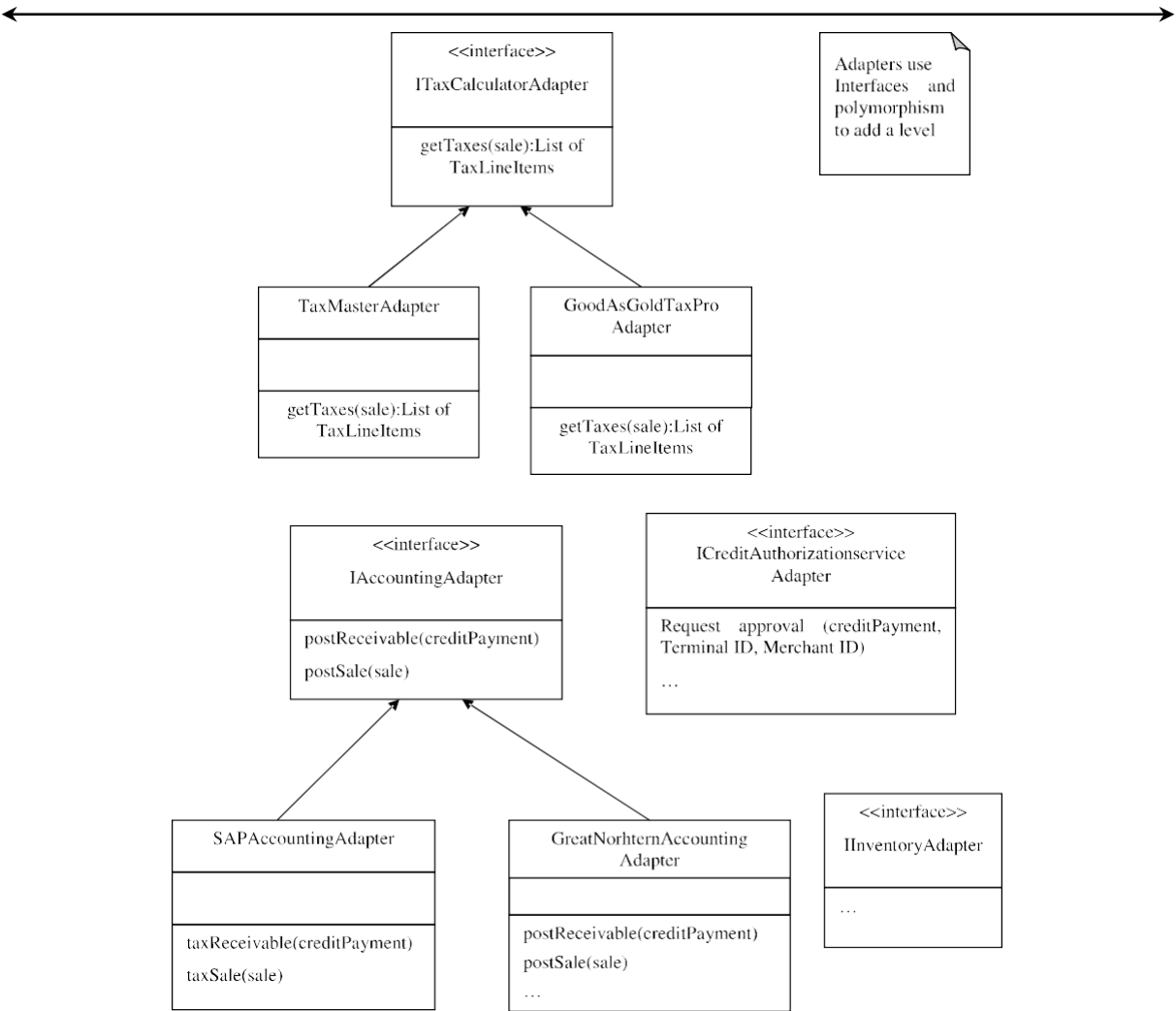


Figure 4.22 Adapter pattern

Here a particular adapter will be instantiated such as

- SAP for accounting
- SOAP XML interface for intranet web services.

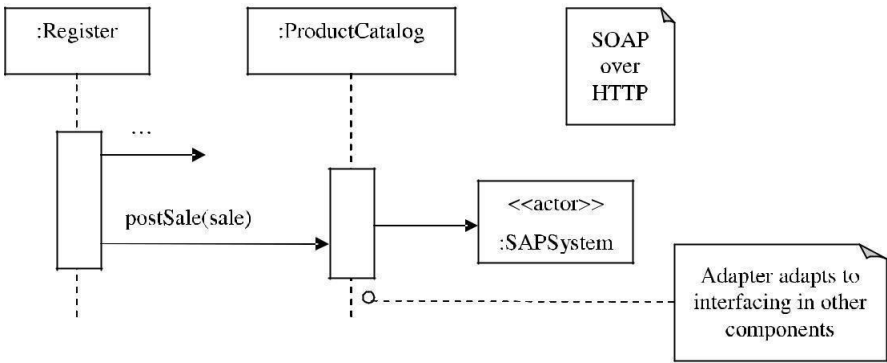


Figure 4.23 Using an Adapter

← Type names are given by pattern name, —Adapterl, to communicate the user of the design pattern being used.

A resource adapter that hides external system is considered as façade object.

The motivation to call resource adapter exist when wrapping objects provide adaptation.

Adapter and GRASP

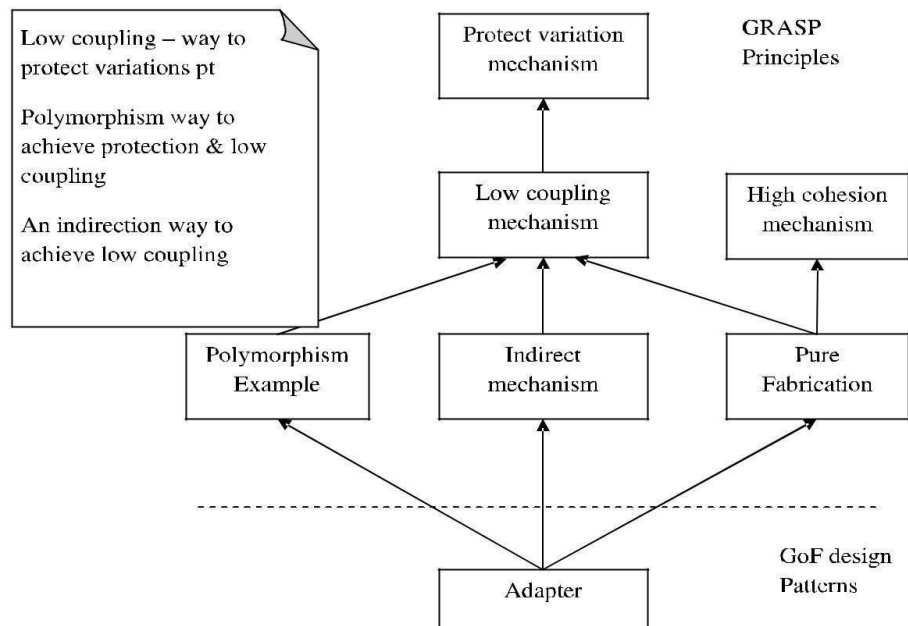


Figure 4.24 Relating Adapter to GRASP principles

Adapters can be related to some core GRASP principles.

But here the underlying themes are highly important and details of adapter are secondary.

4.10 SINGLETON (GOF)

With the services factory, we have a problem who creates the factory.

One instance of factory is needed with the process.

Since at different places the adapter are called, the quick reflection suggest that the methods of this factory may need to be called form various places in code.

This problem can be solved by passing ‘_Service factory’ as a parameter, initialize the objects that need visibility.

But it is not convenient and hence go for ‘_Singleton’ pattern.

Name : Singleton

Problem : Exactly one instance of class allowed called -Singletonl. Objects are global and have single point of access.

Solution : Static method of class is defined, that returns singleton.



Implementation of singleton applying UML

Singleton is implemented with a `_1` in the top right corner of name compartment.
Class X defines a static method `_getInstance` that provides single instance of X.
Here developer has global visibility to this instance by static `_getInstance` method of the class.

```
class.  
public class Register  
{  
    public void initialize ()  
    {  
        ... do some work...  
        //accessing singleton factory in a getInstance call accounting Adapter =  
        ServicesFactory.getInstance ( ).getAccountingAdapter ( );  
        ... do some work...  
    }  
    //other methods...  
} //end of class
```

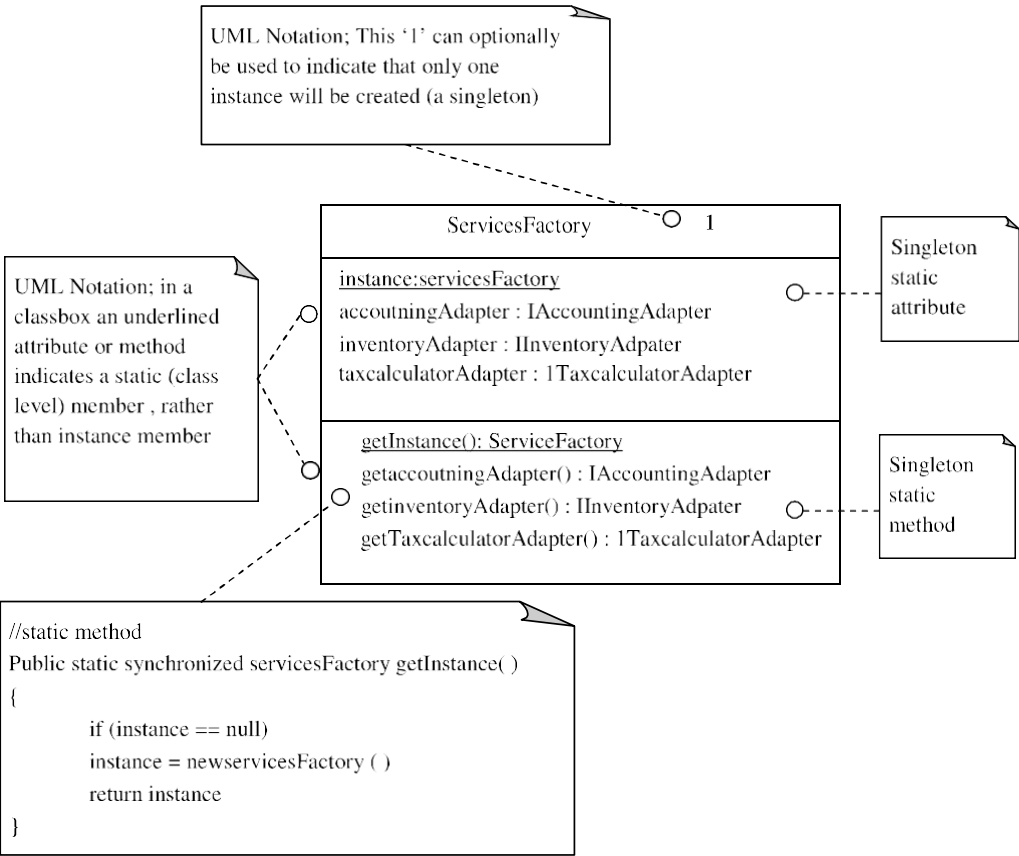
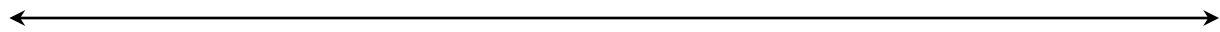


Figure 4.25 Singleton Pattern in the ServicesFactory class



At any point of code, in any method of class, one can write

Singletonclass.getInstance ()

To obtain visibility to singleton instance. Send the message

'Singletonclass.getInstance ()'

Design Issues in implementation

In multithreaded applications, the creation step of -lazy initialization logic is a critical section requiring thread concurrency control.

Example in java

```
public static synchronized ServiceFactory getInstance ( )
{
    if (instance == null)
    {
        //critical sec in Multithreaded application
        instance = new ServicesFactory ( );
    }
    return instance;
}
```

Lazy initializations is preferred for

- 1) Creation work is avoided.
- 2) getInstance lazy initialization sometimes have complex and conditional creation logic.

Eager initialization

```
public class servicesFactory
{
    //eager initialization
    private static servicesFactory instance = new ServicesFactory ( ); public static
    servicesFactory getInstance ( ) {
        Return instance;
    }
    // other methods...
```

```
}
```

```
:Register
```

```
:ServiceFactory
```

```
aa=getAccounting Adapter
```

The `_1` indicates that invisibility to this is achieved by singleton pattern

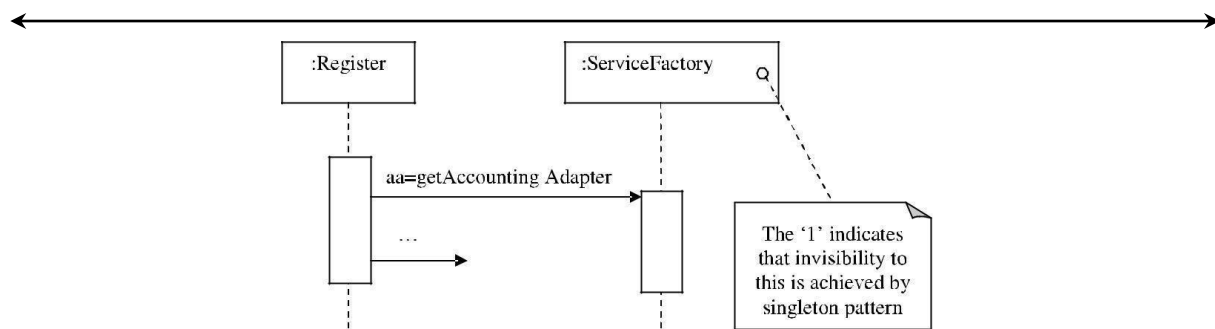


Figure 4.26 getInstance singleton pattern

While implementation, we use instance object with instance side methods.

Example:

Static method getAccountingAdapter to serviceFactory.

These are preferred for reasons like:

- 1) static methods are not polymorphic No overriding in subclasses
But instance side methods permit subclassing and refinement
- 2) singleton instance could be remote enabled.
- 3) In application X, class may be singleton. But in another application __y', it may be multiton.

That is class is not always singleton

So here instance-side solution offers flexibility.

Singleton pattern is used for factory objects and façade objects.

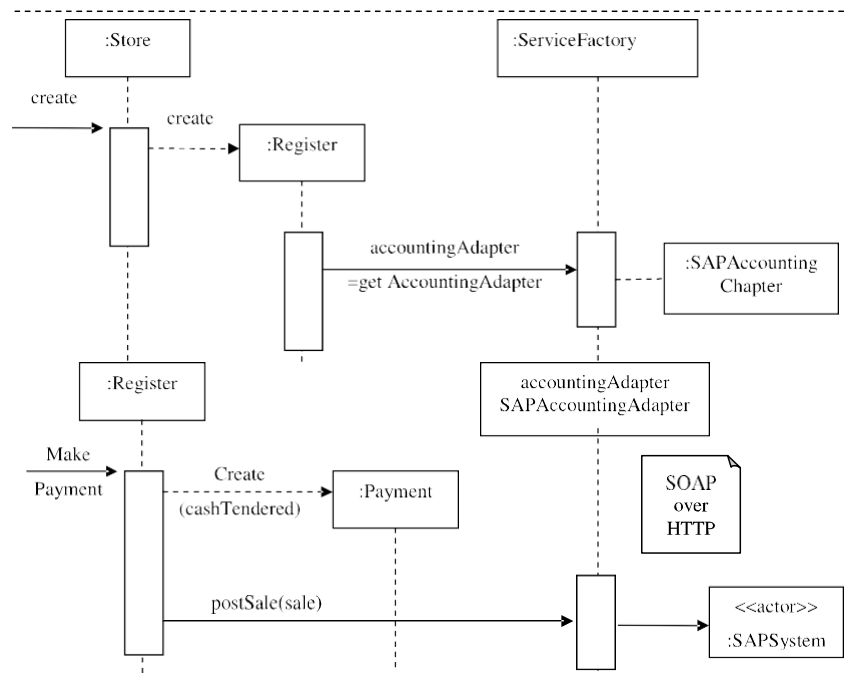


Figure 4.27 Adapter, Factory and Singleton patterns applied to design

4.11 OBSERVER PATTERNS

To extend the solution found for changing data, add the ability for a GUI window to refresh its sale.

The model-view separation principle discourages such solutions. It states that -Model objects should not know about view or presentation objects such as window.

It promotes Low coupling.

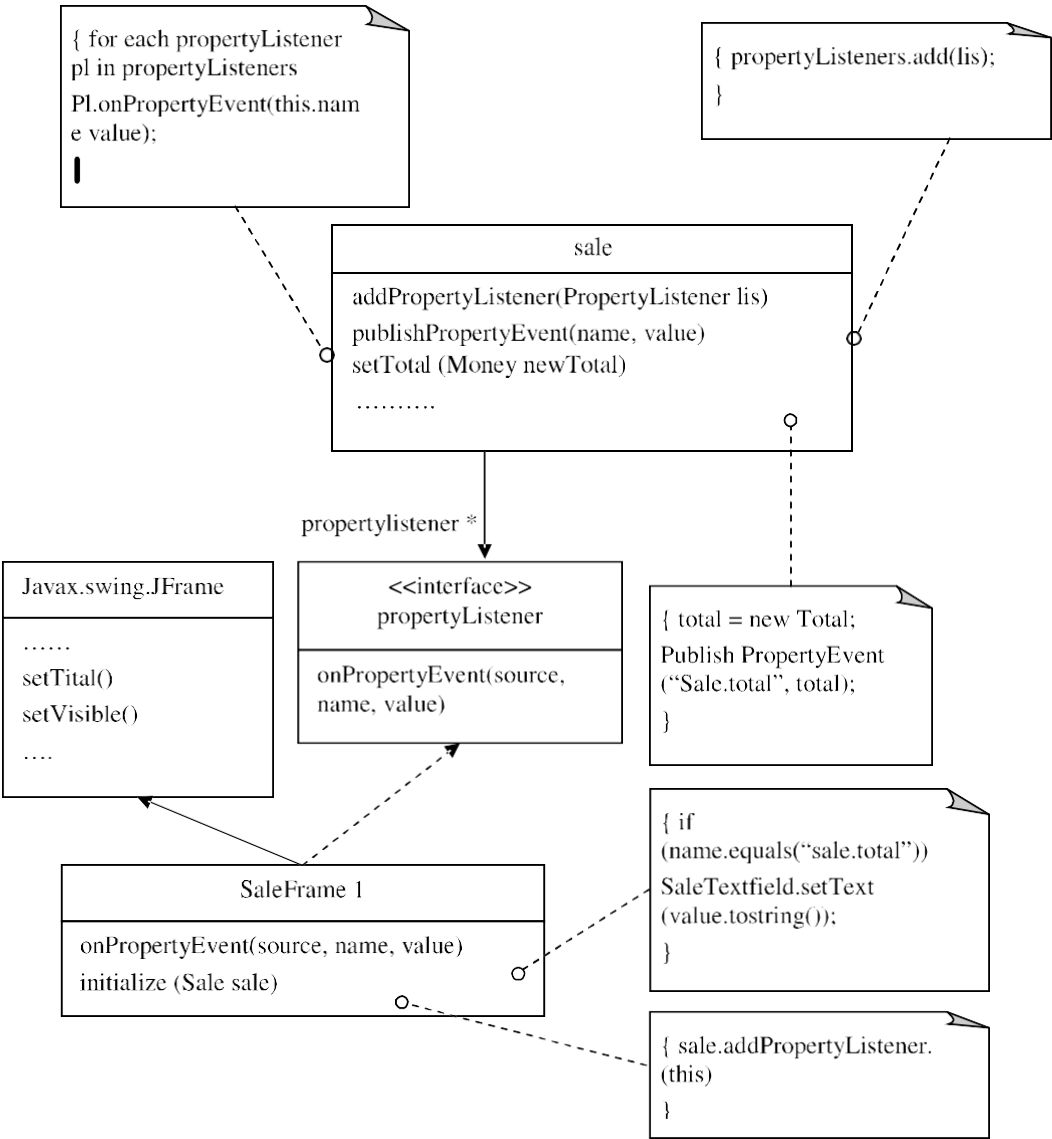


Figure 4.28 The observer pattern

Due to low coupling the replacement of the view or presentation layer by a new one or of particular windows by new windows.

The variations are supported by model view separation.

←—————→

To solve this problem observe patterns are used.

Name : Observer (Publish-subscribe)

Problem : Subscriber objects are interested in state changes or events of a publisher object, want to react in their own unique way when the publisher generates an event.

Solution : Subscriber-implementations this interface. Publisher dynamically registers subscribers

interested in an event and notifies them when event occurs.

The above diagram shows sample solution.

The ideas in this example:

1) An interface is defined

Example: PropertyListener with operation OnPropertyEvent

2) Define window to implement interface.

Example: SaleFrame1 implements method OnPropertyEvent

3) Pass the sale instance from where it is displaying total to SaleFrame window.

4) SaleFrame1 window registers or subscribes to Saleinstance for notification of -Property events in a addPropertyListener message.

5) Sale does not know about SaleFrame1 objects. This lowers coupling.

Sale instance is a publisher of -Property events.

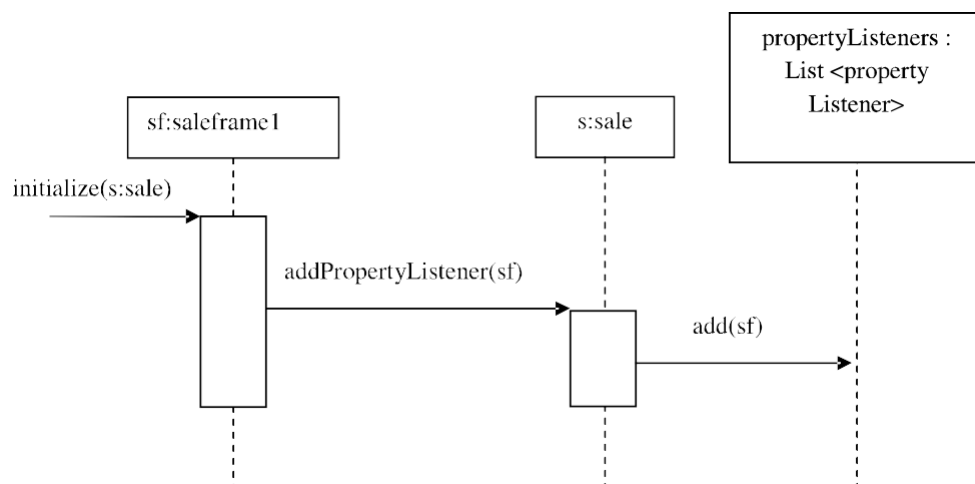


Figure 4.29 The observer SaleFrame1 subscribes to publisher sale

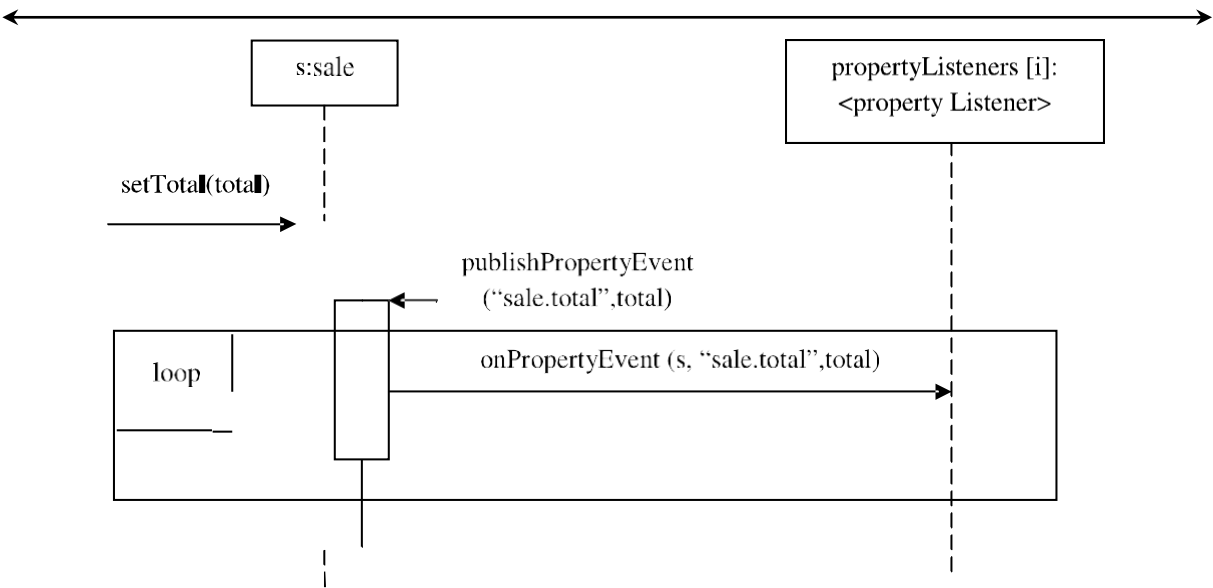


Figure 4.30 Sale publishes a property event to all subscribers

Applying UML

The OnPropertyEvent message is polymorphic in the above interaction diagram.

In the diagram below, SaleFrame1 implements PropertyListener interface thus implements onPropertyEvent.

SaleFrame1 on receiving the message sends to JTextFieldGUI widget object to refresh with new Saletotal.

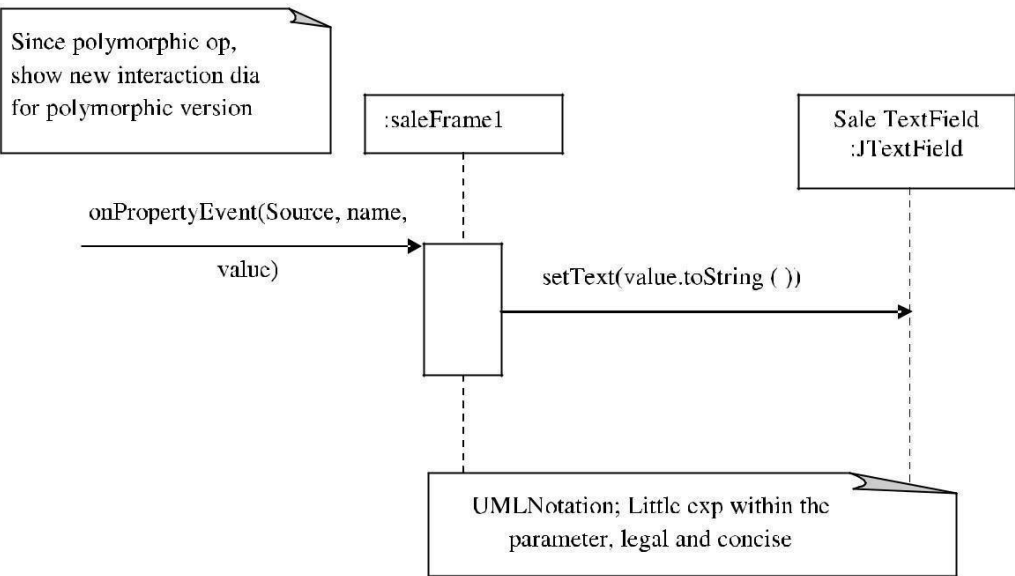


Figure 4.31 Subscriber receives notification of published event



Here there is loose coupling to the interface independent of presentation layer – PropertyListener interface.

Coupling to generic interface need not be present, which can be dynamically added or removed supports low coupling.

Thus protected variations is achieved through use of an interface and polymorphism.

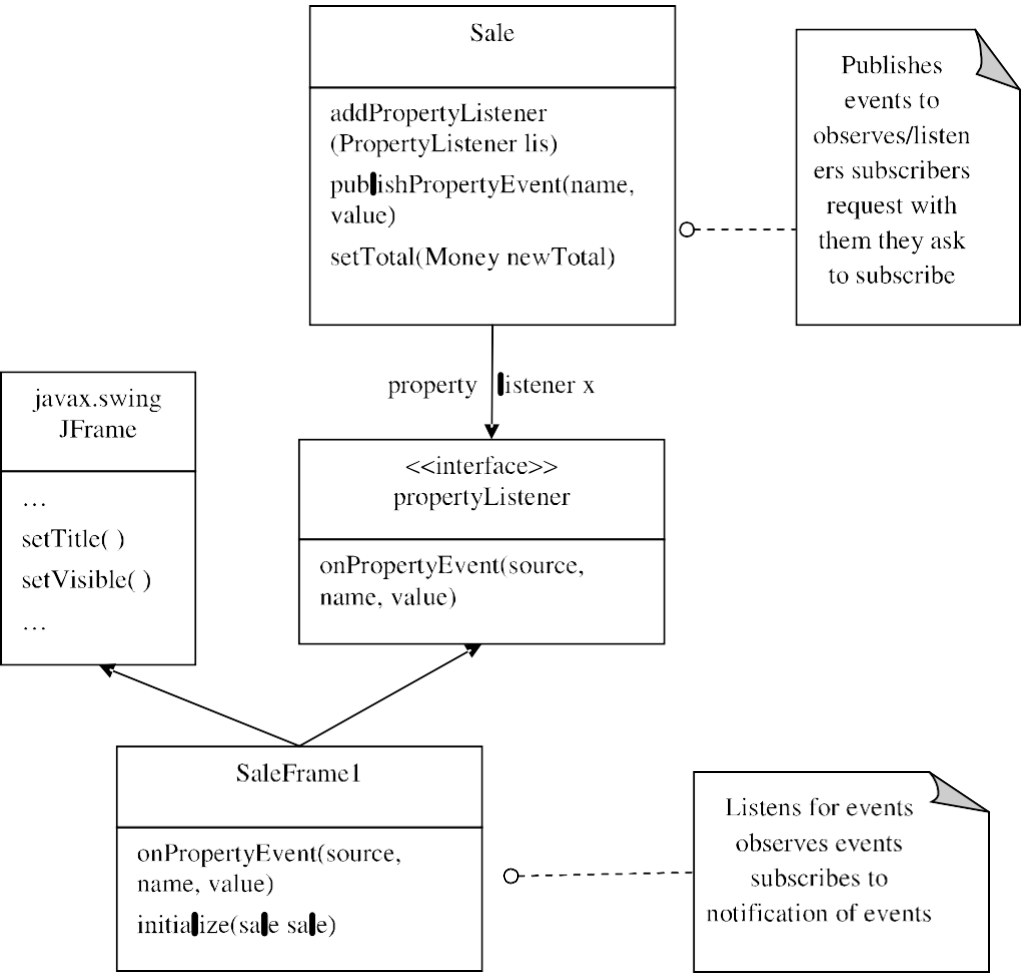


Figure 4.32 Who is observer, listener, subscriber and publisher

The idiom was —publish-subscribe and hence called by that name.

It is observing the event and hence called —observer.

It is also called as —Delegation Event Model in Java because publisher delegates handling of events.

The observer is not only for connecting UIs and model objects.

← This is used in GUI widget event handling in Java technologies-AWT and swing and in Microsoft.NET. →

Example: JButton in swing publishes —action event when it is pressed.

Another example —ALARM CLOCK - publishes Alarm events and various subscribers.

Here AlarmListener interface, many objects can be registered listeners and all can react to —alarm event.

One publisher can have many subscriber for an event

One publisher instance can have zero to many registered subscribers.

Example: One instance of an Alarmclock could have 3 registered subscribers.

- Alarm windows
- Four beepers
- One reliability WatchDog

The OnAlarmEvent notifies eight of these AlarmListeners when event happens.

Implementation

Events:

In java and C#.NET implementation of observer, `_event` is shown as regular message such as `onPropertyEvent`.

In these cases, the event is defined as class and filled with event data.

Event is then passes as a parameter in event message.

Example:

```
class PropertyEvent extends Event
{
    private object sourceOfEvent;
    private string propertyName;
    private object oldvalue;
    private object newvalue;
    //...
}
```

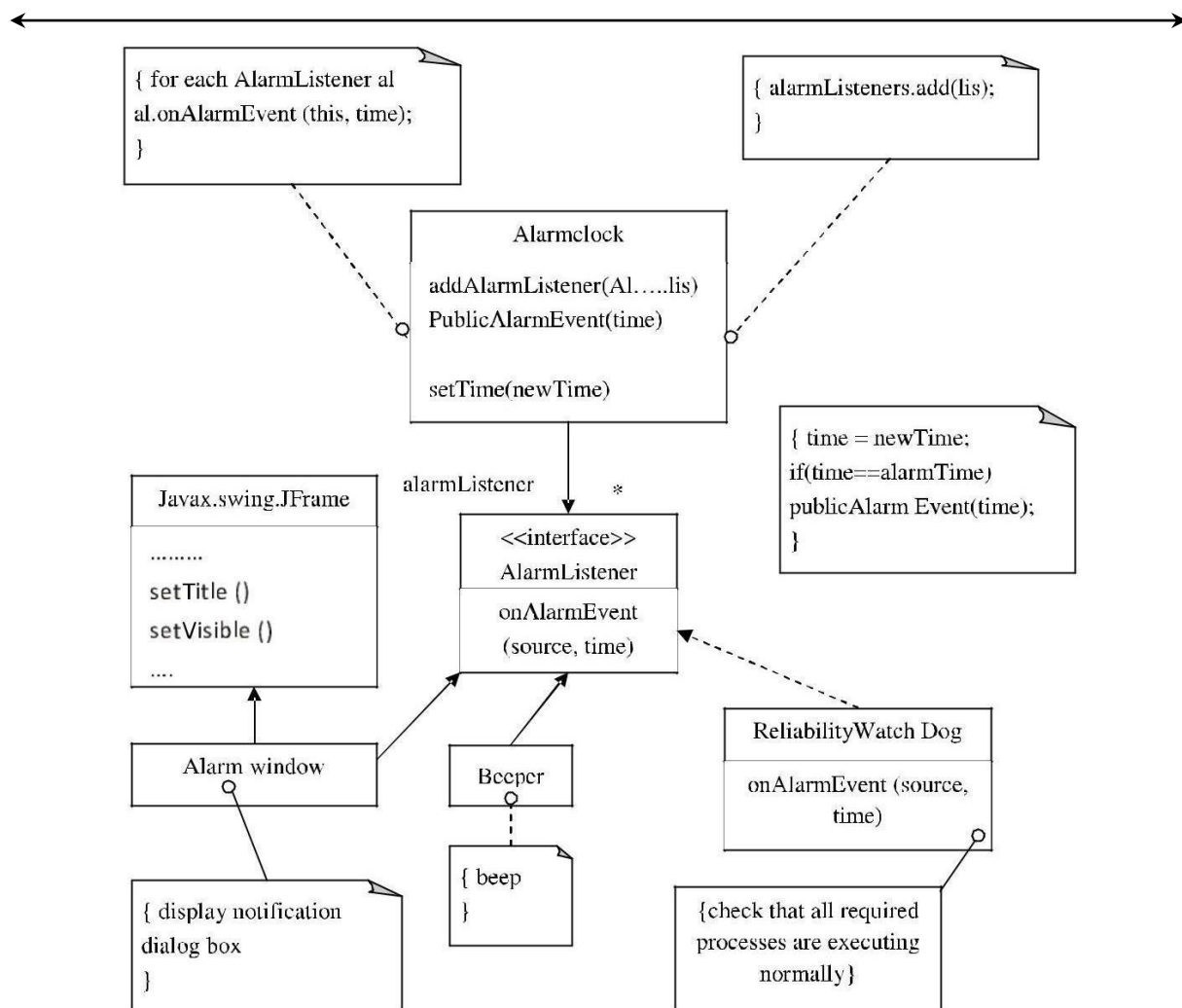



Figure 4.33 Observer applied to alarm events with different subscribers

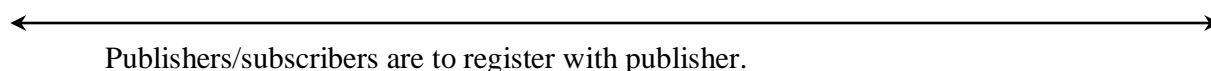
```

class sale
{
    private void publishPropertyEvent (string name, object old, object new)
    {
        PropertyEvent evt = new PropertyEvent (this, —sale-total, old, new); for each
        AlarmListener al in alarm Listeners al.onPropertyEvent (evt);
    }
    //...
}
  
```

Java

In 1996, the observable-observer design was replaced by Java Delegation Event Model (DEM) version of publish-subscribe.

Observer thus provides a way to couple objects for communication.



We conclude that,

Observer is based on polymorphism that provides protected variations for protecting publishers from knowing specific class of object and number of objects that communicates with publisher to generate event.

4.12 APPLYING GOF DESIGN PATTERNS

GOF – Gang Of Four

Here we see use case realizations for NextGen case study.

Gang Of Four Design Patterns

There are 23 patterns useful during object diagram.

Out of the 23 patterns, 15 are common and most useful.

4.13 MAPPING DESIGN TO CODE

In the implementation Model, the UML artifacts created during the code generation is used as input. The implementation artifacts consists of source code, database definitions, JSP/XML/HTML pages and so forth.

Java

C#

Visual Basic

C++

Small talk

Python


and many more languages use the object oriented design principles and map to code.

4.13.1 Programming – Iterative and Evolutionary Development

- Modern development tools are used to design while programming.
- The artifacts in Design Model provides the information to code.
- The strength of OOA/D and OO programming is they provide end•to•end road map from requirements to code.
- The point of having a roadmap provides starting point for experimentation.

Creativity and change during implementation

- Some decision making and creative work are done during design.
- Generation of code is a mechanical translation process.
- The results generated during design modeling are initially incomplete and later during programming and testing changes will be made.

- 
- To encounter the new problems during programming → ideas and understanding are generated during OO design modeling.
 - Always plan for lot of change and derivation from the design during programming. This is a key for iterative and evolutionary methods.

Mapping Designs to Code

Implementation requires writing source code in object oriented language for

- Class and interface definitions.
- Method definitions

The code generation is more or less independent of using UML tool.

Creating Class Definition from DCDS

DCDs depict

- Class or interface name
- Super classes
- Operation signature
- Attributes of a class

This is enough to create a basic class definition in OO language.

Defining a class with method signatures and attributes

The mapping to attribute definitions for SalesLineItem is shown

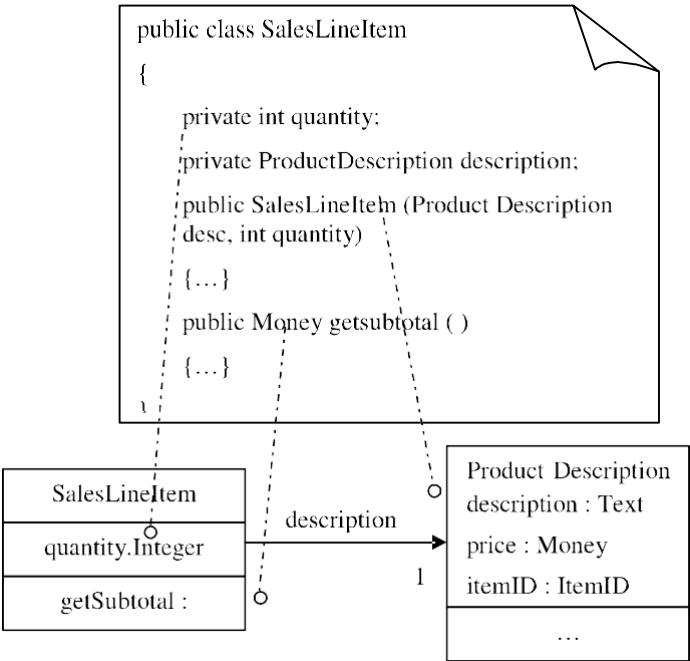


Figure 4.34 SalesLineItem in Java

The sequence of messages in an interaction diagram translates to series of statements in the method definitions.



```
public void enterItem (itemID, ietmID, int qty)
```

```
ProductDescription desc = catalog.getProductDescription(itemID);
```

```
CurrentSale.makeLineItem (desc, qty);
```

The complete enterItem method and its relationship to interaction diagram is given below

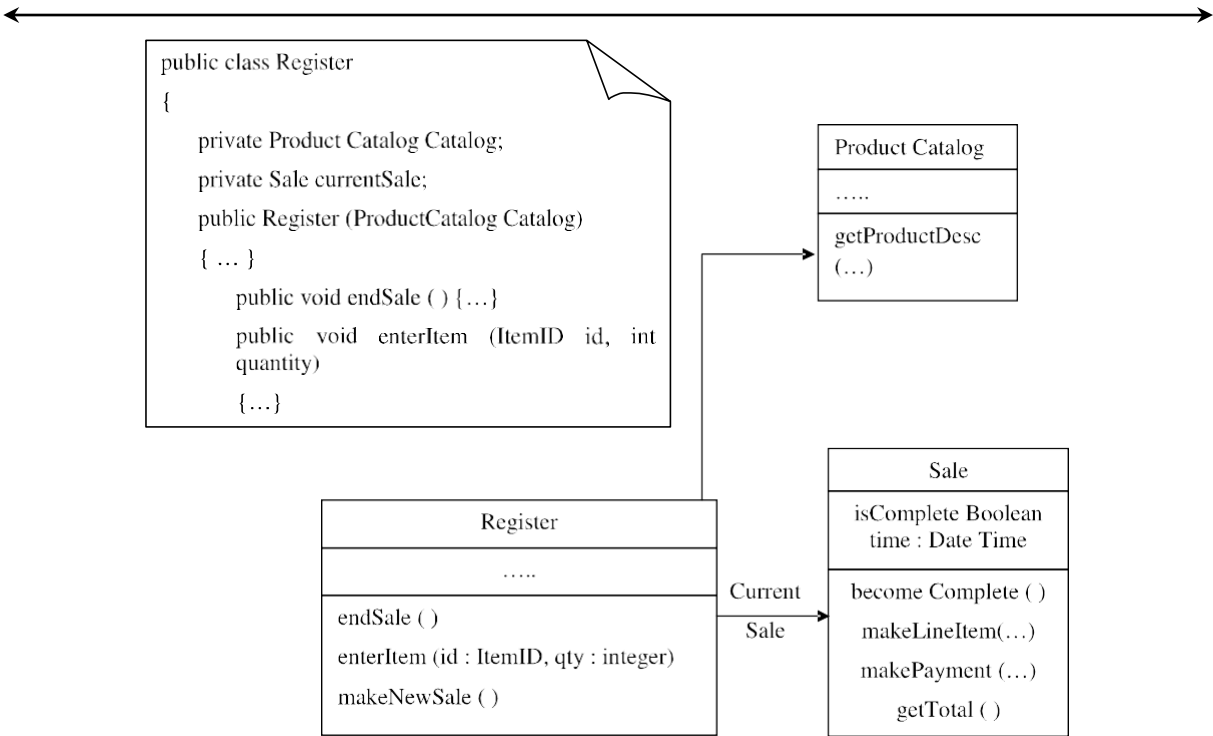


Figure 4.36 The Register enterItem Method

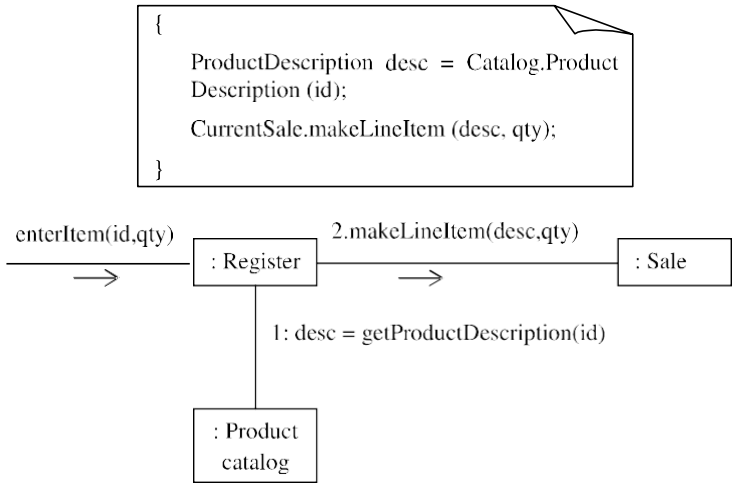


Figure 4.37 enterItem method

Collection Classes in Code

There are many one to many relationships. These are implemented with the collection object such as List or Map or array.

Example:

Sale must contain many SalesLineItem instances.

In Java List and Map interfaces are represented by ArrayList and HashMap.



The requirements for collection class are

- (i) Key based lookup of a map
- (ii) Ordered list requires a List.

LinesItems is described in terms of interface.

```
Private List LineItems = new ArrayList ( );
```

Exceptions and Error Handling

In applications development consider large – scale exception handling strategies during design modeling.

Exceptions are indicated in the property strings of messages and operation declarations.

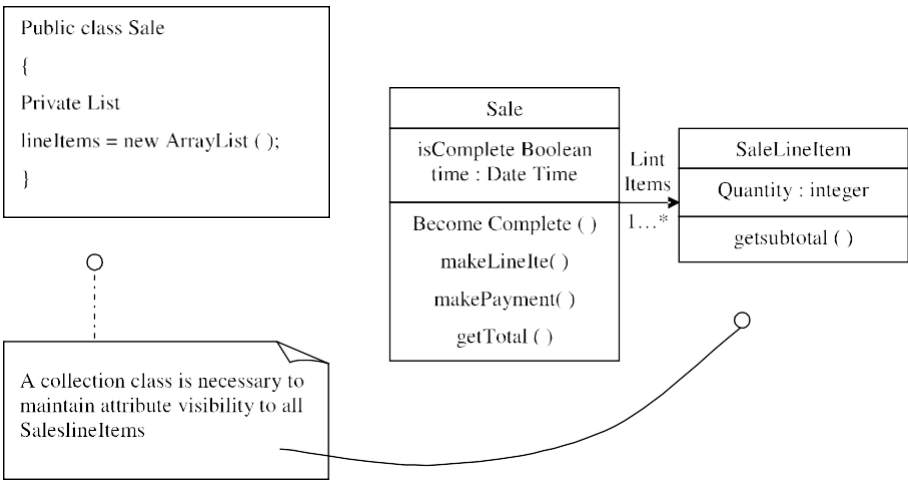


Figure 4.38 Adding a Collection

Definition Sale makeLineItem Method

The makeLineItem method of class sale can be written by inspecting enterItem collaboration diagram.

Sale.makeLineItem method

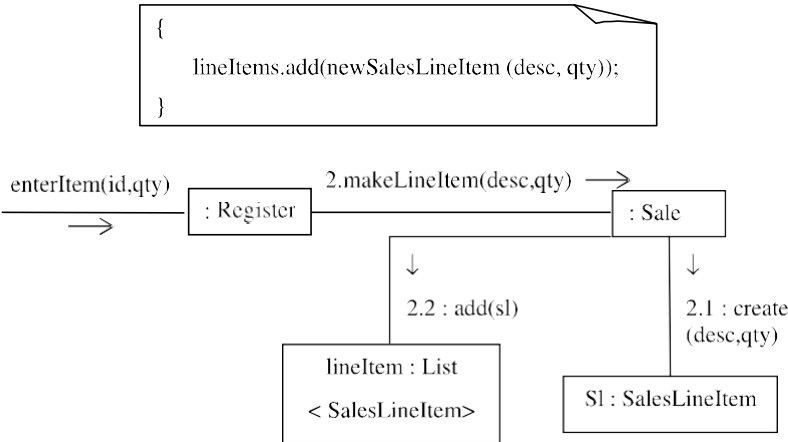


Figure 4.39 Sale.makeLineItem method

Order of Implementation

Implementation of least coupled classes is done followed by most coupled classes.

Example:

First classes payment/product Description is implemented.

Next Product catalog / SaleLineItem are implemented.

Possible order of class Implementation and testing

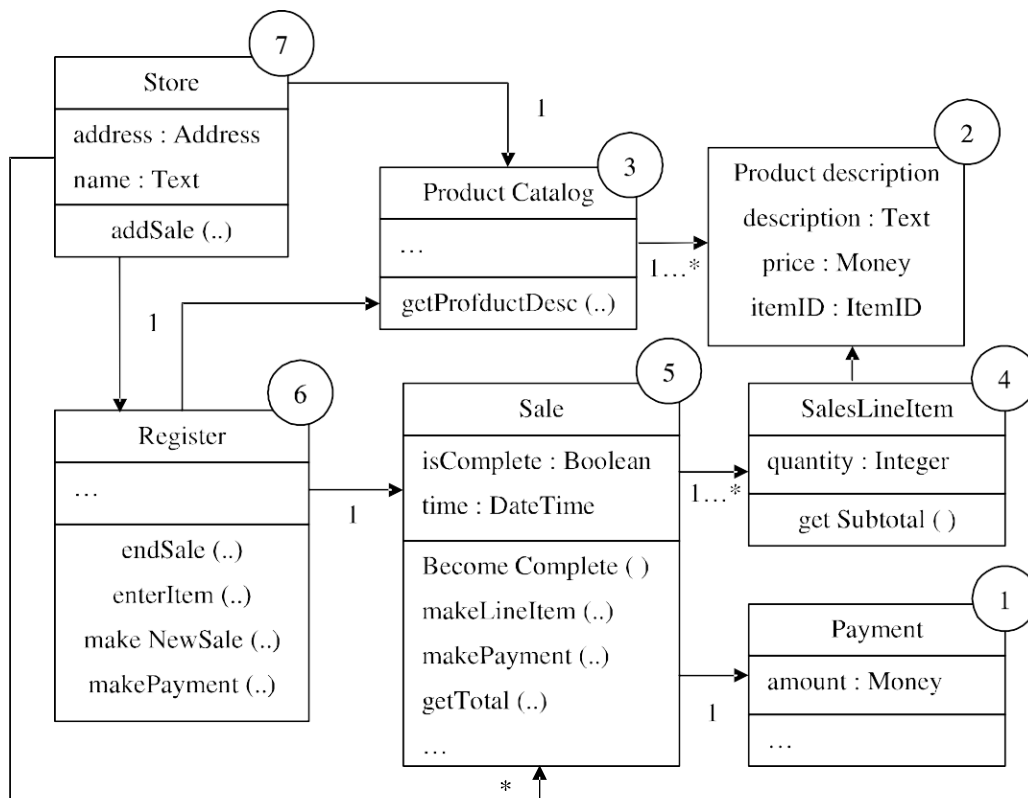


Figure 4.40 Class Implementation and testing

Test Driven of Test First Development

Test Driven Development (TDD) or Test First Development is promoted by Extreme Programming (XP).

Here developer writes unit testing code before testing it.

The rhythm is to write a little test code, then write production code, make it pass the test write some more test code and so forth.

Summary of Mapping Design to Code

There is a translation process from UML class diagrams to class definitions and from interaction diagrams to method bodies.



Introduction to NextGen POS Program Solution

A simple domain layer of classes in Java are largely derived from the design class diagrams and interaction diagrams.

There is a translation from design artifacts to a foundation code.

Class Payment

```
//all classes in package
package com too.nextgen.domain;
public class Payment
{
    private money amount;
    public Payment (Money cash tendered)
    {
        amount = cashTendered;
    }
    public Money getAmount ( )
    {
        return amount;
    }
}
class Productcatalog:
{
    private Map<ItemID, ProductDescription>
    description = newHashMap ( )
    <ItemID, ProductDescription>;
    public ProductCataglog( )
    {
        // Sample Data
        ItemID id1 = new ItemID (10);
        ItemID id2 = new ItemID (20);
        money price = new Money (3);
        product Description d;
        d = new Product Description (id1, price, __Product 1');
        description put (id1, d);
    }
}
```





```
        d = new Product Description (id2, price, __Product 2');
        description put (id2, d);
    }
    public Product Description getProductDescription (ItemID id)
    return description get (id);
}
class Register:
public class Register
{
    private Product Catalog Catalog;
    private Sale currentSale;
    public Register (ProductCatalog Catalog)
    {
        this.catalog = catalog;
    }
    public void endSale ( )
    {
        currentSale.become Complete ( );
    }
    public void enterItem (ItemID id, int quantity)
    {
        productDescription desc = catalog.getProductDescription (id);
        currentSale.makeLineItem (desc, quantity);
    }
    public void makeNewSale ( )
    {
        currentSale = newSale ( );
    }
    public void makePayment (Money cashTendered)
    {
        currentSale.makePayment( cashTendered);
    }
}
```



```
class Productdescription
public class ProductDescription
{
    private ItemID id;
    private Money price;
    private String descripiton;
    public ProductDescription
    (ItemID id, Money price, string descriptio )
    {
        this.id = id;
        this.price = price;
        this.description = description;
    }
    public ItemID getItemID ( ) {return id;}
    public Money getPrice ( ) {return price;}
    public string getDescription {return description;
}
class Sale:
public class Sale
{
    private List <SalesLineItem> lineItems = new ArrayList ( ) <SalesLineItem>;
    private Date date = new Date ( );
    private Boolean is complete = false;
    private Payment Payment;
    public Money getbalance ( )
    {
        return Payment getAmount ( ).minus (getTotal( ));
    }
    public void become Complete ( ) {isComplete = true;}
    public Boolean is complete ( ) {return is complete;}
    public void makeLineItem (Product description desc, int quantity)
    {
        linesItems.add (new SalesLinesItem (desc, quantity));
```



```
    }  
    public Money getTotal ( )  
    {  
        Money Total = new Money ( );  
        Money subtotal = null;  
        for (SalesLineItem lineItem : lineItem)  
        {  
            subtotal = lineItem.getsubtotal ( );  
            total.add (subtotal);  
        }  
        return total;  
    }  
    public void makePayment (Money cashTendered)  
    {  
        payment = new Payment (cash Tendered);  
    }  
}  
class SalesLineItem  
public class SalesLineItem  
{  
    private int quantity;  
    private ProductDescription description;  
    public SalesLineItem (Product Description desc, int quantity)  
    {  
        this.description = d; this.quantity = quantity;  
    }  
    public Money getsubtotal ( )  
    {  
        return description.getPrice ( ).times (quantity);  
    }  
}  
class store  
public class store
```



```
{  
    private ProductCatalog catalog = new Product Catalog( );  
    private Register register = new Register(Catalog);  
    public Register getRegister ( )  
    {return register;}  
}
```

Introduction to Monopoly Program Solution

Here is a sample domain layer of classes in Java.

Class Square

```
//all classes are in the package  
package.com.too.monopoly.domain;  
public class square  
{  
    private string name;  
    private square nextsquare;  
    private int index;  
    public square (string name, int index)  
    {  
        this.name = name; this.index = index;  
    }  
    public void setNextSquare (square S)  
    {  
        nextSquare = S;  
    }  
    public Square getnextSquare ( )  
    {  
        return nextSquare;  
    }  
    public string getname ( )  
    {  
        return name;  
    }  
    public int getIndex ( )  
    {  
        return index;  
    }  
}
```

←-----→
}

Class Piece

```
public class Piece
{
    private Square location; public Piece (Square location)
    {
        this.location = location;
    }
    public Square getLocation ( )
    {
        return location;
    }
    public void setLocation (Square location)
    {
        this.location = location;
    }
}
```

Class Die

```
public class Die
{
    public static final int MAX = 6; private int faceVlaue;
    public Die ( )
    {
        roll ( );
    }
    public void roll ( )
    {
        faceValue = (int) ((Math.random ( ) * MAX) + 1);
    }
    public int getfaceValue ( )
    {
        return faceValue
    }
}
```



Class Board

```
public class Board
{
    private static final int SIZE = 40;
    private List squares = new ArrayList (SIZE);
    public Board ( )
    {
        build squares ( );
        link Squares ( );
    }
    public Square getSquare (Square start, int distance)
    {
        int endIndex = (start.getIndex( ) + distance) %size;
        return (Square) Square.get (endIndex);
    }
    public Square getstartSquare ( )
    {
        return (Square) Squares.get (0);
    }
    private void bulidSquare ( )
    {
        for (int i=1 i<=SIZE; i++)
        {
            build (i);
        }
    }
    private void build (int i)
    {
        Square S = new Square (—Squarel +I, i-1);
        Square.add(S);
    }
    private void linkSquare ( )
    {

```



```
        for (int i=0;i<(SIZE•1); i++)
        {
            link (i);
        }
        Square first = (square) Square.get(0);
        Square last = (square) Square.get(SIZE•);
        last.setNextSquare (first);
    }
    private void link (int i)
    {
        Square current = (Square) Squares.get(i);
        Square next = (Square) Squares.get(i+1);
        current.setNextSquare (Next);
    }
}
```

Class player

```
public class player
{
    private string name;
    private Piece piece;
    private Board board;
    private Die [ ] dice;
    public player (string name, Die [ ] dice, Board board)
    {
        this.name = name;
        this.dice = dice;
        this.board = board;
        piece = new Piece (board.getstartsquare ( ));
    }
    public void take turn ( )
    {
        int roll Total = 0;
```

Class Monopoly Game

```
public Class Monopoly Game
{
    private static final int ROUNDS – TOTAL = 20;
    private static final int PLAYERS – TOTAL = 2;
    private List players = new ArrayList (PLAYERS – TOTAL);
    private Board board = new Board ( );
    private Die [ ] dice = {new Die, new Die ( )};
    public Monopoly Game ( )
    {
        player p;
        p = new player (—Horsell, dice, board);
        players.add (p);
    }
    public void palyGame ( )
```




```
{
    for (int i=0; i<ROUNDS – TOTAL; i++)
    {
        play Round ( );
    }
}
public List getPlayers ( )
{
    return players;
}
private void playground ( )
{
    for (iterator iter = Players.iterator ( ); iter.hasNext ( );)
    {
        player player = (Player)iter.Next ( );
        player.takeTurn ( );
    }
}
}
```