UNIT IV
Design Patterns

GRASP: Designing objects with responsibilities – Creator – Information expert – Low Coupling – High Cohesion - Controller

Design patterns – **Creational** - factory method – **Structural** – Bridge - adapter, **Behavioral** – Strategy - observer – Applying GOF design patterns – Mapping design to code.

1. GRASP: DESIGNING OBJECTS WITH RESPONSIBILITIES P-1

After identifying the 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 designing of objects starts with

- Inputs
- Activities
- Outputs

GRASP as a Methodical Approach to Learning Basic Object Design

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

Responsibilities and Methods

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).

The translation of responsibilities into classes and methods is influenced by the granularity of the responsibility.

The responsibility to "provide access to relational databases" may involve dozens of classes and hundreds of methods, packaged in a subsystem.

By contrast, the responsibility to “create a Sale” may involve only one or few methods.

A responsibility is not the samething 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 collabrate with other objects, such assending a *getSubtotal* message to each *SalesLineltem* object asking for its subtotal.

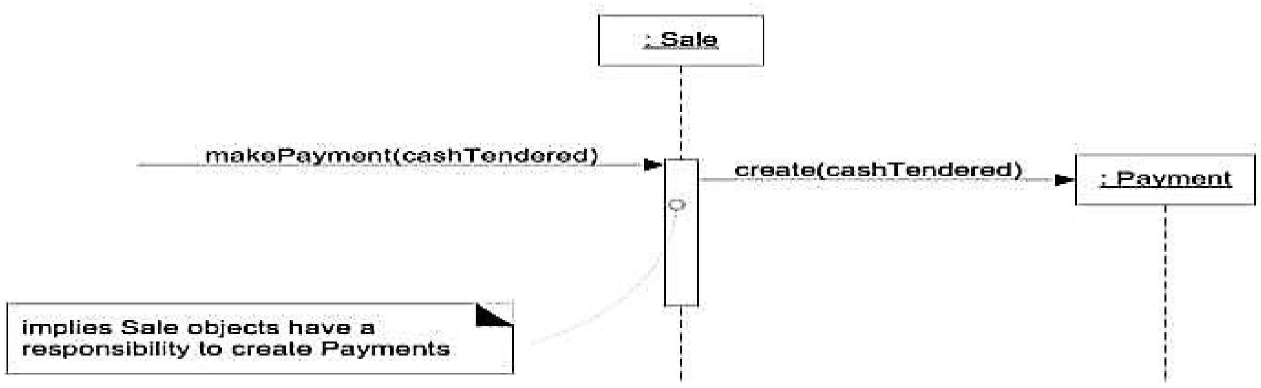

: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
implies Sale objects have a responsibility to create Payments

Figure: Responsibilities and methods are related.

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

The fulfillment of this responsibility requires collaboration to create the SalesLineltem object and invoke its constructor.

Interaction diagrams show choices in assigning responsibilities to objects.

When created, decisions in responsibility assignment are made, which are reflected in what messages are sent to different classes of objects.

Patterns

The principles and idioms, if codified in a structured format describing the problem and solution, and given a name, may be called patterns.

Sample pattern:

Pattern Name: Information Expert

Solution: Assign a responsibility to the class that has the information needed to fulfill it.

Problem It Solves: What is a basic principle by which to assign responsibilities to objects?

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.

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.

How to Apply the GRASP Patterns

The first five GRASP patterns:

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

1. <u>CREATOR</u> <u>P-2</u>

Problem: Who should be responsible for creating a new instance of some class?
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.
Example: In the POS application, who should be responsible for creating a SalesLineItem instance?

➢ Creator guides assigning responsibilities related to the creation of objects, a very common task.

➢ 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:

- Aggregate **aggregates** Part
- Container **contains** Content
- Recorder **records** Recorded

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

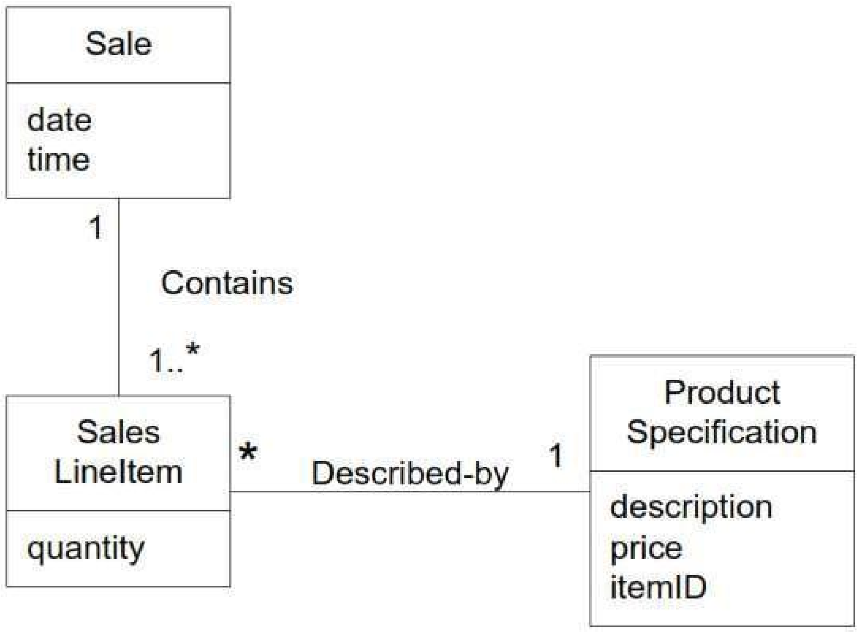


Figure: Partial domain model

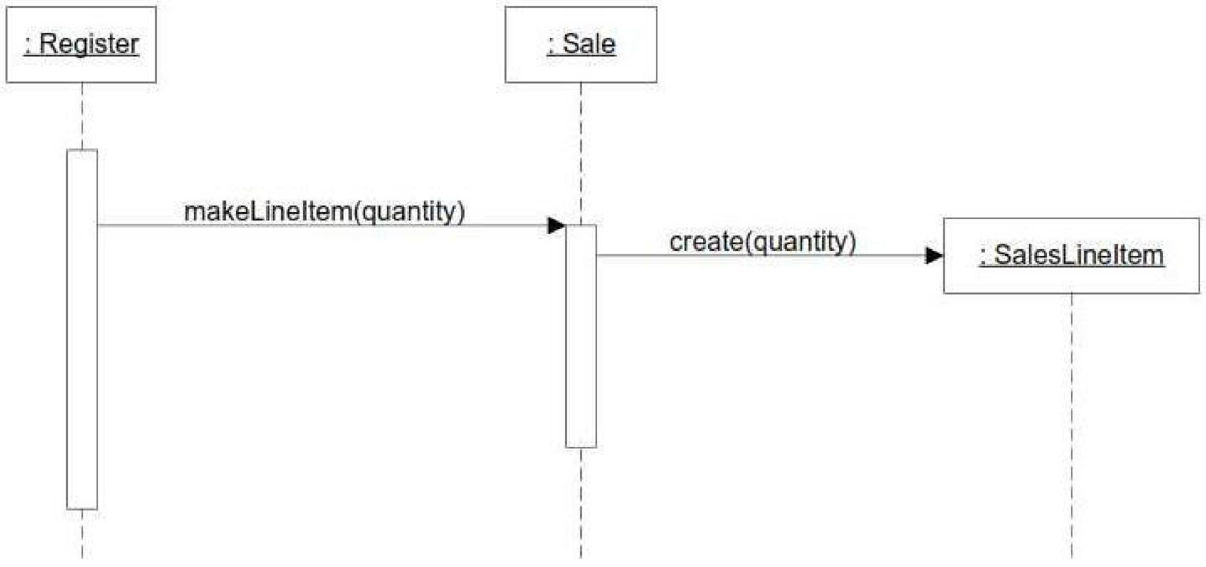


Figure: Creating a SalesLineItem

Benefits:
Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.

2. INFORMATION EXPERT P-3

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

Solution: Assign a responsibility to the information expert - the class that has the ***information*** necessary to fulfil the responsibility.

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

➢ From the suggestion given above, the question is :
Who should be responsible for knowing the grand total of a sale?

Answer:
1. If there are relevant classes in the design model, look there first
2. Or look in the domain model, and attempt to use its representations to inspire the creation of corresponding design classes

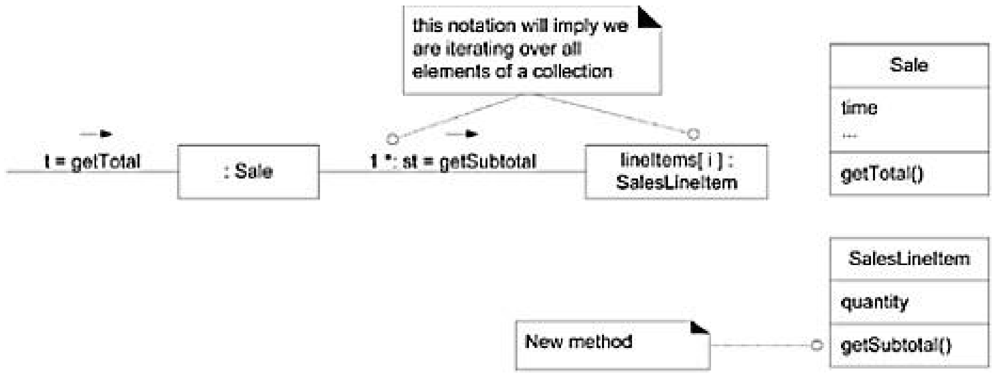

this notation will imply we are iterating over all elements of a collection
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i] : SalesLineItem
Sale
time
...
getTotal()
SalesLineItem
quantity
getSubtotal()
New method

Figure: Calculating the Sale total Figure: Calculating the Sale total

➢ To fulfill the responsibility of knowing and answering the sale's total, we assigned three responsibilities to three design classes of objects as follows:

Design Class	***Responsibility***
Sale	Knows sale total
SalesLineItem	Knows line item subtotal
ProductDescription	Knows product price

Benefits:
➢ Information encapsulation is maintained since objects use their own information to fulfill tasks
➢ Behavior is distributed across the classes that have the requires information

Related Patterns: Low Coupling, High Cohesion

3. LOW COUPLING P-4

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

Solution: Assign a responsibility so that coupling remains low.

- **Coupling** is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements; "too many" is context-dependent, but will be examined.
- A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:
  - ✓ 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.

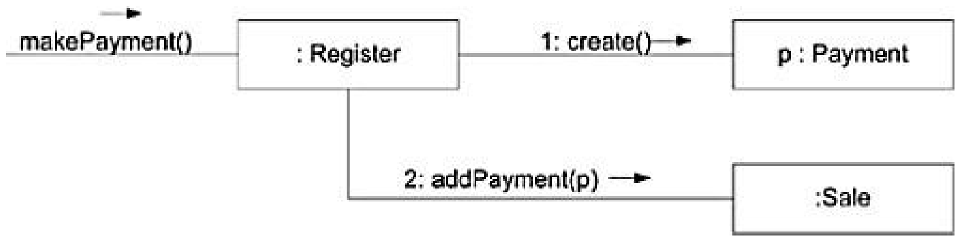


Figure: Register creates Payment

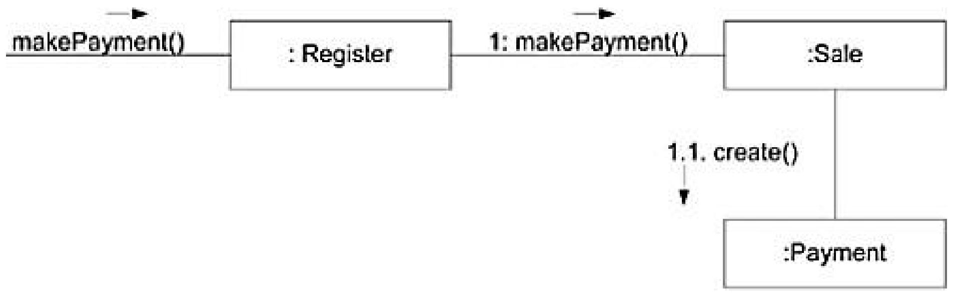


Figure: Sale creates Payment

- In object-oriented languages such as C++, Java, and C#, common forms of coupling from **TypeX** to **TypeY** include:
  - ✓ **TypeX** has an attribute (data member or instance variable) that refers to a **TypeY** instance, or **TypeY** itself.
  - ✓ A **TypeX** object calls on services of a **TypeY** object.
  - ✓ **TypeX** has a method that references an instance of **TypeY,** or **TypeY** itself, by any means. These typically include a parameter or local variable of type **TypeY,** or the object returned from a message being an instance of **TypeY**.
  - ✓ **TypeX** is a direct or indirect subclass of **TypeY.**
  - ✓ **TypeY** is an interface, and **TypeX** implements that interface

- Low Coupling supports the design of classes that are more independent
- Low Coupling encourages assigning a responsibility so that its placement does not increase the coupling to such a level that it leads to the negative results that high coupling can produce.

Benefits

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

4. <u>HIGH COHESION</u>

Problem: How to keep objects focused, understandable and manageable, and as a side effect, support low coupling?

Solution: Assign a responsibility so that cohesion remains high.

- **Cohesion** (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are.
- An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.
- A class with low cohesion does many unrelated things, or does too much work. Such classes are undesirable; they suffer from the following problems:
  - ✓ hard to comprehend
  - ✓ hard to reuse
  - ✓ hard to maintain
  - ✓ delicate; constantly effected by change

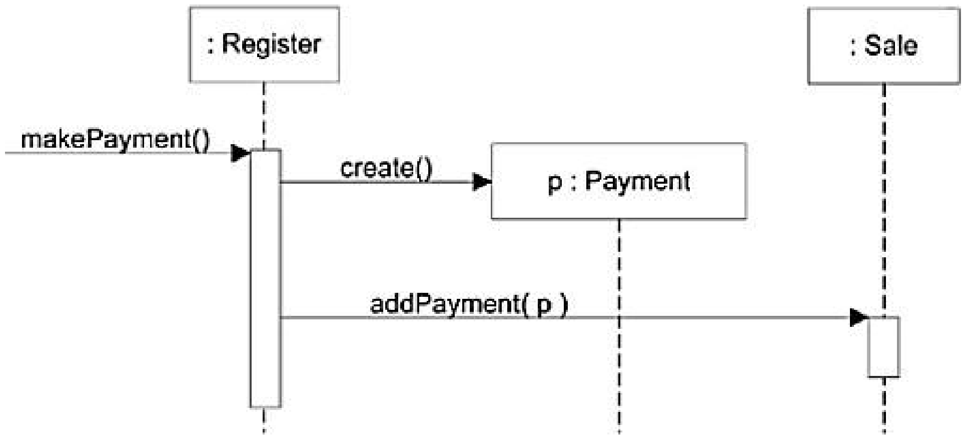


Figure: Register creates Payment

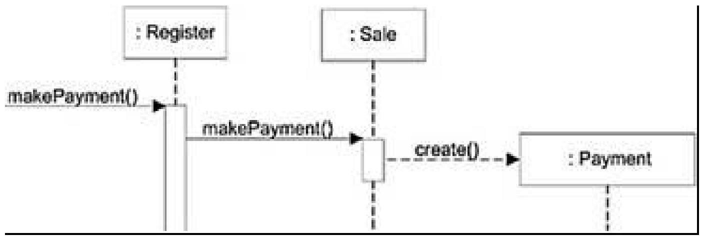


Figure: Sale creates Payment

- The second design supports both high cohesion and low coupling, it is desirable.

- Some scenarios that illustrate varying degrees of functional cohesion:
  2. **Very low cohesion:** A class is solely responsible for many things in very different functional areas.
  3. **Low cohesion:** A class has sole responsibility for a complex task in one functional area.
  4. **High cohesion:** A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
  5. **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.

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.
- High Cohesion is: **easy to maintain, understand and reuse.**
- **Modularity** is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- A modular design creates methods and classes with high cohesion.

Benefits:

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

5. <u>CONTROLLER P-5</u>

Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation?
A **Controller** is the first object beyond the UI layer that is responsible for receiving or handling a system operation message
Solution: Assign the responsibility to a class representing one of the following choices:

- ✓ Represents the overall system, a root object, a device, or a major subsystem (facade controller).
- ✓ Represents a use case scenario within which the system event occurs, often named <UseCaseName> Handler, <UseCaseName

- Use the same controller class for all system events in the same use c> Coordinator, or <Use- CaseName> Session (use-case or session controller).ase scenario.
- 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: Who should be the controller for system events such as enterItem and endSale?

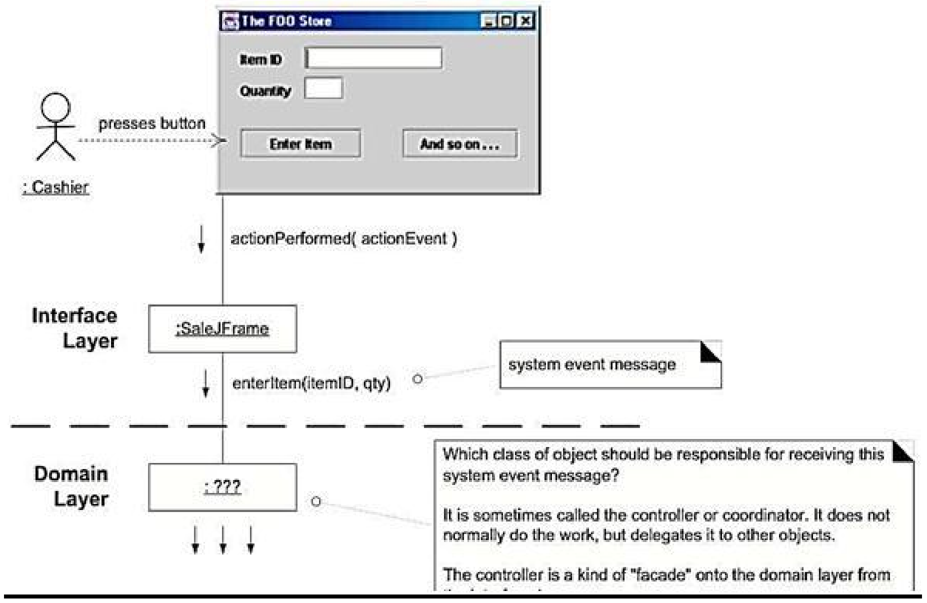

The FOO Store
Item ID
Quantity
Enter Item
And so on...
presses button
: Cashier
actionPerformed(actionEvent)
Interface Layer
:SaleJFrame
enterItem(itemID, qty)
system event message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes 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

Figure: what object should be the controller for enterItem

By the Controller pattern, some choices:

- ✓ represents the overall "system," root object, device, or subsystem
  - ▪ Register, POSSystem
- ✓ represents a receiver or handler of all system events of a use case scenario
  - ▪ ProcessSaleHandler, ProcessSaleSession

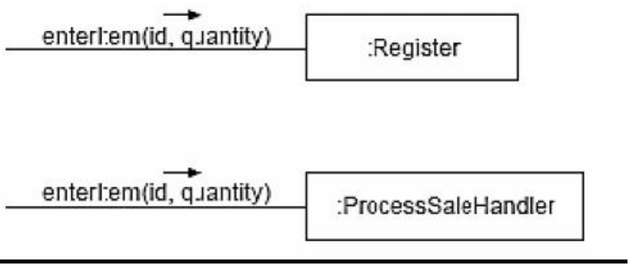


Figure: Controller Choices

Benefits:

- ✓ Increased potential for reuse, and pluggable interfaces
  - ▪ No application logic in the GUI
- ✓ Opportunity to reason about the state of the use case
  - ▪ E.g. operations must be performed in a specific order

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:
  - ✓ Only a single controller class receiving all system events in the system
  - ✓ Controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work
  - ✓ Controller has many attributes and it maintains significant information about the system or domain, which should be distributed to other objects or it

Cure:

- ➢ Add more controllers
- ➢ Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects

Related Patterns:

- ✓ Command
- ✓ Façade
- ✓ Layers
- ✓ Pure Fabrication

2. DESIGN PATTERNS

1. CREATIONAL - FACTORY METHOD P-6

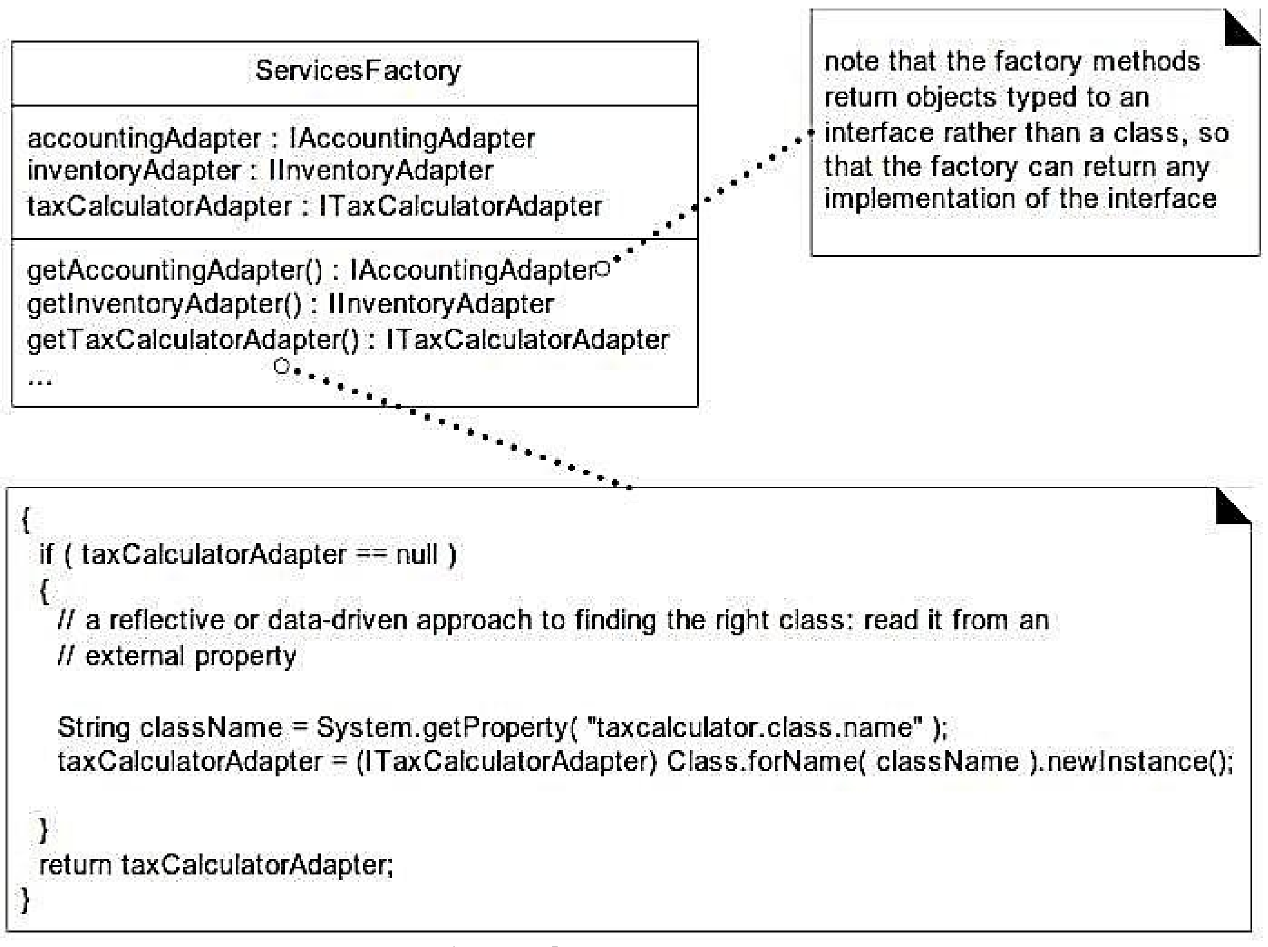
- ➢ This is also called Simple Factory or Concrete Factory.
- ➢ It is also a simplification of the GoF Abstract Factory pattern
- ➢ The adapter raises a new problem in the design:
  - ✓ In the prior Adapter pattern solution for external services with varying interfaces, who creates the adapters?
  - ✓ And how to determine which class of adapter to create, such as TaxMaster - Adapter or Good As GoldTaxPro Adapter?
- ➢ If some domain object creates them, the responsibilities of the domain object are going beyond pure application logic (such as sales total calculations) and into other concerns related to connectivity with external software components.
- ➢ It does not support separation of concerns and it lowers cohesion
- ➢ A common alternative in this case is to apply the **Factory** pattern, in which a Pure Fabrication “factory” object is defined to create objects

Name :	**Factory**
Problem :	Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?
Solution (advice)**:**	Create a Pure Fabrication object called a Factory that handles the creation.

Advantages of Factory:

- ➢ Separate the responsibility of complex creation into cohesive helper objects. Hide potentially complex creation logic.
- ➢ Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

Related Patterns: Singleton Pattern


ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
{
  if ( taxCalculatorAdapter == null )
  {
    // a reflective or data-driven approach to finding the right class: read it from an
    // external property

    String className = System.getProperty( "taxcalculator.class.name" );
    taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

  }
  return taxCalculatorAdapter;
}
```

Figure: The Factory Pattern

2. STRUCTURAL – BRIDGE P-7

Problem: To decouple abstraction from implementation so that the two can vary independently

Motivation: delegate implementations

- This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes' independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

Applicability:

- ✓ When interface & implementation should vary independently
- ✓ Require a uniform interface to interchangeable class hierarchies

Consequences

- ✓ abstraction interface & implementation are independent
- ✓ implementations can vary dynamically
- ✓ Can be used transparently with STL algorithms & containers
- ✓ one-size-fits - all Abstraction & Implementor interfaces

Implementation

- ✓ sharing Implementors & reference counting
- ✓ See reusable Refcounter template class (based on STL/boost shared_pointer)
- ✓ creating the right Implementor (often use factories)

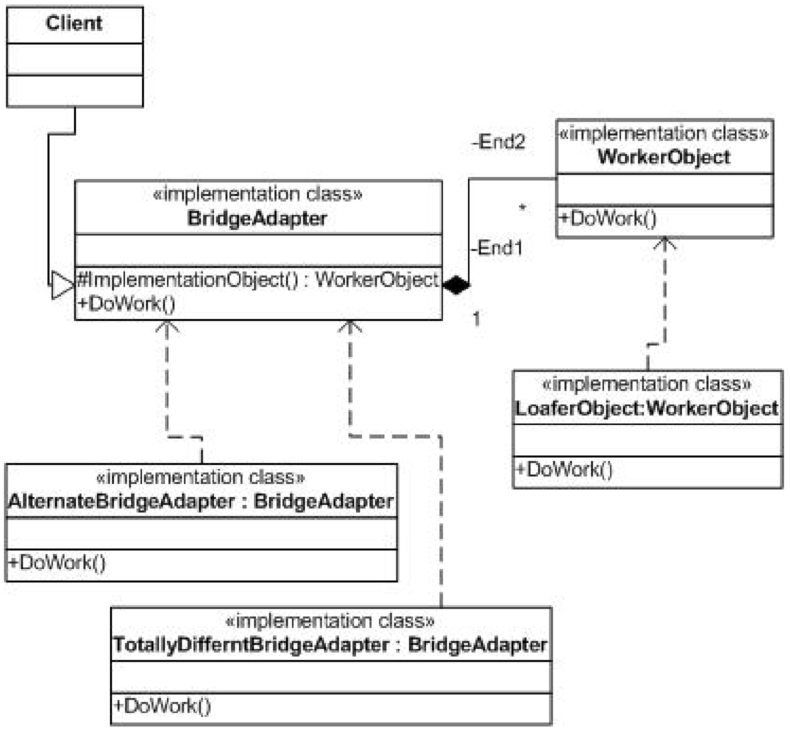


Figure: Bridge Pattern

Known Uses

- ✓ ET++ Window/WindowPort
- ✓ libg++ Set/{LinkedList,HashTable}
- ✓ AWT Component/ComponentPeer

Related Patterns: Abstract Factory Pattern

3. ADAPTER

- ➢ The NextGen problem explored to motivate the Polymorphism pattern and its solution is more specifically an example of the GoF **Adapter** pattern.

Name :	**Adapter**
Problem :	How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
Solution :	(advice) Convert the original interface of a component into another interface, through an intermediate adapter object.

- ➢ **To Review:** The NextGen POS system needs to support several kinds of external third-party services: tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.
- ➢ **Solution**: Add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application.

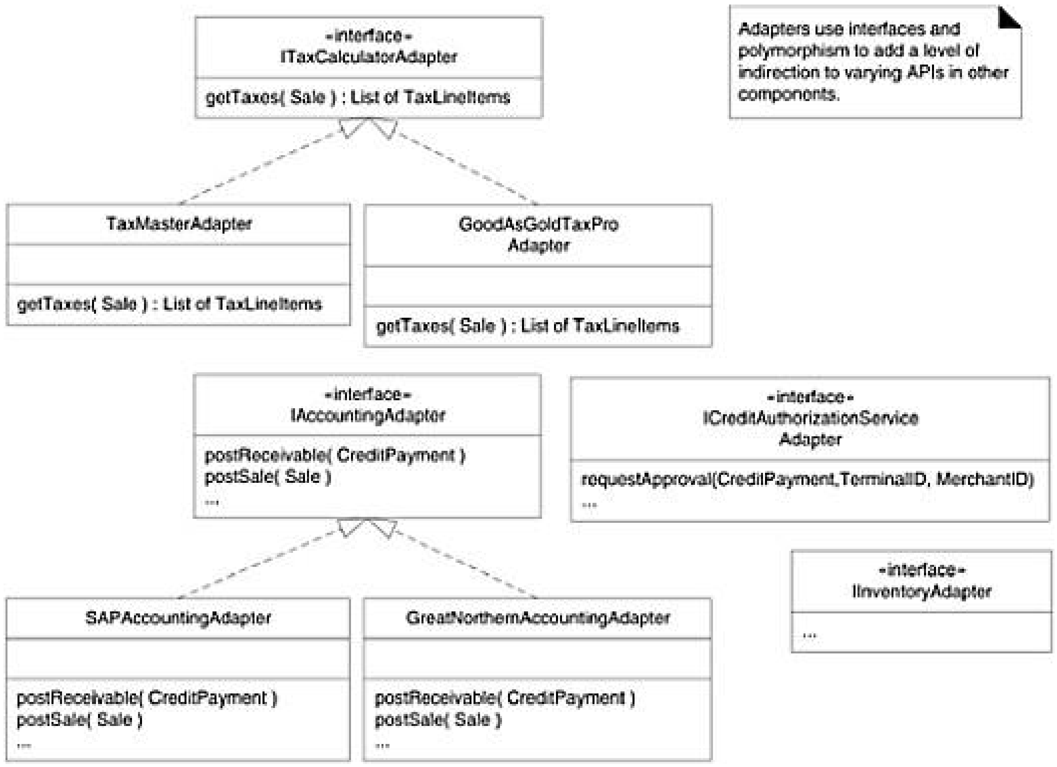

«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
Adapters use interfaces and polymorphism to add a level of indirection to varying APIs in other components.
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
«interface»
IAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
«interface»
ICreditAuthorizationService Adapter
requestApproval(CreditPayment,TerminalID, MerchantID)
...
«interface»
IInventoryAdapter
...
SAPAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...
GreatNorthernAccountingAdapter
postReceivable(CreditPayment)
postSale(Sale)
...

Figure: Adapter Pattern

Related Patterns: Façade object

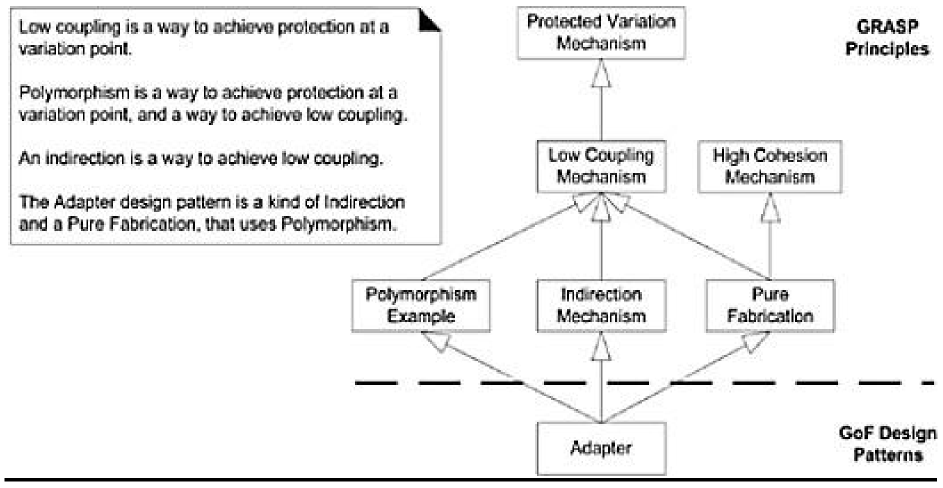

Low coupling is a way to achieve protection at a variation point.
Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.
An indirection is a way to achieve low coupling.
The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.
Protected Variation Mechanism
GRASP Principles
Low Coupling Mechanism
High Cohesion Mechanism
Polymorphism Example
Indirection Mechanism
Pure Fabrication
Adapter
GoF Design Patterns

Figure: Relating Adapter to some core GRASP Patterns

4. BEHAVIORAL – STRATEGY

- Problem: To provide more complex pricing logic, such as a store-wide discount for the day, senior citizen discounts, and so forth.
- The pricing strategy for a sale can vary. How do we design for these varying pricing algorithms?

Name :	**Strategy**
Problem :	How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?
Solution(advice)**:**	Define each algorithm/policy/strategy in a separate class, with a common interface.

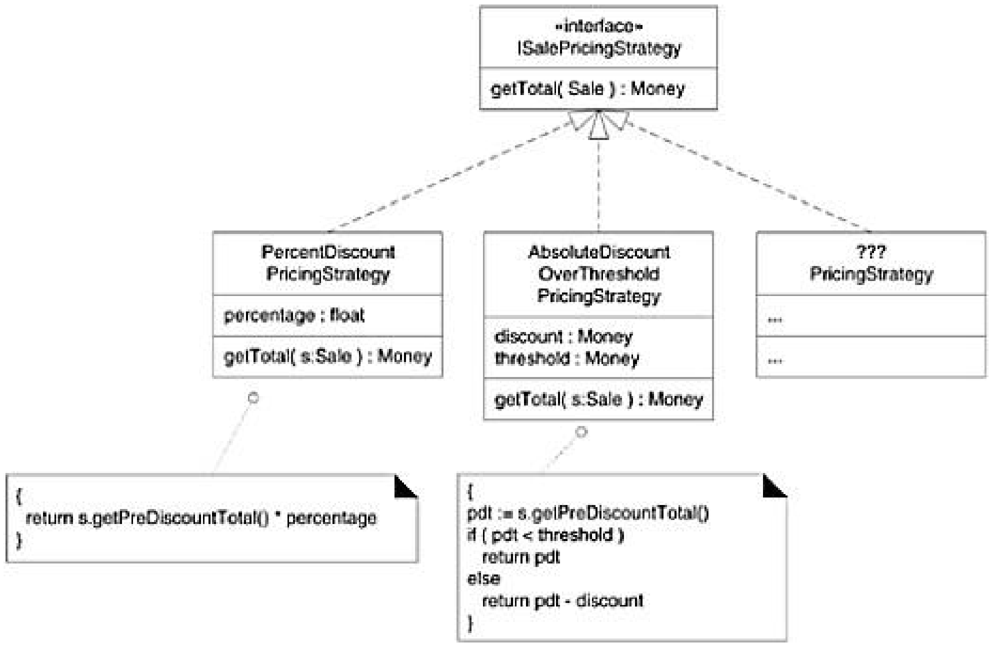

«interface»
ISalePricingStrategy
getTotal(Sale) : Money
PercentDiscount PricingStrategy
percentage : float
getTotal(s:Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(s:Sale) : Money
??? PricingStrategy
...
...
{ return s.getPreDiscountTotal() * percentage }
{ pdt := s.getPreDiscountTotal() if (pdt < threshold) return pdt else return pdt - discount }

Figure: Pricing Strategy Classes

A Strategy object is attached to the **Context – object** – the object to which it applies the algorithm

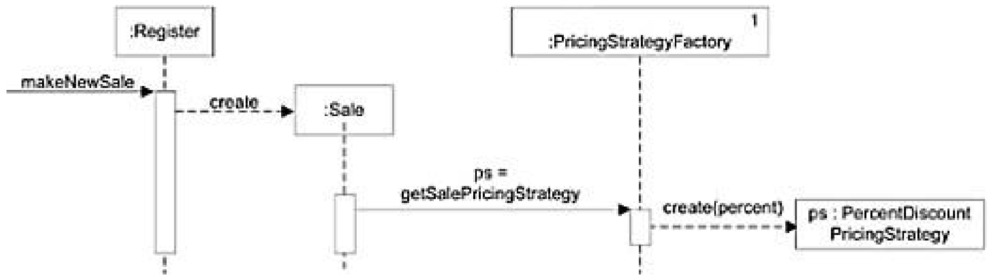


Figure: Creating a Strategy

Related Patterns:

- ✓ Based on Polymorphism and provides Protected Variations
- ✓ Strategies are often created by a Factory

5. <u>OBSERVER</u>

- ➢ To extend the solution for changing data, add the ability for a GUI window to refresh its display of the sale total when the total changes.
- ➢ The model view separation principle discourages the following solution :
  - ✓ When the *Sale* changes its total, the *Sale* object sends a message to a window, asking it to refresh its display.
- ➢ It states that "Model" objects should not know about view or presentation objects such as window. It promotes low coupling.
- ➢ Low coupling allows the replacement of the view or presentation layer by a new one, or of particular windows by new windows, without impacting the non-UI objects.
- ➢ Thus, Model-View Separation supports Protected Variations with respect to a changing user interface.
- ➢ To solve this design problem, the Observer pattern can be used.

Name :	**Observer Pattern**
Problem :	Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
Solution(advice)**:**	Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

Example:
The major ideas and steps in the given example:

1. An interface is defined; in this case, PropertyListener with the operation onPropertyEvent.
2. Define the window to implement the interface. SaleFrame1 will implement the method onPropertyEvent.
3. When the SaleFrame1 window is initialized, pass it the Sale instance from which it is displaying the total.
4. The SaleFrame1 window registers or subscribes to the Sale instance for notification of "property events," via the addPropertyListener message. That is, when a property (such as total) changes, the window wants to be notified.
5. Note that the Sale does not know about SaleFrame1 objects. It only knows about objects that implement the PropertyListener interface. This lowers the coupling of the Sale to the window the coupling is only to an interface, not to a GUI class.
6. The Sale instance is thus a publisher of "property events." When the total changes, it iterates across all subscribing PropertyListeners, notifying each.

The SaleFrame1 object is the observer/subscriber/listener.

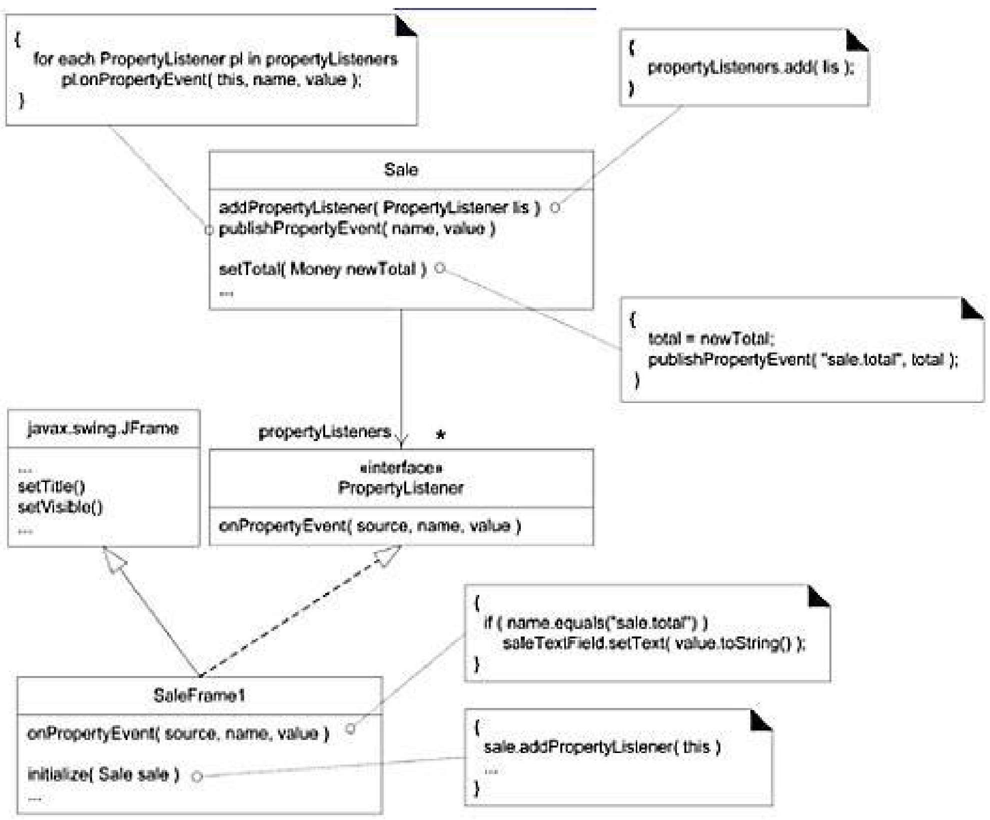

{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

Figure: The Observer Pattern

- In the below figure, it subscribes to interest in property events of the Sale, which is a publisher of property events. The Sale adds the object to its list of PropertyListener subscribers.

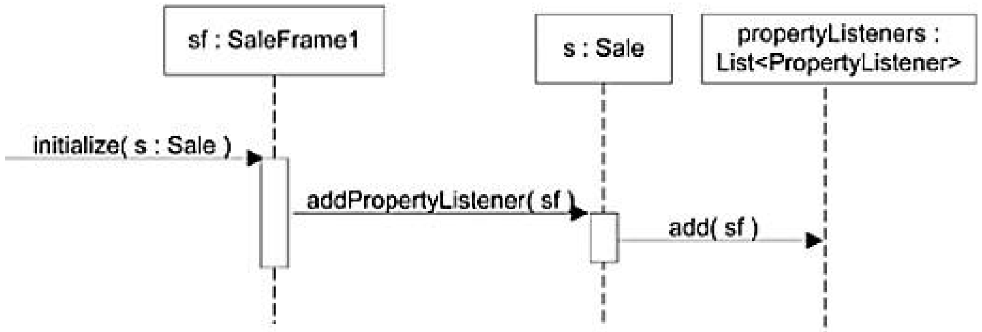

sf : SaleFrame1
s : Sale
propertyListeners : List<PropertyListener>
initialize(s : Sale)
addPropertyListener(sf)
add(sf)

Figure: The Observer SaleFrame1 subscribes to the publisher Sale

- As in the below figure, when the Sale total changes, it iterates across all its registered subscribers, and "publishes an event" by sending the onPropertyEvent message to each.

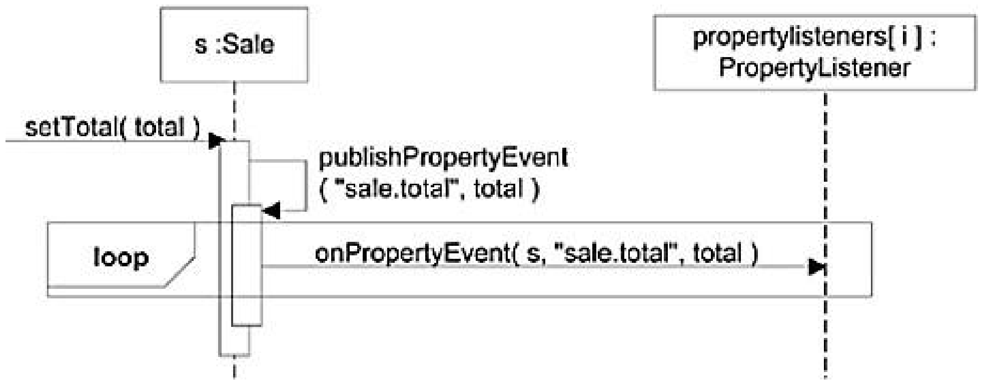

s :Sale
propertylisteners[i] : PropertyListener
setTotal(total)
publishPropertyEvent ("sale.total", total)
loop
onPropertyEvent(s, "sale.total", total)

Figure: The Sale publishes a property event to all its subcribers

- Initially the idiom was called publish-subscribe. One object “publish events” such as the Sale publishing the “property event” when the total changes.
- When the event happens, the registered subscribers are notified by a message
- It has been called Observer pattern because the listener or subscriber is observing the event

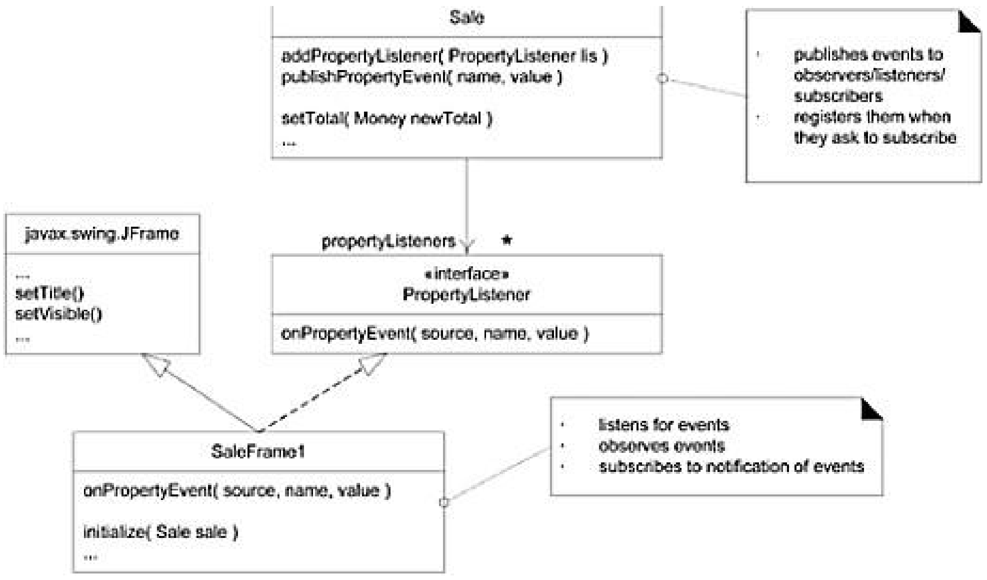

Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
publishes events to observers/listeners/ subscribers
registers them when they ask to subscribe
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface» PropertyListener
onPropertyEvent(source, name, value)
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
listens for events
observes events
subscribes to notification of events

Figure: Who is Observer, Listener, Subscriber and Publisher?

Related Patterns: Polymorphism

6. APPLYING GOF DESIGN PATTERNS P-9

GOF – Gang of Four Design Patterns

Who are the Gang of Four?

- The *Gang of Four* are the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software".
- This important book describes various development techniques and pitfalls in addition to providing twenty-three object-oriented programming design patterns.
- The four authors were Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Gang of Four Design Patterns

- There are 23 patterns useful during object diagram
- Out of 23 patterns, 15 are common and most useful
- Each pattern description includes a link to a more detailed article describing the design pattern and including a UML diagram, template source code and a real-world example programmed using C#.

Creational Patterns

- The first type of design pattern is the *creational* pattern. Creational patterns provide ways to instantiate single objects or groups of related objects. There are five such patterns:

  Abstract Factory: The abstract factory pattern is used to provide a client with a set of related or dependant objects. The "family" of objects created by the factory are determined at run-time.

  Builder: The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

  Factory Method: The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.

  Prototype: The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone. This practise is particularly useful when the construction of a new object is inefficient.

  Singleton**:** The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance.

Structural Patterns

- The second type of design pattern is the *structural* pattern. Structural patterns provide a manner to define relationships between classes or objects.

  Adapter. The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

  Bridge. The bridge pattern is used to separate the abstract elements of a class from the implementation details, providing the means to replace the implementation details without modifying the abstraction.

  Composite. The composite pattern is used to create hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and utilised in a standard manner.

  Decorator. The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behaviour.

  Facade. The facade pattern is used to define a simplified interface to a more complex subsystem.

  Flyweight. The flyweight pattern is used to reduce the memory and resource usage for complex models containing many hundreds, thousands or hundreds of thousands of similar objects.

  Proxy. The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. The proxy provides the same public interface as the underlying subject class, adding a level of indirection by accepting requests from a client object and passing these to the real subject object as necessary.

Behavioural Patterns

- The final type of design pattern is the behavioural pattern. Behavioural patterns define manners of communication between classes and objects.
  - Chain of Responsibility. The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.
  - Command. The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.
  - Interpreter. The interpreter pattern is used to define the grammar for instructions that form part of a language or notation, whilst allowing the grammar to be easily extended.
  - Iterator. The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.
  - Mediator. The mediator pattern is used to reduce coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.
  - Memento. The memento pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.
  - Observer. The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.
  - **State**. The state pattern is used to alter the behaviour of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.
  - Strategy. The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.
  - Template Method. The template method pattern is used to define the basic steps of an algorithm and allow the implementation of the individual steps to be changed.
  - Visitor. The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

7. MAPPING DESIGN TO CODE

Programming and the Development Process

- The prior design work should not be taken to imply that there is no prototyping or design while programming
- Modern development tools provide an excellent environment to quickly explore alternate approaches, and some (or even lots) design-while-programming is usually worthwhile
- The creation of code in an object-oriented programming language, such as Java or C# is not part of OOAD; it is an end goal.
- The artifacts created in the UP Design Model provide some of the information necessary to generate the code.
- Strength of OOAD and OO programming - when used with the UP -is that they provide an end-to-end roadmap from requirements through to code.

Creativity and Change during Implementation

- Some decision-making and creative work was accomplished during design work.
- Generation of the code is a relatively mechanical translation process.

- The results generated during design are an incomplete first step; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved.
- The design artifacts will provide a resilient core that scales up with elegance and robustness to meet the new problems encountered during programming. Consequently, expect and plan for change and deviation from the design during programming.

Mapping Designs to Code

- Implementation in an object-oriented programming language requires writingsource code for:
    - class and interface definitions
    - method definitions

Creating Class Definitions from DCDs

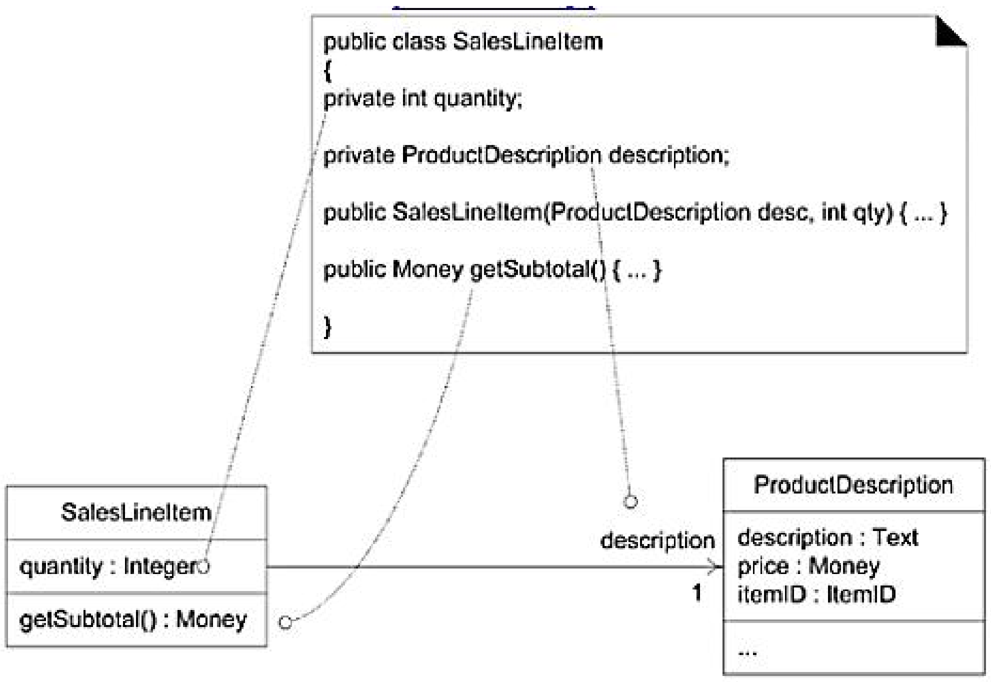
- DCDs depict the class or interface name, super classes, method signatures, and simple attributes of a class.
- Basic class definition in an object-oriented programming language can be created from DCD’s

Defining a Class with Methods and Simple Attributes

- From the DCD, a mapping to the basic attribute definitions (simple Java instance fields) and method signatures for the Java definition of *SalesLineItem*is straightforward, as shown in Figure.

Creating Methods from Interaction Diagrams

- An interaction diagram shows the messages that are sent in response to a method invocation.
- The sequence of these messages translates to a series of statements in the method definition.
- The *enterltem*interaction diagram in Figure illustrates the Java definition of the *enterltem*method.



```
public class SalesLineItem
{
private int quantity;

private ProductDescription description;

public SalesLineItem(ProductDescription desc, int qty) { ... }

public Money getSubtotal() { ... }

}
```

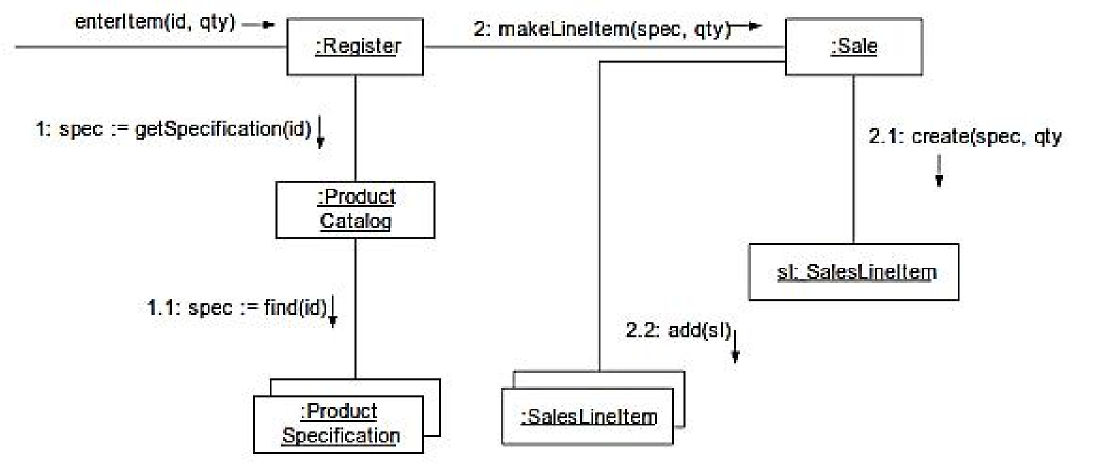


Figure: The enterltem interaction diagram.

- ➢ The *enterltem*message is sent to a *Register* instance; therefore, the *enterltem*method is defined in class *Register.*

  public voidenterltem (ItemIDitemID, intqty)

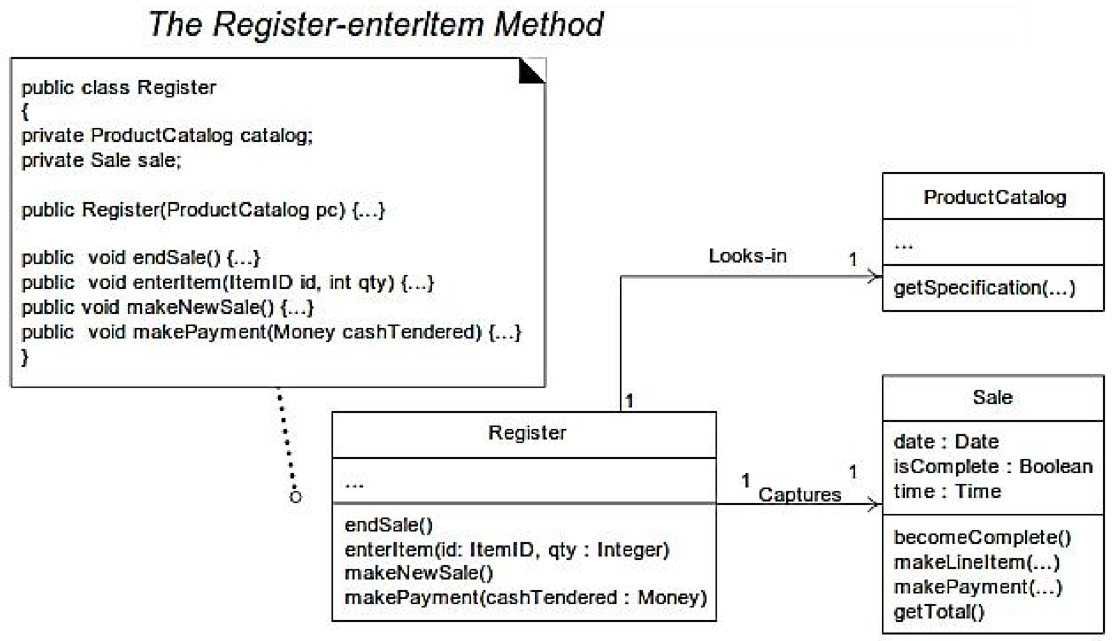
- ✓ **Message 1:** A getSpecification message is sent to the ProductCatalog to retrieve a ProductSpecification.

  ProductSpecification spec = catalog. getSpecification(itemID);

- ✓ **Message 2:** The makeLineltem message is sent to the Sale.

  sale .makeLineltemf spec, qty);

- ➢ Each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method.


The Register-enterItem Method

```
public class Register
{
private ProductCatalog catalog;
private Sale sale;

public Register(ProductCatalog pc) {...}

public  void endSale() {...}
public  void enterItem(ItemID id, int qty) {...}
public void makeNewSale() {...}
public  void makePayment(Money cashTendered) {...}
}
```

Figure: The register class

Container/Collection Classes in Code

- ➢ A Collection class is a container which holds a number of items in a data structure and

- In OO programming languages, these relationships are often implemented withthe introduction of an intermediate container or collection. The one-side classdefines a reference attribute pointing to a container/collection instance, whichcontains instances of the many-side class.
- For example, the Java libraries contain collection classes such as *ArrayList*and*HashMap,* which implement the *List* and *Map* interfaces, respectively. Using*ArrayList*, the *Sale* class can define an attribute that maintains an ordered list *ofSalesLineltem*instances.

Exceptions and Error Handling

- In application development, exception handling should be considered during design work, and certainly during implementation.
- Briefly, in the UML, exceptions are illustrated as asynchronous messages ininteraction diagrams.

Defining the Sale--makeLineltem Method

- The *makeLineltem*method of class *Sale* can also be written by inspecting the *enterltem*collaboration diagram.
- An abridged version of the interaction diagram, with the accompanying Java method, is shown in Figure

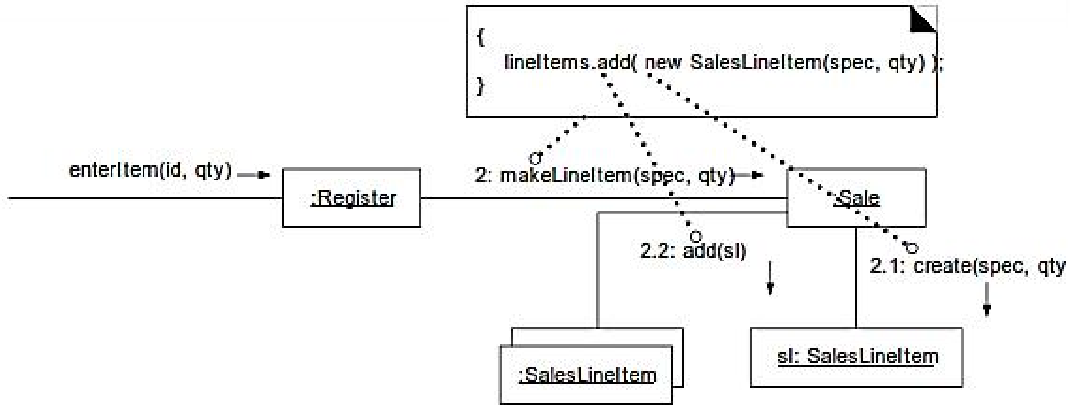


Figure: Sale-makeLineltem method.

Order of Implementation

- Numbers attached to the classes represents order for implementation classes
- Classes need to be implemented from least-coupled to most-coupled
- For example, possible first classes to implement are either Payment or *ProductSpecification*; next are classes only dependent on the prior implementations -*ProductCatalog* or *SalesLineltem*.

Test Driven or Test-First Development

- **Test-first Development (TDD)** is a software development process where the developers first writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards
- In this practice, unit testing code is written *before* the code to be tested, and the developer writes unit testing code for *all* production code. The basic rhythm is to write a little test code, then write a little production code, make it pass the test, then write some more test code, and so forth.

Advantages:

- The unit tests actually get written
- Programmer satisfaction
- Clarification of interface and behavior
- Provable verification
- The confidence to change things

PART-A (2 Marks)

1. **What is GRASP? [May 2013][Nov 2017]**

General Responsibility Assignment Software Patterns (or Principles), abbreviated GRASP, consists of guidelines for assigning responsibility to classes and objects in object oriented design.

2. **What is Responsibility-Driven Design?**

A popular way of thinking about the design of software objects and also larger scale Components are in terms of responsibilities, roles, and collaborations. This is part of a larger approach called responsibility-driven design or RDD.

3. **What is Responsibilities? [May 2019]**

The UML defines a responsibility as “a contract or obligation of a classifier”. Responsibilities are related to the obligations or behavior of an object in terms of its role.

4. **What are the two responsibilities?**

The responsibilities are of the following two types: doing and knowing.

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

5. **Define Pattern.[Nov 2016]**

A pattern is a named and well-known 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, implementations, variations, and so forth.

6. **What are the GRASP patterns?**

Fundamental principles of object design and responsibility assignment are described which is expressed as patterns.

7. **How to Apply the GRASP Patterns?**

The following sections present the first five GRASP patterns:

- Information Expert
- Creator
- High Cohesion
- Low Coupling

Controller

8. Define Creator.

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

9. What is Controller?

The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event.

10. Define Low Coupling. **[Nov 2014] [Nov 2013]**

Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

- low dependency between classes;
- low impact in a class of changes in other classes;
- high reuse potential

11. Define High Cohesion.

High Cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system.

12. What is Information Expert?

Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on. Using the principle of Information Expert a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored. Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it

13. What is singleton pattern?

The singleton pattern is a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

14. What is adapter pattern?

The adapter pattern is a design pattern that translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface.

The adapter is also responsible for transforming data into appropriate forms.

15. What is Observer pattern?

The observer pattern (a subset of the publish/subscribe pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.

16. What is design pattern? [May 2011][Nov 2014] [Nov 2013] (or) When to use patterns? [May 2013] [Nov 2016][Nov 2017][May 2019]

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.

17. State the use of design pattern. [May 2014]

- ✓ Patterns show how to build systems with good OO design qualities.
- ✓ Also provides a shared language that can maximize the value of our communication with other developers.

18. What is bridge pattern?

The **bridge pattern** is a design **pattern** used in software engineering which is meant to "decouple an abstraction from its implementation so that the two can vary independently". The **bridge** uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

19. What is programming and development process?

- ✓ Modern development tools provide an excellent environment to quickly explore alternate approaches, and some (or even lots) design-while-programming is usually worthwhile
- ✓ The creation of code in an object-oriented programming language, such as Java or C# is not part of OOAD; it is an end goal.
- ✓ The artifacts created in the UP Design Model provide some of the information necessary to generate the code.
- ✓ Strength of OOAD and OO programming - when used with the UP -is that they provide an end-to-end roadmap from requirements through to code.

20. Why creativity and changes takes place during implementation?

- ✓ Some decision-making and creative work was accomplished during design work.
- ✓ Generation of the code is a relatively mechanical translation process.
- ✓ The results generated during design are an incomplete first step; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved.
- ✓ The design artifacts will provide a resilient core that scales up with elegance and robustness to meet the new problems encountered during programming. Consequently, expect and plan for change and deviation from the design during programming.

21. How to map design to code?[May 2017][Nov 2015]

Implementation in an object-oriented programming language requires writing source code for:

- ▪ class and interface definitions
- ▪ method definitions

22. How do you create Class Definitions from DCDs ?

- ✓ DCDs depict the class or interface name, super classes, method signatures, and simple attributes of a class.
- ✓ Basic class definition in an object-oriented programming language can be created from DCD's

23. Define Collection class.

A Collection class is a container which holds a number of items in a data structure and provides various operations to manipulate the contents of the collection

24. What is TDD?

Test-first Development (TDD) is a software development process where the developers first writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards

25. What are the advantages of TDD?

- The unit tests actually get written
- Programmer satisfaction
- Clarification of interface and behavior
- Provable verification
- The confidence to change things

26. Define Coupling. [May 2014]

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

27. What is strategy pattern?

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

28. What is observer pattern?

The observer pattern (a subset of the publish/subscribe pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.

29. Define singleton class.

There is only one instance of a class then it is called as singleton class. It is representedwith a number "1" at the right corner of the class box.

PART –B

1. **Explain GRASP Patterns. [Nov 2014][Nov 2013]**

   (or)

   Explain GRASP: designing objects with responsibilities. [May 2014[May 2018][May 2017]

   (or)

   Describe the concept of creator, Low coupling, Controller and high cohesion.[Nov 2016] [May 2013]
2. **Discuss the GRASP creator pattern in detail, including the problem it addresses, the solution and benefits.[Nov 2017]**
3. **Discuss the need for the GoF observer pattern, the solution and its advantages.[Nov 2017]**
4. **Explain the design principles in object modeling. Explain in detail the GRASP method for designing objects with examples.[Nov 2016][**
5. **Write short notes on adapter, factory and observer patterns. [Nov 2014] [May 2014] [Nov 2013] [May 2013] [May 2018]**
6. **Explain with an example the factory method design pattern.[Nov 2017][May 2017][May 2019]**
7. **Explain with an example the creator and information expert GRASP pattern.[Nov 2017]**
8. **Write short notes on strategy and bridge pattern. [8 M]**
9. **Write short notes on applying GOF design patterns. [Nov 2017]**
10. **Compare between different categoriies of design patterns.[May 2018]**
11. **Explain Mapping design to code with NextGen POS Program solution. [16M]**
12. **Expalin in detail about the mapping of design to code implementation in an object oriented language. [Nov 2016]**
13. **Elucidate the operation of Mapping Designs to code. [Nov 2015]**
14. **Compare cohesion and coupling with suitable example. [Nov 2015][May 2019]**
15. **State the role and patterns while developing system design. [Nov 2015]**
16. **Differentiate Bridge and Adapter. [Nov 2015][May 2019]**
17. **How will you design the behavioral pattern? [Nov 2015]**
18. **Explain in detail about controller[May 2019]**