GRASP : Designing Objects with responsibilities

GRASP - General responsibility Assignment software patterns or principles consists of guidelines for assigning responsibility to classes and objects in object oriented design.

* various patterns used in GRASP are:
  - ⇒ controller.
  - ⇒ creator
  - ⇒ Indirection
  - ⇒ Information Expert.
  - ⇒ High cohesion.
  - ⇒ Low coupling.
  - ⇒ polymorphism.
  - ⇒ protected variations
  - ⇒ pure fabrications.

UML Vs Design principles.

* UML is a standard visual modeling language.
* The design tool for software development is created using design principles.

Object design :

* Object designing is a process of planning a system of interacting objects for solving a software problem.

JIT-2106

Scanned by CamScanner

Inputs to Object design:

1. Two way Requirement workshop:

This is an interactive workshop which targets on professionals who are in need to improve their ability to translate customer needs into project requirements.

2. Use cases:

* Each case provide one or more scenarios that contains information, how the system should interact with the users to acheive the goal.
* only 10% to 20% of requirements are analysed in detail

3. programming Experiments.

To identify whether the proposed programming language is suitable for writing the program.

4. Scenarios:

It states how the system should interact with the users to acheive the "business goals".

5. Large Scale logical Architecture:

* It is drawn using UML package diagram.

6. Use case text:

It defines the visible behaviour that software objects should support.

7. System sequence diagram:

* System operation messages are identified by system Sequence diagrams.
* System operation messages are the starting messages in interaction and collaborating objects diagram.

JIT-2106

Scanned by CamScanner

8. Operation contracts:

It is to identify what the software objects must achieve in system operations.

9. Supplementary specification:

* It defines the non-functional goals like internatization that an objects should satisfy

10. Glossary.

* It has details of parameters and other details like,
* Data coming from UI layer.
* Data being passed to database.
* detailed item-specific logic.
* Validation requirements.

11. Domain model:

* Domain model has names and attributes of software domain objects in domain layer of the software architecture.

Activities of object design:

i) With the above inputs, developers.

1. Start coding
2. Start some UML modeling for the object design.
3. Start with another modeling technique such as CRC cards

ii) UML Modeling is acheived by drawing Interaction diagrams and class diagrams

iii) GRASP - General Responsibility Assignment Software patterns and GoF -(Gang for four) design patterns are used for drawing also for coding

JIT-2106

Scanned by CamScanner

iv) OO Design modeling is based on the metaphor of responsibility-driven design (RDD), for assigning responsibilities to collaborate on Objects.

v) The design group of team works in small group for 2-6 hrs with S/w modeling tools, performing different kind of modeling works on creative part of design. Which includes. UI, UO, modeling and database modeling with UML diagrams, prototyping tools, sketches.

Outputs of object design:

The following are the object design outputs

⇒ UML Interaction, class and package diagrams

⇒ UI sketches and prototypes.

⇒ Database models

⇒ Report sketches and prototypes.

JIT-2106

Responsibilities Driven Design (RDD)

* It is design technique in object oriented programming

It involves.

⇒ Describing actions and activities for which the S/w is responsible.

→ Describing the responsibilities in terms of both users and developers.

⇒ Designing S/w objects that implements those responsibilities.

Scanned by CamScanner

Responsibility:

* It is defined as a contract or obligation of a classifier in UML

* In terms of role, object behaviour or obligations of an object is related to responsibilities

* Responsibilities are assigned to classes of objects during Object design.

Types of Responsibilities

⇒ Doing responsibilities
→ knowing responsibilities.

Doing responsibilities of an object include.

1. Doing something like creating an object or doing a calculation.
2. Initiating action in another object.
3. controlling and co-ordinating objects in another objects by activities.

knowing responsibilities of an object include,

1. knowing about private encapsulated data.
2. knowing about related objects.
3. knowing about things it can derive or calculate

example:

Salary → Responsible for calculations like net salary and gross salary

Salary is responsible for knowing basic salary of an employee.

JIT-2106

Scanned by CamScanner

RDD is a metaphor:

* RDD is a general metaphor of OO software design
* RDD leads to view an OO Design as a community of collaborating responsible objects

GRASP – Methodological Approach to basic OO Design.

* The grasp principle or patterns are used for learning the object design and apply design in methodical, rational and in explanations.

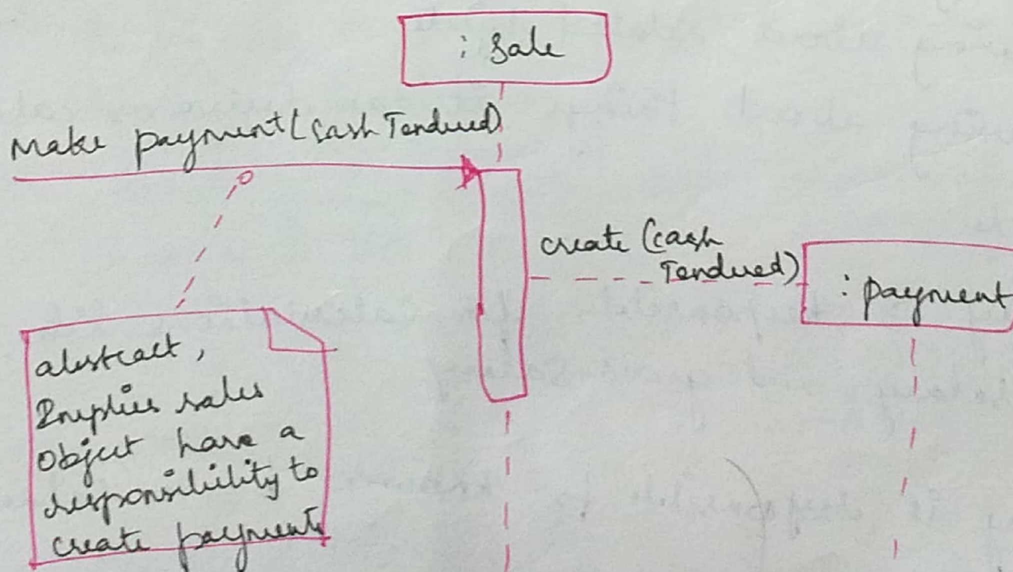
* It is used as a tool to master the OOD and to understand the responsibility assignment in object design.

Connection between Responsibilities, GRASP and UML diagrams

* GRASP → principles that are used to apply in drawing UML Interaction diagrams and during coding.

* Drawing UML Interaction diagrams lead to decide on responsibility assignments.

Example: Relating responsibilities and methods.



JIT-2106

Scanned by CamScanner

Patterns

Definitions:

* A pattern is a named description of a problem and solution that can be applied to new contexts.
* pattern helps to apply its solution varying circumstance and considers the forces and trade offs.

Patterns → given a specific category of problem, It guides the assignments of responsibilities to objects.

Sample pattern:

Pattern Name: Information Expert

Problem: What is the basic principle to assign responsibilities to objects?

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

Good pattern:

* A pattern is said to be a good pattern if it is a problem / solution pair that can be applied in new contexts.
* Also contains information on how to apply it in novel situation and discussion of its trade off, implementation and variations.

Why patterns are Named? and its advantages.

* patterns are named so that the solutions can be communicated and discussed.
* pattern names establish a common understanding of a key characteristics of implementation.

eg: Bubble sort, Abstract factory

JIT-2106

Scanned by CamScanner

Gang of Four Design patterns.

GoF Design patterns.

Introduction

* It contains recurring solutions to common problems in software Design. Referred as GoF or Gang of Four.
* The author of the book is Erich Gamma, Richard Helm, Ralph Johnson and Vlissides.
  The book is divided into 2 parts.
  ⇒ Exploring the capabilities and pitfalls of object oriented programming.
  ⇒ The 23 classic software design patterns in second half.

Is GRASP a set of patterns or principles?

GRASP defines 9 basic OO design principles or basic building blocks in design.

⇒ According to GoF, one person's pattern is another person's "primitive building block".

Applying GRASP to object Design - 9 patterns.

1. creator.
2. Information Expert.
3. Low coupling.
4. controller.
5. High cohesion.
6. polymorphism.
7. pure fabrication.
8. Indirection
9. protected variations.

JIT-2106

Scanned by CamScanner

Creator who create obj

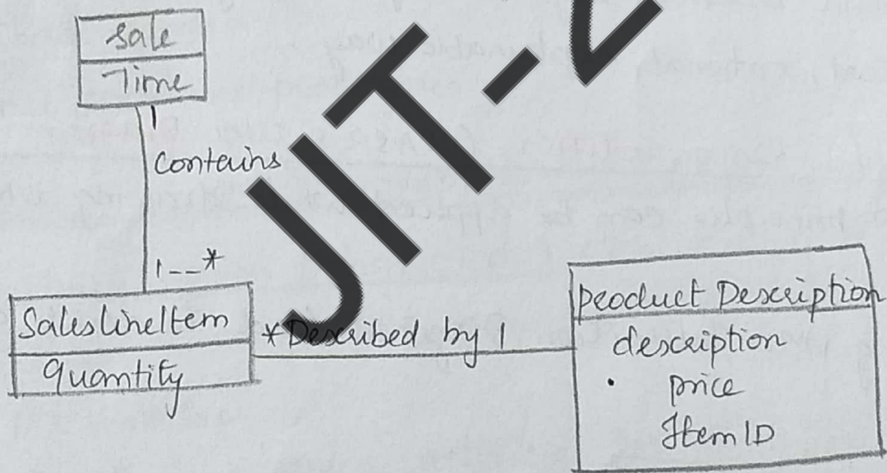
Creation of objects is one of the most common activities in an OO system, which class is responsible for creating objects is a fundamental property of the relationship b/w objects of particular classes

1. problem - Who should be responsible for creating a new instance of some class

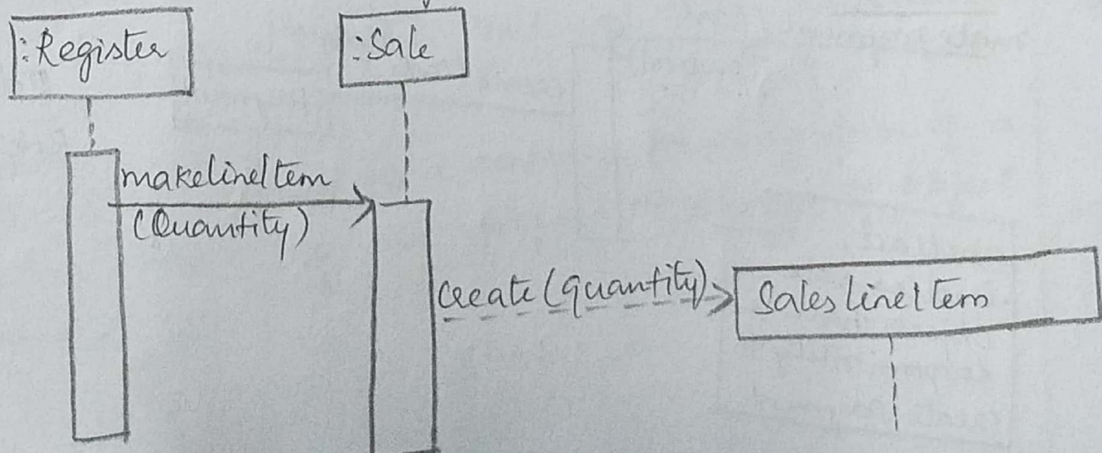
2. Solution - Assign class B the responsibility to create an instance of a class A if one of these is true

- B contains or compositely aggregates A
- B records A
- B closely uses A
- B has the initializing data for A that will be passed to A when it is created
- B is an expert with respect to creating A
- class B which aggregates or contains class A

3. Example - partial Domain model



This leads to the design of object interactions as shown



JIT-2106

Scanned by CamScanner

4. Discussion

* The creator pattern is to find a creator that needs to be connected to the created Object in any event

* Composite aggregate part, container contains content & recorder records.

* Recorded are all very common relationships b/n classes in a Class Diagram

5. Contradictions

* Creation requires significant complexity. Using recycled instances for performance, conditionally creating an instance from one of a family of ||ar classes based upon some external property value

* To delegate creation to a helper class called a Concrete factory or an Abstract factory rather than class suggested by creator.

6. Benefits

Low coupling is supported, which implies lower maintenance dependencies & higher opportunities for reuse

7. Related Patterns or principles

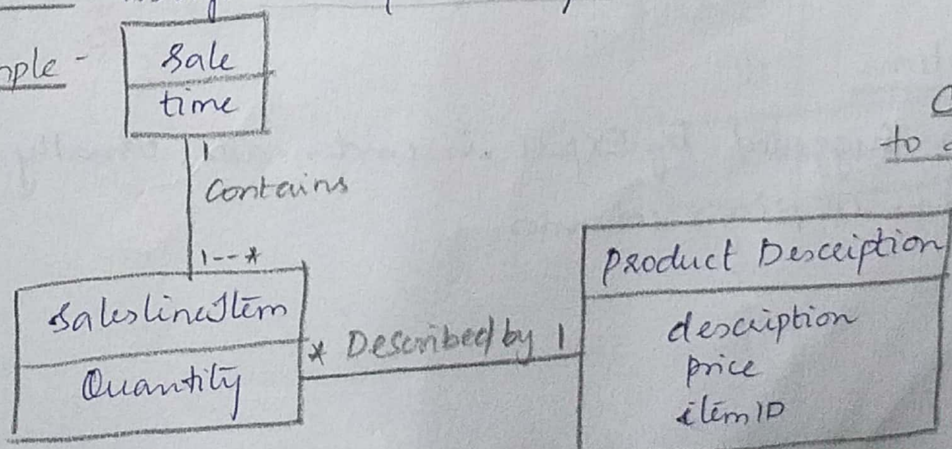
* low coupling
* Concrete factory & Abstract Factory

JIT-2106

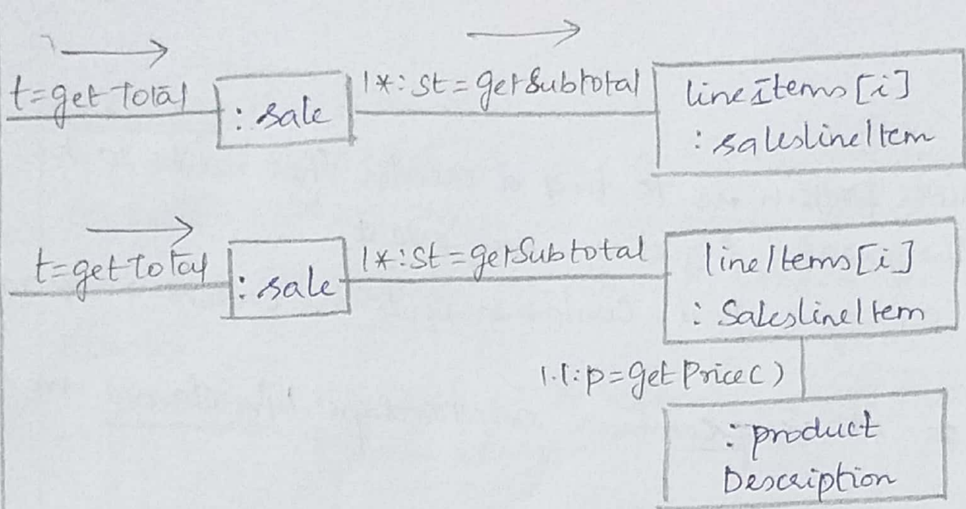
INFORMATION EXPERT obj. info subtotal

Information Expert is a principle used to determine where to delegate responsibilities.

1. Problem - What is a general principle of assigning responsibilities to objects
2. Solution - Assign a responsibility to the information expert -
3. Example -



Scanned by CamScanner



product Description class depicting Information Expert

To fulfill the responsibility of knowing & answering the sale's total, Assign responsibilities to design classes of objects as shown below.

Design class	Responsibility
Sale	Knows sale total
SalesLineItem	Knows lineItem Subtotal
Product Description	Knows product price

JIT-2106

4. Discussion

* It is a basic guiding principle used continuously in object Design
* Fulfillment of a responsibility often requires information i.e., spread across different classes of objects
* Expert usually leads to designs where a s/w object does those operations that are normally done to the inanimate real-world thing it represents.
* The Information Expert Pattern like many things in the object technology has a real world analogy.

5. Contradications

A Soln. suggested by Expert is undesirable, usually because of problems in coupling & cohesion

Scanned by CamScanner

6. Benefits

* low coupling, leads to more robust & maintainable systems
* Behaviour is distributed across the classes that have the required information, thus encouraging more cohesive "light weight" class defn. that are easier to understand & maintain

7. Related patterns or principles

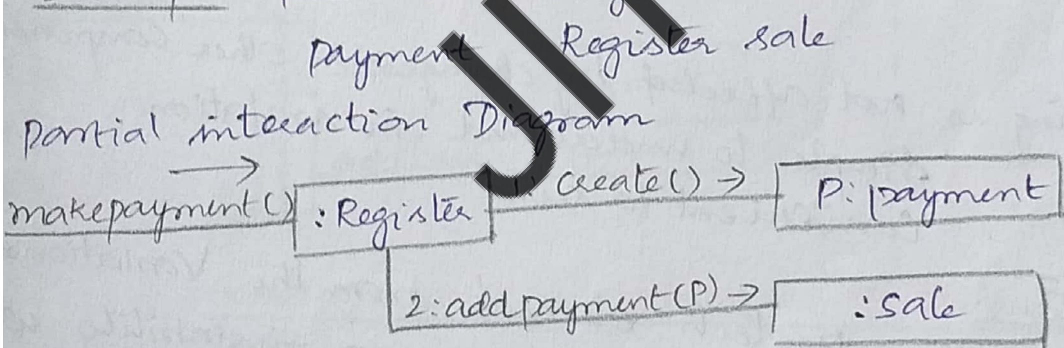
* low coupling
* High Cohesion

Low Coupling which dictates how to assign responsibilities to support

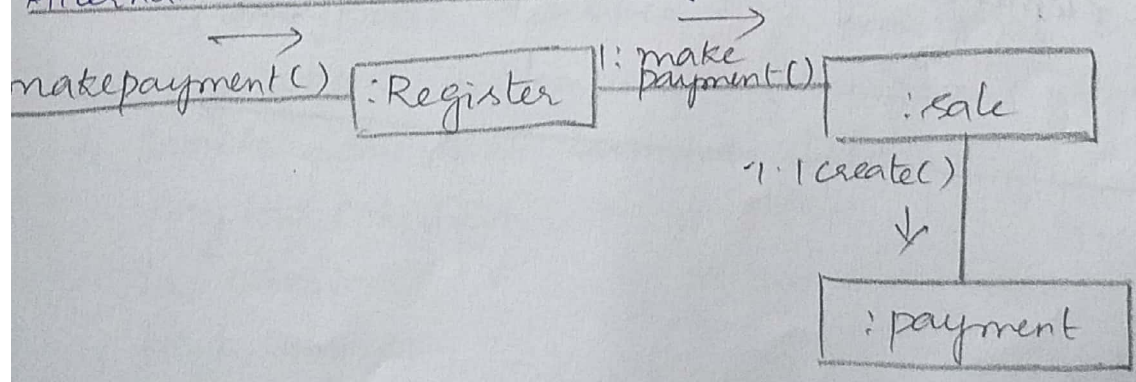
i) low dependency b/n classes
ii) low impact in a class of changes in other classes
iii) High reuse potential

1. Problem - How to support low dependency, low change impact & increased reuse?
2. Solution - Assign a responsibility so that coupling remains low.
3. Example - Partial class Diagram
   Payment Register sale

Partial interaction Diagram



Alternative soln. to create payment & Associate it with the sale



JIT-2106

Scanned by CamScanner

4. Discussion

Low coupling is a principle to keep in mind during all design decisions

Common forms of coupling from TypeX to TypeY in OO languages such as C++, Java & C#

i) TypeX has an attribute that refers to a TypeY instance of TypeY itself

ii) TypeX object calls on services of a TypeY object

iii) TypeX has a method references an instance of TypeY parameter or local variable of type TypeY, or the object returned from a msg. being an instance of TypeY.

iv) TypeX is a direct or indirect subclass of TypeY

v) TypeY is an i/f & TypeX impl. that i/f

5. Contradications

High coupling to stable elements & to pervasive elements is seldom a problem.

6. Benefits

low coupling is not affected by changes in other components
simple to understand in isolation
convenient to reuse

7. Related patterns

Protected variation: protects elements from the variations on other elements by wrapping the focus of instability with an i/f.

JIT-2106

Scanned by CamScanner

High cohesion:

High cohesion is an evaluating pattern that attempts to keep objects appropriately focused, manageable and understandable.

Problem:

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

Solution:

Assign a responsibility so that cohesion remains high.

cohesion:

cohesion is a measure of how strongly related and focused the responsibilities of an elements are.

Breaking problems and programs into classes and subsystems are example of activities that increase the cohesive properties of a system.

A low cohesion class has following problems.

i) Hard to comprehend.

ii) Hard to reuse.

iii) Hard to maintain.

iv) Delicate, constantly affected by change.

JIT-2106

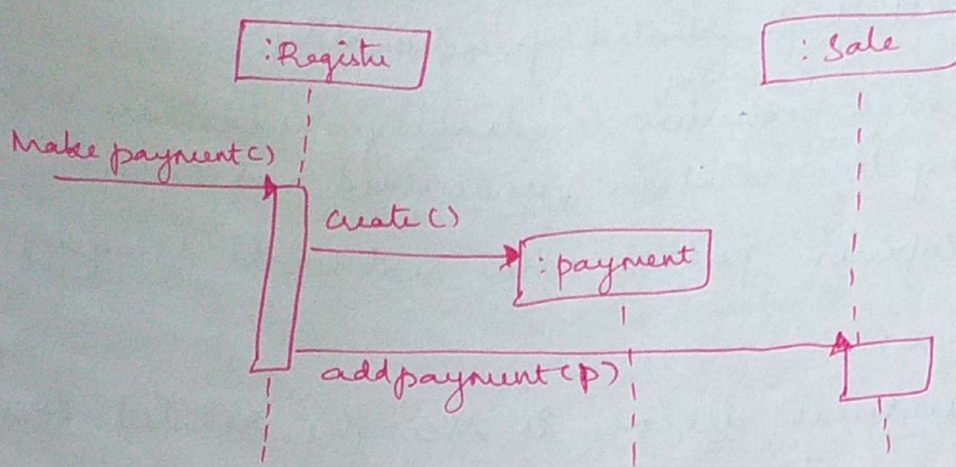
Example:

* If payment instance is to be created and associated with sale: which class is responsible? Register records a payment in the real world domain.

* The creator pattern suggests Register as a candidate for creating the payment.

the Register instance can send an addpayment message to the sale, passing new payment as parameter

Scanned by CamScanner



Sending addpayment message to sale class.

Discussion:

1. High cohesion is a principle to remember during all design decisions.
2. It is an evaluative principle that a designer applies while evaluating all design decisions.
3. GradyBooch describes high functional cohesion as existing when the elements of a component all work together to provide some well bounded behaviour.
4. Scenarios illustrate various degrees of functional cohesion.

→ Very low cohesion: A class is responsible for many things in different functional areas.

→ Low cohesion: A class that is responsible for a complex task in one functional area.

→ High cohesion: A class that has moderate responsibilities in one functional area and collaborate with other classes to fulfill tasks.

→ 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.

JIT-2106

Scanned by CamScanner

5. A class with high cohesion has a relatively small number of methods with high related functionality

6. A class with high cohesion is advantageous because it is relatively easy to maintain, understand and reuse.

7. The high cohesion pattern has real world analogy.

Moderate Design:

Moderate modular design is strongly related to coupling and cohesion.

Modularity: It is a property of a system that has been decomposed into a set of cohesive and loosely coupled modules. It is achieved by creating methods and classes with high cohesion.

* It is achieved by object level by designing each method with a clear single purpose by grouping a related set of concerns into a class.

Contraindications:

i) Case 1: grouping of responsibilities or code into one class or component to simplify maintenance by one person

eg: embedded SQL stmts. Group SQL stmts into one class, RDB operations.

ii) case 2: It is desirable to create less and larger less cohesive server objects to provide interface for many objects.

Benefits: Ease of comprehension of design is increased.

* maintenance and enhancements are simplified.
* low coupling is supported
* Reuse is fine grained; functionality is increased.

JIT-2106

Scanned by CamScanner

Controller:

* A controller is the first object beyond the UI layer responsible for receiving or handling the system operation message.
* The controller pattern assigns the responsibility of dealing with systems events to a non-UI class that represents the overall system or a use case scenario

Problem:

Which 1st object beyond the UI layer receives and co-ordinates a system operation?

Solution:

Assign the responsibility to a class

i) Representing the overall system, root object, device that the software is running within or a major subsystem

ii) Represents a use case scenario within which the system event occurs, named <use case Name> Handler.

<use case Name> Coordinator or

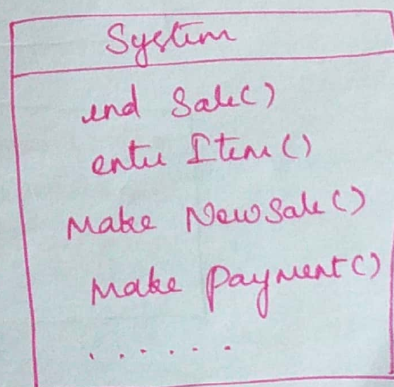
<use case Name> Session.

a) Use the same controller class for all system events in the same use case scenario.

b) A session is an instance of a conversation with an actor

Example:

Some system operations of NextGen pos Application.

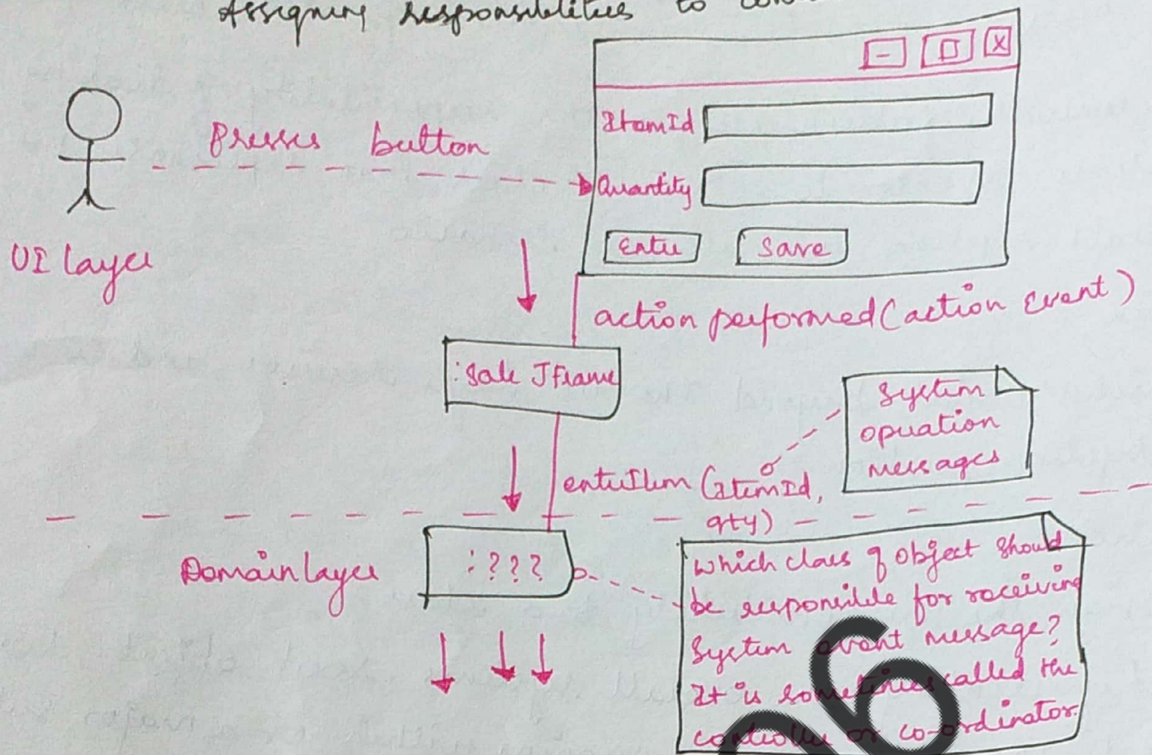


JIT-2106

Scanned by CamScanner

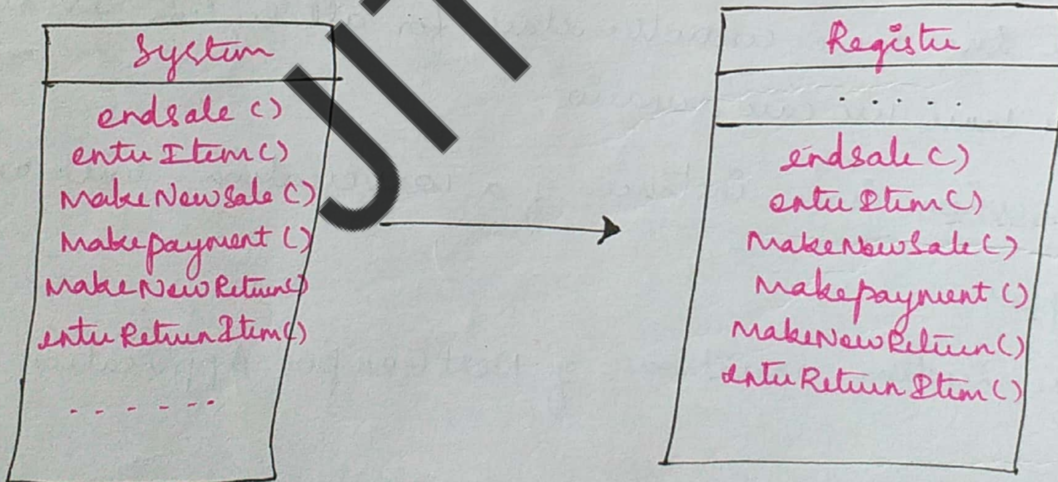
During design, a controller class is assigned the responsibility for system operations

Assigning responsibilities to controller class



Controller classes.

During design, the system operation identified during system behaviour analysis are assigned to one or more controller classes such as register



System operations discovered during system behaviour analysis

allocation of system operation during design using one facade controller.

JIT-2106

Scanned by CamScanner

Discussion:

i) UI layer does not contain application logic rather UI layer Objects must delegate work requests to another layer.

ii) System receive external input events, that involves GUI operated by a person. Other inputs are external messages For. eg. In a call processing telecommunications switch or signals from sensors such as in process control systems.

iii) A common defect in the design of controllers results from over assignments of responsibility

iv) Facade controllers are choosen, when there are not too many system events (or) when the UI cannot redirect system event messages to alternating controllers.

When to choose Use case controller?

Use case controller can be used when there are many system events across different purpose.

v) Boundary, control and Entity classes in UP and Jacobsons older objectory method.

→ Boundary objects are abstractions of the interfaces.

→ Entity objects are the application independent domain software objects.

⇒ control objects are use case handlers as in controller pattern.

Benefits:

* Increased potential for reuse and pluggable interfaces

* An interface as controller design reduces the opportunity to reuse logic in future applications.

* But delegating a system operation responsibility to a controller supports the reuse of the logic in future applications

JIT-2106

Scanned by CamScanner

* opportunity to reason about the state of the use case: controller is a reasonable choice to grasp the current state of activity and operation within the usecase.

Implementations:

Java technologies are used for implementation for the two reasons.

→ Rich client in Java swing.

→ Web UI with Structs on the server.

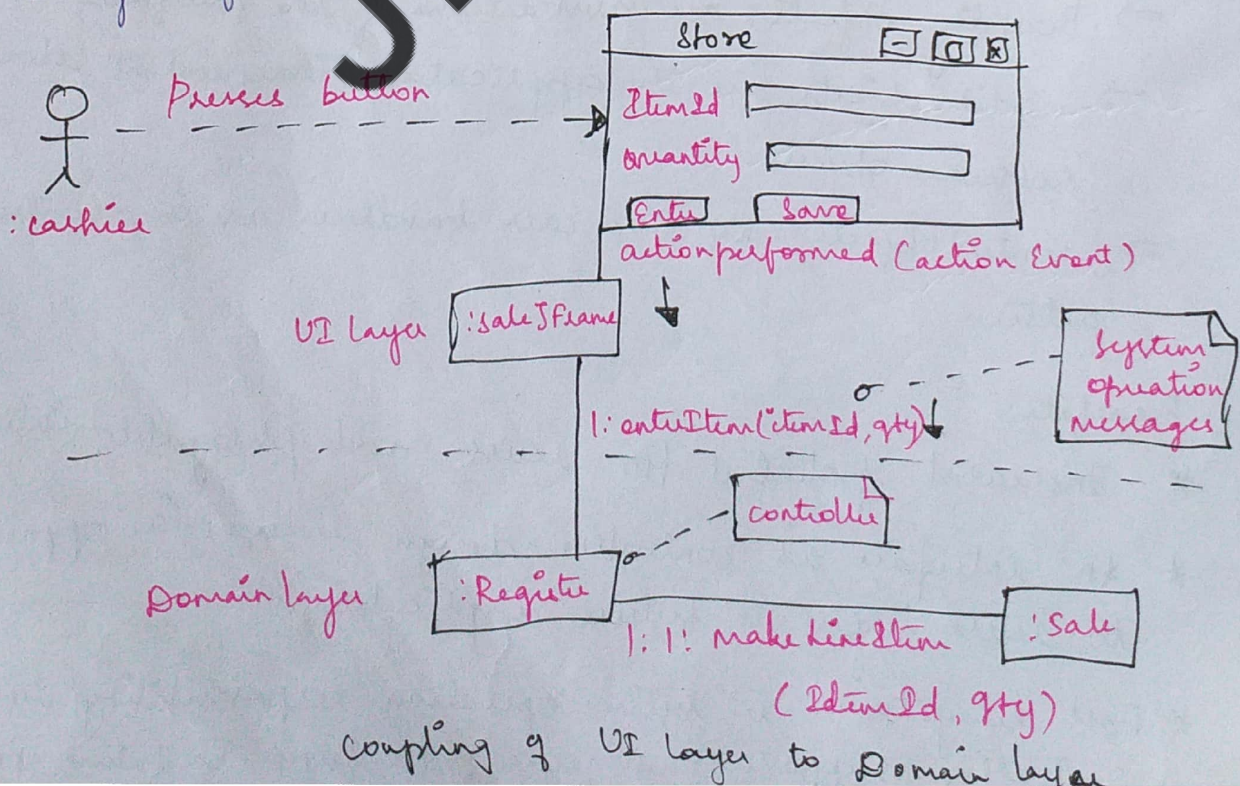
Controller of the controller pattern:

* UI objects and UI layer should not have responsibilities for handling system events

* Reuse of business logic vanishes, if UI layer object handles as system operation that represents of business process.

Message Handling systems and the command pattern:

Applications that resembles message handling systems or servers that receive requests from other processes, the design of the interface and controller is different.



coupling of UI layer to Domain layer

JIT-2106

Scanned by CamScanner

Related patterns:

Command - In a message handling system, each message may be represented and handled by a separate command object.

Facade - A facade controller is a kind of facade

Layers - Placing domain logic in the domain layer rather than the presentation layer is part of layers.

Pure Fabrication - A use case controller is a kind of pure Fabrication.

Design pattern:

* Design patterns are termed as reusable solution for commonly occuring problems in software design.

Design patterns identifies the participating classes and instances their roles and collaboration and the distribution of responsibilities

Essential elements of a pattern:

Pattern Name: Naming a pattern provides easy way to describe a design problem.

Problem: Problem describes specific design problem,
i) when to apply the pattern
ii) how to represent algorithm as objects.

Solution: It provides elements to design for the above problem, their relationships, responsibilities and collaboration

Consequences: Consequences are results and trade offs of applying the patterns.

* Consequences describe the pattern impact on systems flexibility, extensibility or portability.

JIT-2106

Scanned by CamScanner

Describing Design patterns / Template to design a pattern.

i) Pattern Name and classification: A good name assigned to a pattern becomes a part of design vocabulary.

ii) Intent: a) What does the design pattern do?
b) What is its rationale and Intent?
c) What particular design issue or problem does it address?

iii) Also known as: Other well known names for the pattern

iv) Motivation: A Scenario that depicts a design problem

v) Applicability: a) What are the situations in which design pattern can be applied?
b) What are the examples of poor designs that pattern can address

vi) Structure: A graphical representation of the classes in the pattern using a notation based on Object modeling technique

vii) participants: The classes / objects participating in the design pattern

viii) Collaboration: It indicates how the participants collaborate to carry out the responsibilities.

ix) consequences:
a) How does the pattern support its objectives?
b) What are the trade offs and results of using the pattern?
c) What aspects of system structure does it let you vary independently?

X Implementation: It indicates the pitfalls, hints or technique to implement the pattern.

JIT-2106

Scanned by CamScanner

xi) Sample code: Code fragments illustrate how to implement the pattern in any programming languages.

xii) Known uses: It shows the example of pattern found in the real systems.

xiii) Related patterns: This sections tells the design patterns closely related to other.

How to select A Design pattern:

a) consider how design pattern solve design problems.

b) Scan intent sections.

c) how design patterns interrelate.

d) Study patterns of like purpose.

e) Examine a cause of redesign.

f) What should be the variable in design should be considered.

How to use a design pattern:

Step by step procedure

i) Read the pattern.

ii) Study the structure, participants and collaborations

iii) Set a concrete example of the pattern in the code

iv) Choose names for pattern participants in application context

v) Define the classes.

vi) Define application specific names for operations in patterns.

vii) Implement the operations to carry out the responsibilities and collaborations.

JIT-2106

Scanned by CamScanner

Design patterns are categorized into 5 categories.

⇒ creational pattern
⇒ Structural pattern
⇒ Behavioural pattern
⇒ Model - view - controller pattern.
⇒ Concurrency pattern.

Creational patterns:

* These design patterns provide a way to create objects while hiding the creation logic.

* creational patterns make a system independent of how its objects are created composed and represented.

* It uses inheritance to vary the class that's instantiated and object creational patterns delegate instantiation to another object

Definition of creator class:

* creator class is abstract and generating method that does not have any implementation.

* creator class is a concrete class. the generating method having a default implementation.

* Factory method is just a particular case of the factory design pattern

* In modern programming languages, the factory with registration is more used.

Factory:

* Factory pattern is a design pattern to implement the concept of factories

* It is GoF Abstract factory pattern also called as simple factory or concrete factory

JIT-2106

Scanned by CamScanner

Separation of concerns:

* It is an application of the GRASP high cohesion principle.
* It is a design principle to modularize or separate distinct concerns into different areas so that it has a cohesive purpose.

Advantages of factory objects.

1. Separate the responsibility of complex creation into cohesive helper objects.
2. Hide potentially complex creation logic.
3. Introduction of performance - enhancing memory management strategies such as object caching or recycling.

Factory pattern.

Services factory
accountingAdapter : IAccountingAdapter inventoryAdapter : IInventoryAdapter taxcalculatorAdapter : ITaxcalculatorAdapter
getAccountingAdapter() : IAccountingAdapter getInventoryAdapter() : IInventoryAdapter getTaxcalculatorAdapter() : ITaxcalculatorAdapter ----

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 find
the right class:
// read it from an external property.
String className = System.getproperty("tax
calculator.class.name");
taxCalculatorAdapter = (ITaxcalculatorAdapter)
  class.forName(className).newInstance();
}
return taxCalculatorAdaptor
```

JIT-2106

Scanned by CamScanner

Name: Factory

Problem: who would be responsible for creating objects when there are special considerations such as complex creation logic, a desire to seperate the creation responsibilities for better cohesion.

Solution: A pure fabrication is created called a factory that handles the creation.

Partial Data - Driven Design.

In service factory, the logic to decide which class to create is resolved by reading in the class name from an external source and then dynamically loading the class.

Factory method pattern:

* Factory method pattern is also known as Virtual constructor

eg: library.

A library uses abstract classes for defining and maintaining relation between objects.

Intent:

Defines an interface for creating an object but leaves the choice of its type to the subclasses.

Also known as Virtual constructor.

Motivation: Factory method pattern offers a solution. It encapsulates the knowledge of which document subclass to create and moves this knowledge out of the framework.

Applicability:

Applicable when,

i) A class can't anticipate the class of objects it must create.

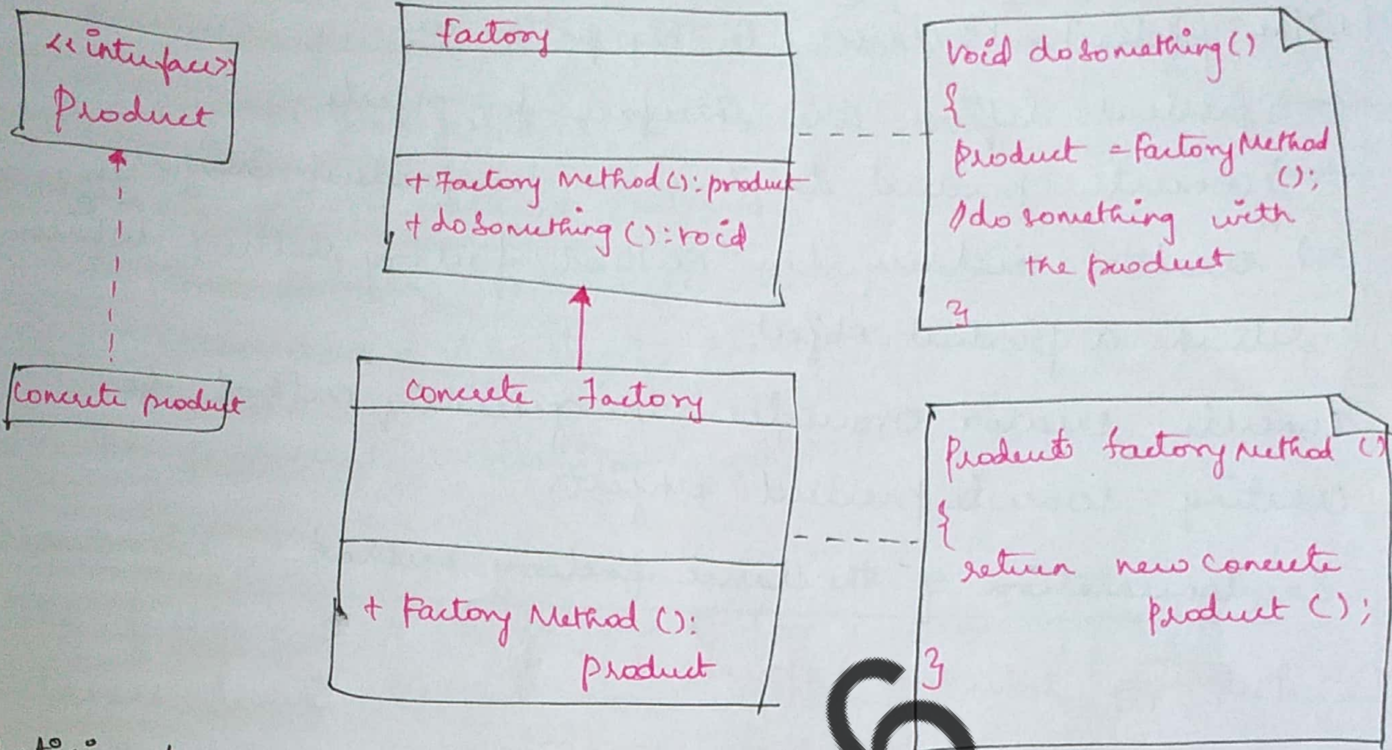
ii) A class wants its subclasses to specify the object its create.

iii) classes delegate responsibility to one of several helper subclasses.

JIT-2106

Scanned by CamScanner

Structure



Participants:

i) Product (Document) - defines the interface of object the factory method creates.

ii) concrete product (My document) - implements the product interface

iii) creator (Application) - declares the factory method, which returns an object of type product.

iv) Concretecreator (my application) - overrides the factory method to return an instance of a concrete product

collaboration: creator relies on its subclasses to define the factory method so that it returns an instance of the concrete product.

consequences:

i) Eliminate the need to blind applications specific classes in the code

ii) Provides hooks for subclasses

iii) connects parallel class hierarchies.

JIT-2106

Scanned by CamScanner

Implementation :

The participants classes in this pattern are.

⇒ product defines the interface for objects.

⇒ concrete product implements the product interface

⇒ creator declares the method factory method, which returns a product object.

concrete creator overrides the generating method for creating concrete product objects.

Implementation of the classic factory method.

```
public interface product { }
public abstract class creator
{ public void anoperation ()
  { product product = factory method ();
  }
protected abstract product factory method ();
}
public class Concrete product implements product ()
public class Concrete creator extends creator
{ protected product factory method ()
  { return new Concrete product ();
  }
} public class client
{ public static void main (string args[])
  { creator creator = new Concrete creator ();
    creator. anOperation ();
  }
}
```

JIT-2106

Scanned by CamScanner

Related patterns:

i) Abstract factory is often implemented with factory method
ii) factory methods are often called within template methods

Benefits of factory method pattern:

1. Factory pattern introduces a separation between the application and a family of classes.

2. It provides customization hooks.

Drawbacks:

The factory has to be used for a family of objects. If the classes doesn't extend common base classes or Interface they cannot be used in a factory design template.

Uses:

1. The factory method is one of the most used and more robust design patterns.
2. Whenever an application is designed, factory plays a vital role in creating objects.
3. factory is used to manipulate objects of same type as abstract objects.

Structural patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures.

Bridge pattern:

Definition: Bridge is used to decouple an abstraction from its implementation. It is a type of structural pattern.

* This pattern decouples implementation class and abstract class by providing a bridge structure between them.

JIT-2106

Scanned by CamScanner

Intent: Decouple an abstraction from its Implementation such that the two can vary independently.

Also known as: Handle / body.

motivation: Inheritance binds its implementation to the abstraction permanently which makes its difficult to modify extend, reuse abstractions and implementation independently.

eg: Bridge pattern addresses the problem by window abstraction and its implementation in separate class hierarchies.

Applicability:

* Bridge pattern is used to avoid a permanent binding between abstraction and implementation.
* Both abstraction and implementation must be extensible by subclassing.
* changes in abstraction and implementation should have no impact of clients.
* The implementation of an abstraction is completely hidden from the clients.
* Implementation can be shared among multiple objects using this pattern.

Participants:

i) Abstraction: defines the abstraction Interface

ii) Refined Abstraction: Extends the Interface identified by Abstraction

iii) Implementor - defines the interface for Implementation

iv) concrete Implementor. Implements the Implementor interface and defines its concrete implementation

JIT-2106

Scanned by CamScanner

Collaboration: Abstraction forward client request to its implementor objects consequences:

i) Decoupling interface and implementation:

* It elliminates compile-time dependencies on the implementation.

* Decoupling encourages layering.

ii) Improved Extensibility: Abstraction and Implementor hierarchies can be extended independently.

iii) Hiding implementation details from client. - protects clients from implementation details

Implementation Issues:

⇒ only one Implementor

→ creating the right implementor object.

⇒ Sharing Implementor.

⇒ using multiple inheritance

participant classes in bridge pattern.

⇒ Abstraction

⇒ Abstraction Impl.

→ Implementor

⇒ Concrete Implementor 1, concreteImplementor 2

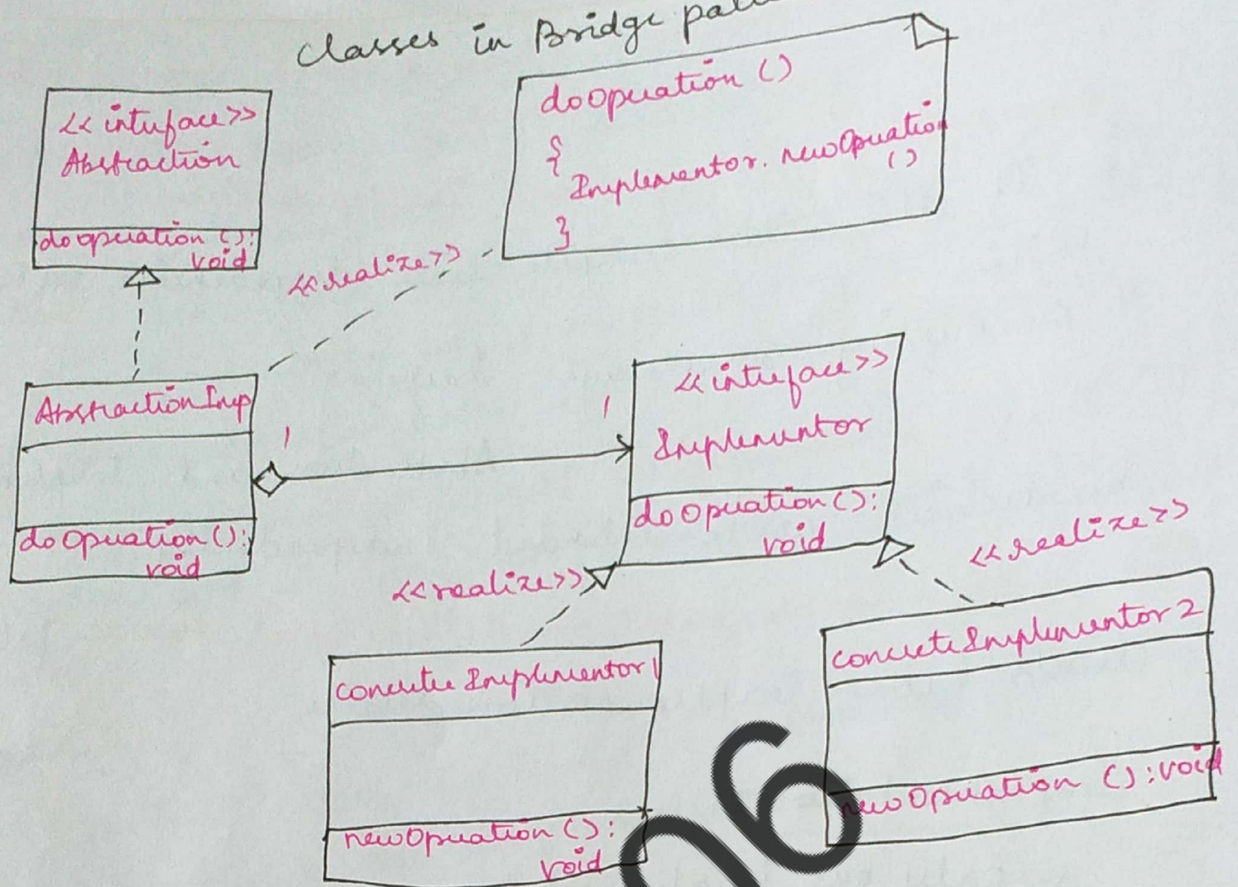
The ability to control the access to an object can be required for variety of reasons.

→ controlling the objects needed to be instantiated and initializing them giving different access rights to an object.

JIT-2106

Scanned by CamScanner

Classes in Bridge pattern.



Graphical user Interface frameworks:

GUI frameworks uses the bridge pattern to seperate abstraction from platform specific implementation.

Related pattern:

Abstract factory pattern: It is used to create and configure particular bridge. eg:. concrete Implementor at runtime.

Uses:

* Decoupling Interface and Implementation.
* Abstraction and Implementation hierarchies can be extended Independently.

JIT-2106

Scanned by CamScanner

same functionality
multiple class → adapter class
4%
2% } → Cust adapter
Norez

Adapter (GOF)

* Adapter pattern is used to convert the programming i/f of one class into that of another.

problem How to resolve incompatible i/f to all or components with different i/f?

Solution Convert the original i/f of a component into another i/f, through an intermediate adapter object

Intent Convert the i/f of a class into another i/f clients expect.

Also known as Wrapper

Motivation To make existing & unrelated classes in an appln. work together with a different & incompatible i/f

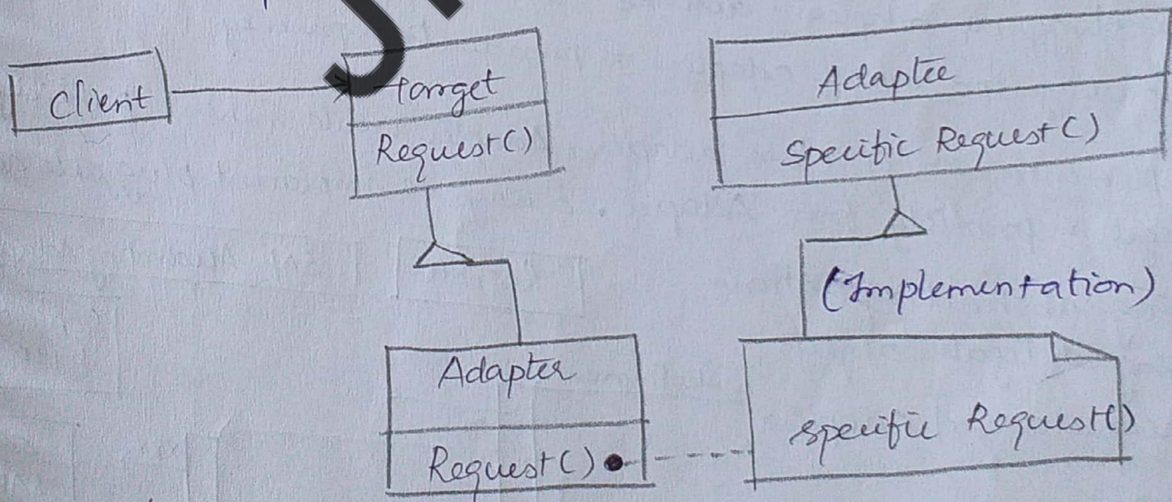
Applicability * An existing class & its i/f doesn't match the one we require

* A reusable class that cooperates with unrelated classes is to be created

* It is impractical to adapt the i/f of existing subclasses

Structure

A class adapter uses multiple inheritance to adapt one i/f to another



participants

Target - domain specific i/f that client uses (Eg. shape)

Client - Collaborates with objects conforming to the Target i/f (Eg. Drawing Editor)

Adaptee - Existing i/f that needs adapting (Eg. Text View)

Adapter - adapts the i/f of Adaptee to the Target i/f (Text shape)

JIT-2106

Scanned by CamScanner

Known uses - ET++ Draw reuse the ET++ classes for text editing by using TextShape Adapter class
- Interviews 2.6 defines an object adapter called GraphicBlock, a subclass of interactor that contains a Graphic instance

Related patterns

- Bridge has a ||lar structure as adapter but with different intent
- Decorator pattern is more transparent to the appln. like an adapter
- proxy defines a representative for another object but don't change its ilf

3. BEHAVIORAL PATTERN

Algo Assign responsibilities b/n obj — behavior b/n classes — Inheritance

Behavioral patterns are concerned with algorithms & the assignment of responsibilities between objects. It use inheritance to distribute behavior b/n classes

class behavior / Alg changes during run time

Strategy pattern is a behavior pattern where a class behavior or its algorithm can be changed at run time.

Intent Define a family of alg., encapsulate each one & make them inter changeable.

Also known as policy

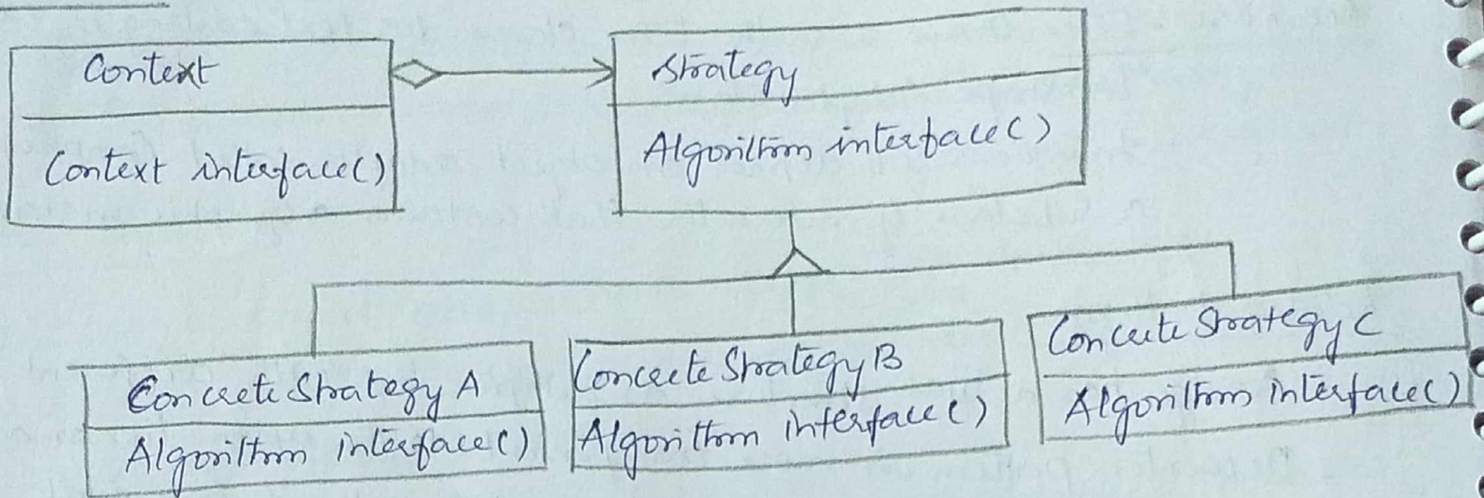
Motivation clients that need linebreaking get more complex if they include the linebreaking code.

Applicability
* Many related classes differ only in their behavior
* Different variants of algorithms required
* An alg. uses data that clients shouldn't about, a class defines many behaviors & they appear as multiple cdn. stmt in its operations

JIT-2106

Scanned by CamScanner

Structure



Participants

- Strategy (compositor) - declares an i/f common to all supported alg
- Concrete Strategy - impl. the alg. using the strategy i/f
- Context - is configured with a ConcreteStrategy object & maintains a reference to a strategy object

Collaborations

A context fwd. requests from its clients to its strategy

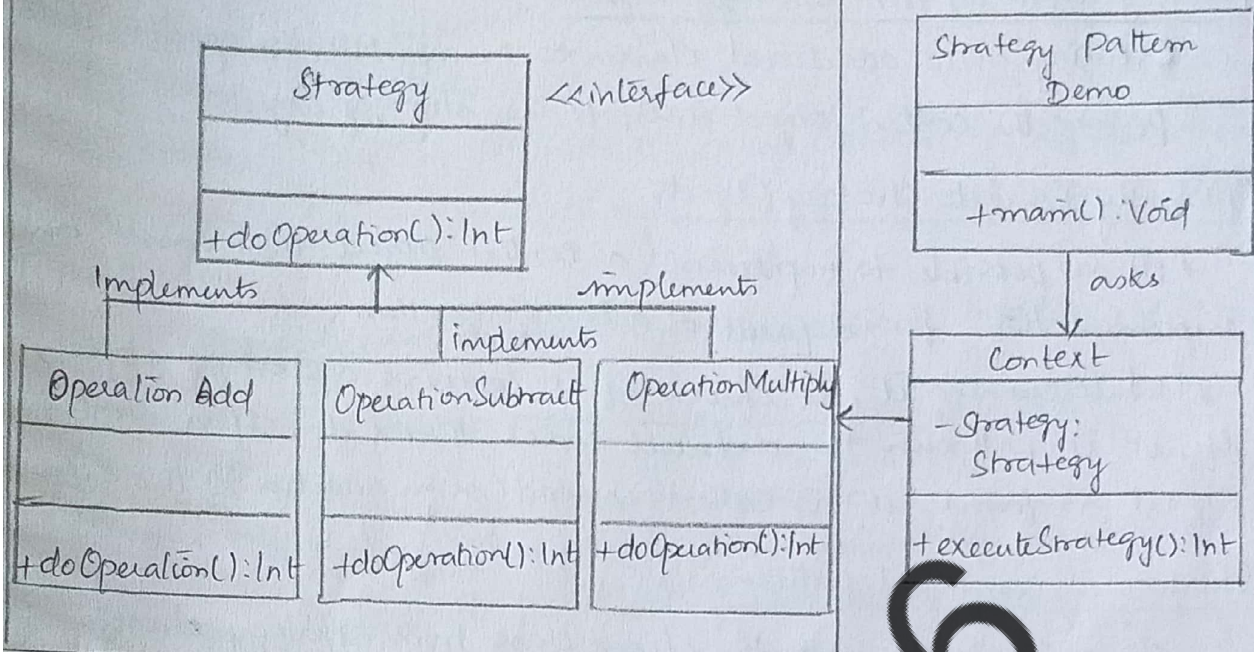
Consequences

Benefits & drawbacks of strategy pattern

i) Families of related algorithms
ii) An alternative to sub classing
iii) Strategies eliminate conditional stmt.
iv) A choice of impl.
v) clients must be aware of different strategies
vi) Comm. overhead b/n strategy & context
vii) Increased no. of objects

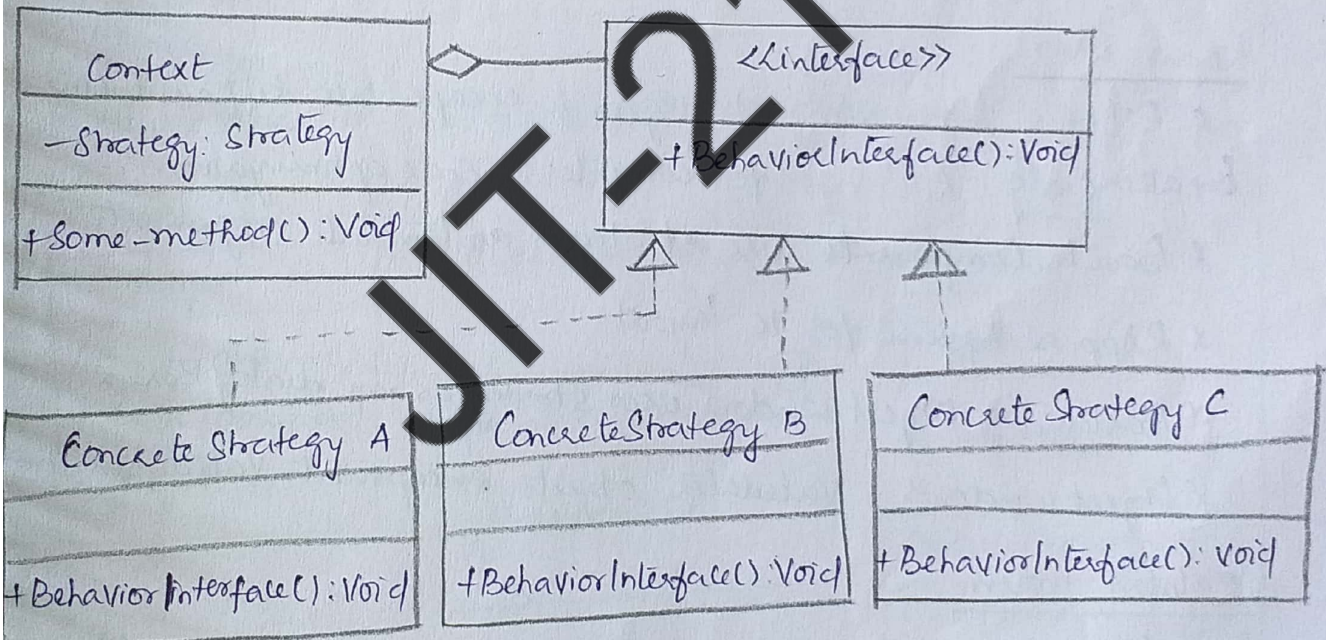
JIT-2106

Scanned by CamScanner



Implementation

Implementation of Concrete Strategy



Strategy - defines an i/f common to all supported alg.

Concrete Strategy - Impl. an alg.

Context contains a ref. to a strategy object, define an i/f that lets strategy accessing its data

JIT-2106

Scanned by CamScanner

Passing data to/from Strategy Object

Creating some additional classes to encapsulate the specific data

. passing the context object itself to the strategy objects

Optionally Concrete Strategy Objects

* It is possible to implement a context object that carries an implementation for default or a basic algorithm.

* While running it, it checks if it contains a strategy object, if not it will run the default code & that's it. If a strategy object is found, it is called instead (or in addition) of the default code.

Strategy & creational patterns

decouple the client class] → factory class inside obj.

In order to decouple the client class from strategy classes is possible to use a factory class inside the context object to create the strategy object to be used.

Known uses

* ET++ & Interviews strategies to encapsulate different line breaking alg. RTL sys. for compiler & code optimization.
* Booch Components use strategies as template arguments
* RApp a system for IC layout.
* Borland's object window uses strategies in dialog Boxes
* Object windows - Validator objects encapsulate Validation strategies

Related pattern

Flyweight pattern

Strategy & Bridge.

related behavior, structural

* Both of the patterns have same UML diagram
* But they differ in intent since the strategy is related with the behavior & bridge is for structure

JIT-2106

Scanned by CamScanner

Observer Pattern

dependents state changes calling some of their methods

* It is a s/w design pattern in which an object called the subject maintains a list of its dependents, called observers & notifies them automatically of any state changes usually by calling one of their methods.
* It is mainly used to impl. distributed event handling systems

Intent Define a one to many dependency b/w objects

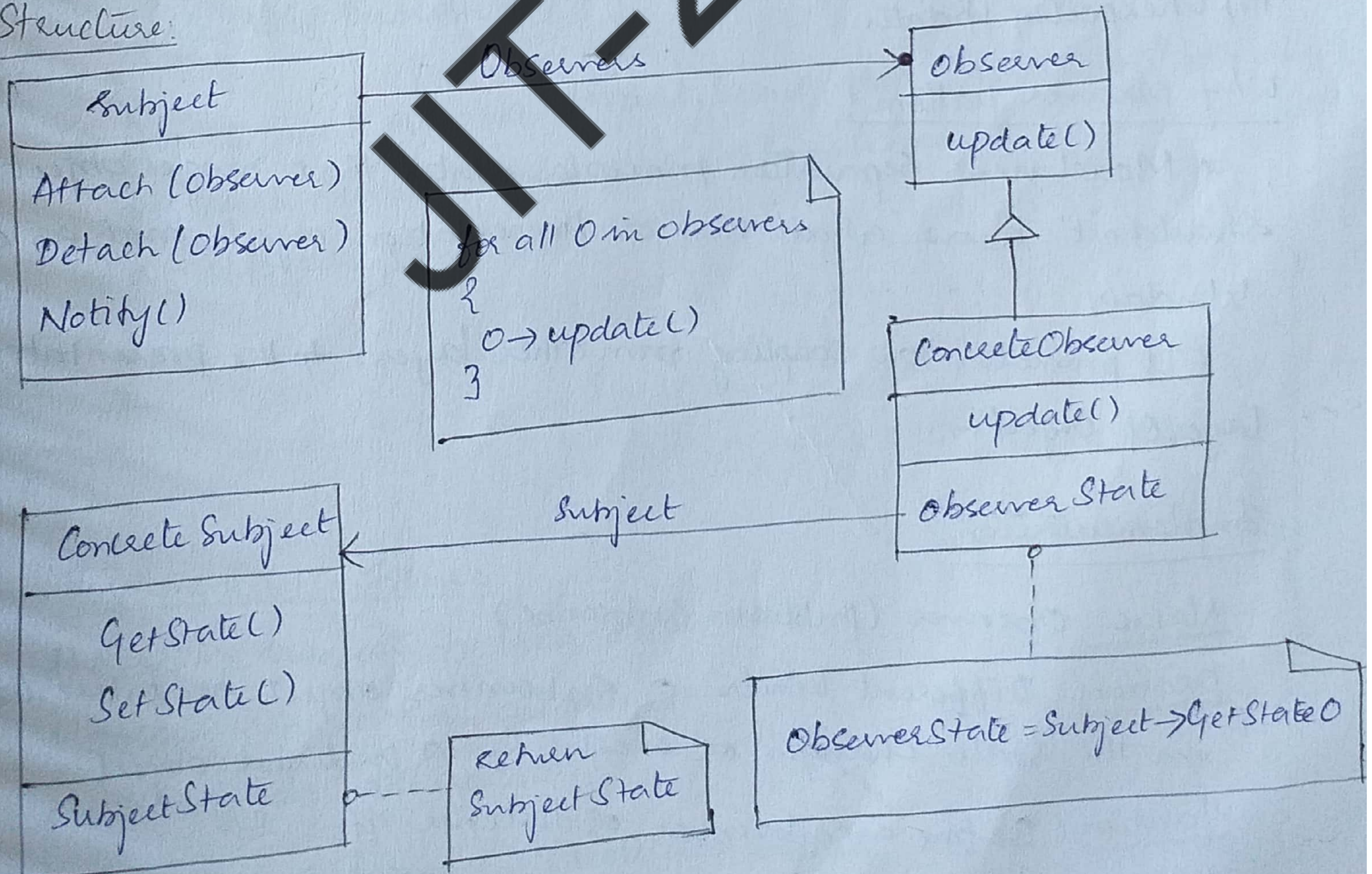
Also known as Dependents, publish-subscribe

Motivation It is difficult to maintain consistency due to partitioning a system into collection of cooperating classes

Publish subscribe All observers are notified whenever the subject undergoes a change in state. Each observer will query the subject to synchronize its state with the subject's state

Applicability
* An abstraction has two aspects one dependent to the other.

Structure:



JIT-2106

Scanned by CamScanner

Participants

* **Subject** — Any no. of Observer Objects may observe a Subject
* **Observer** — defines an updating i/f for objects that should be notified of changes in subject
* **Concrete Subject** — Stores State of interest to ConcreteObserver objects
* **Concrete Observer** — maintains a reference to Concrete Subject object & Stores states that should stay consistent with the Subjects

Collaborations — whenever a change occurs that could make its Observers state inconsistent with its own.

Consequences

Benefits & liabilities of the observer pattern

i) Abstract coupling b/n subject & observer
ii) Support for broadcast communication
iii) Unexpected updates

Why observer pattern?

* Model view separation principle states that model objects should n't know about view or presentation objects such as a window.
* It promotes low coupling from other layers to the presentation (UI) layer of objects

Implementation

Name: Observer (publish-subscribe)
Problem: Different kinds of subscriber objects are interested in the state changes or events of a publisher object.
Solution: Define a subscriber or listener i/f. Subscribers impl. this i/f.

JIT-2106

Scanned by CamScanner

Implementation

Events

In both the Java & C#.NET impl. of Observer, a "event" is communicated via a regular msg, such as OnPropertyEvent.

Eg.

```
class PropertyEvent extends Event
{
  private Object SourceOfEvent;
  private String propertyName;
  private Object oldValue;
  private Object newValue;
  //...
}
//...
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 alarmListeners
   al.onPropertyEvent(evt);
  }
  //...
}
```

Implementation Issues

- Mapping subjects to their observers
- observing more than one subject
- who triggers the update?
- Dangling references to deleted subjects

JIT-2106

Scanned by CamScanner

* Making sure subject state is self-consistent before notification
* Avoiding observer specific update protocols
  push model, pull Model, Specifying modifications of interest explicitly
  Void Subject :: Attach (Observer*, Aspect& interest);
  Void Observer :: Update (Subject*, Aspect& interest);
* Encapsulating complex update semantics

Responsibilities of change Manager

It maps a subject to its observers & provides an i/f to maintain this mapping defines a particular update strategy

Known as Smalltalk Model/View/controller, ET++

Unidraw splits graphical editor objects into View (for observer & subject parts

Related patterns

Mediator

Singleton

JIT-2106

Applying GOF Design

GOF patterns are considered as the foundation of all other patterns.

Design patterns are recurring solutions to s/w design problems

Adapter (GOF)

The Adapter pattern is used to convert the programming i/f of one class into that of another.

Problem : How to resolve incompatible i/f, or provide a stable i/f to similar components with different i/f

Solution : Convert the original i/f of a component into another i/f thru' an intermediate adapter object

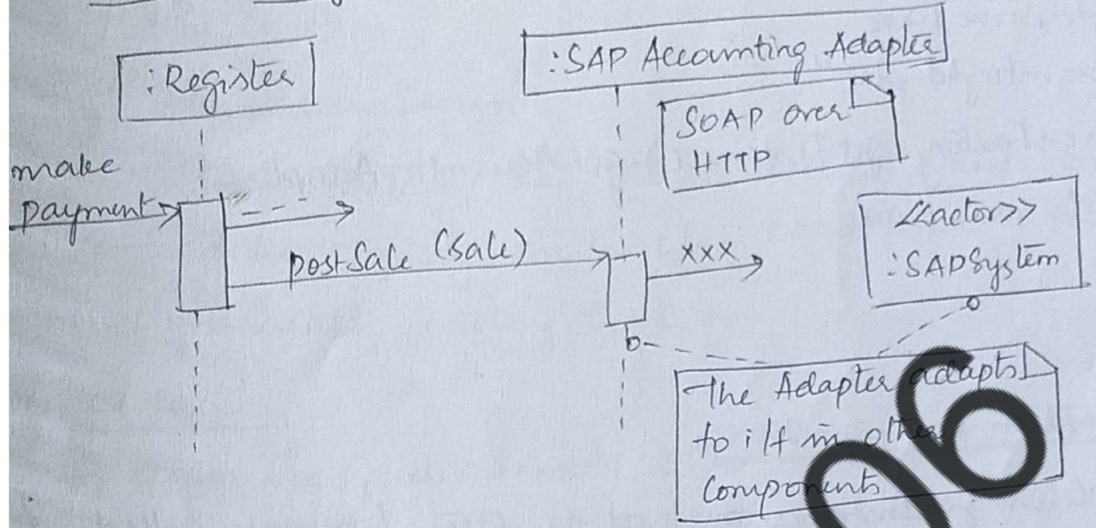
Scanned by CamScanner

Guideline

Include pattern in Type Name

(comm. other reading code / diagram)

Notice that the type names include the pattern name Adapter. This is a relatively common style & has the adv. of easily communicating to others reading the code or diagrams what design patterns are being used.



Related pattern

A resource adapter that hides an external system may also be considered a facade object (another GoF pattern) as it wraps access to the subsystem or system with a single object

Singleton – 1 class / one obj

Singleton pattern is a design pattern used to impl. the mathematical concept of a singleton, by restricting the instantiation of a class to one object.

Solution

One way is to pass the services factory instance around as a parameter to wherever a visibility need is discovered for it.

Name: Singleton

Problem: Exactly one instance of a class is allowed

Solution: Define a static method of the class that returns the singleton

JIT-2106

Scanned by CamScanner

Applying UML

```
Public Class Register
{
 Public void initialize()
 {
 ...do some work...
 accountingAdapter"
 ServicesFactory.getInstance().getAccountingAdapter();
 ...do some work...
 }
}
```

Implementation & design Issues

A Singleton getInstance method is often frequency called. In multi-threaded appln., the creation step of the lazy initialization logic is a critical section requiring thread concurrency ctrl.

```
Public static synchronized ServicesFactory getInstance()
{
 if (instance==null)
 {
 instance=new ServicesFactory();
 }
 return instance;
}
```

Factory

This is design principle to modularize or separate distinct concerns into different areas so that each has a cohesive purpose.

Fundamentally, it is an application of the GRASP High cohesion principle

JIT-2106

Scanned by CamScanner

Name: Factory

Problem: who would 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

Solution: Create a pure fabrication object called a factory that handles the creation

Partial Data-Driven Design:

In ServicesFactory, the logic to decide which class to create is resolved by reading in the class name from an external source & then dynamically loading the class.

Related pattern

Factories are often accessed with the singleton pattern

Observer pattern

* It is a s/w design pattern in which an object called the subject, maintains a list of its dependents, called observers, & notifies them automatically of any state changes usually by calling one of their methods.

* It is mainly used to impl. distributed event handling systems.

Implementation

Events Java & C#. NET impl. The event is more formally defined as a class, & filled with appropriate event data. The event is then passed as a parameter in the event msg.

Eg.

```
class PropertyEvent extends Event
{ private Object SourceOfEvent;
  private String PropertyName;
  private Object oldValue;
  private Object newValue;
  //... } //..
Class Sale
{
  private Void publishPropertyEvent (String name, Object old, Object new)
```

JIT-2106

Scanned by CamScanner

PropertyEvent evt = new PropertyEvent (this, "sale-total", old, new);
for each AlarmListener al in alarmListeners

al. onPropertyEvent (evt);

}

//...

}

Mapping Designs to code

Implementation in an OO language requires writing source code for * Class & i/f Definitions

* Method Definitions

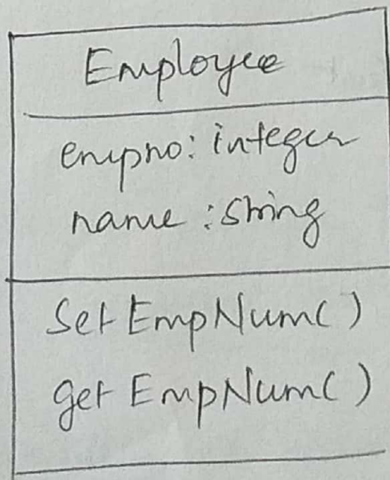
Creating class Definitions from DCD's

* Basic class definitions in OO language can be created from DCD's

* DCD's depict the class or i/f name, superclasses, operation signatures & attributes of a class used for creating class Definition. If was drawn in a UML tool, it can generate the basic class defn. from the diagrams.

Defining a class with method Signatures & Attributes

Employee class

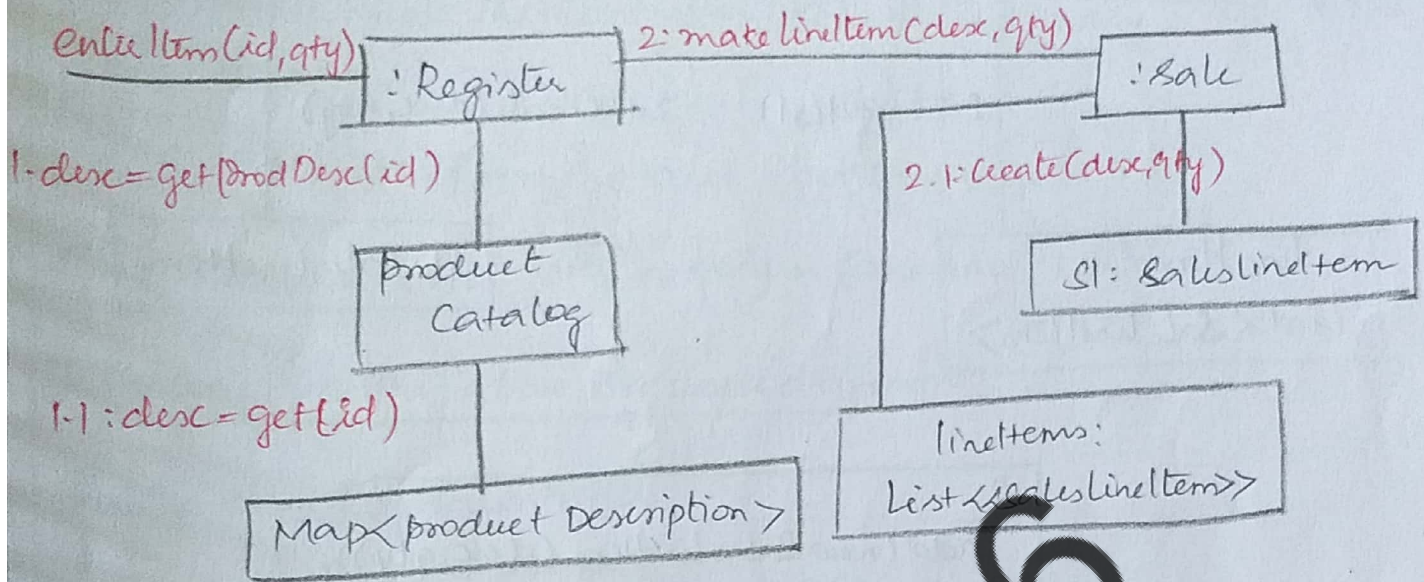


JIT-2106

Scanned by CamScanner

Creating methods from interaction Diagrams

Sequences of msg. in an Interaction Diagram are translated into series of stmt in the method defn.



Collection Classes in code

Collection class is a container which holds a no. of items in a data structure & provides various operations to manipulate the contents of the collection.

Eg - List, Map, etc

Exceptions & Error Handling

Exception handling is a programming language construct to handle the occurrence of Exceptions, special conditions that change the normal flow of pgm. Exe.

The point of exception handling routines is to ensure that the code can handle error cdn's.

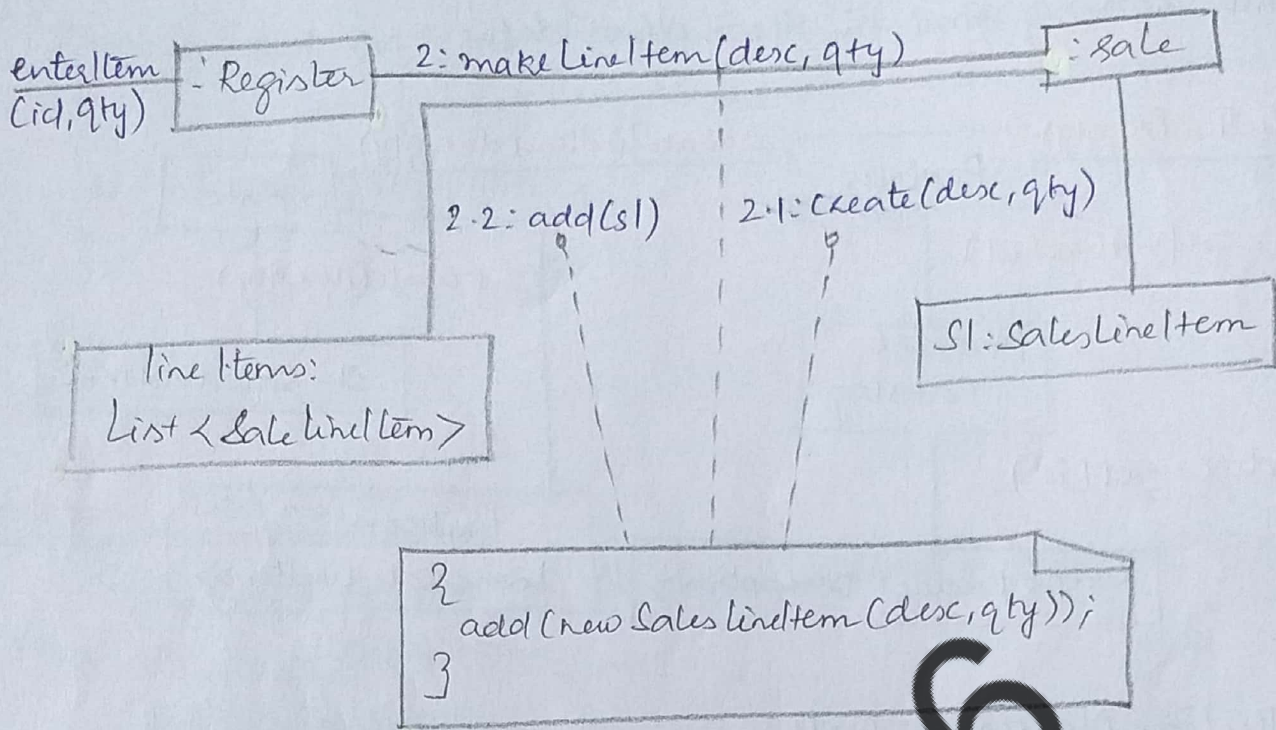
Eg: +getplayer(name: string): player {exception IOException}

Public player getplayer(String name) throws IOException

JIT-2106

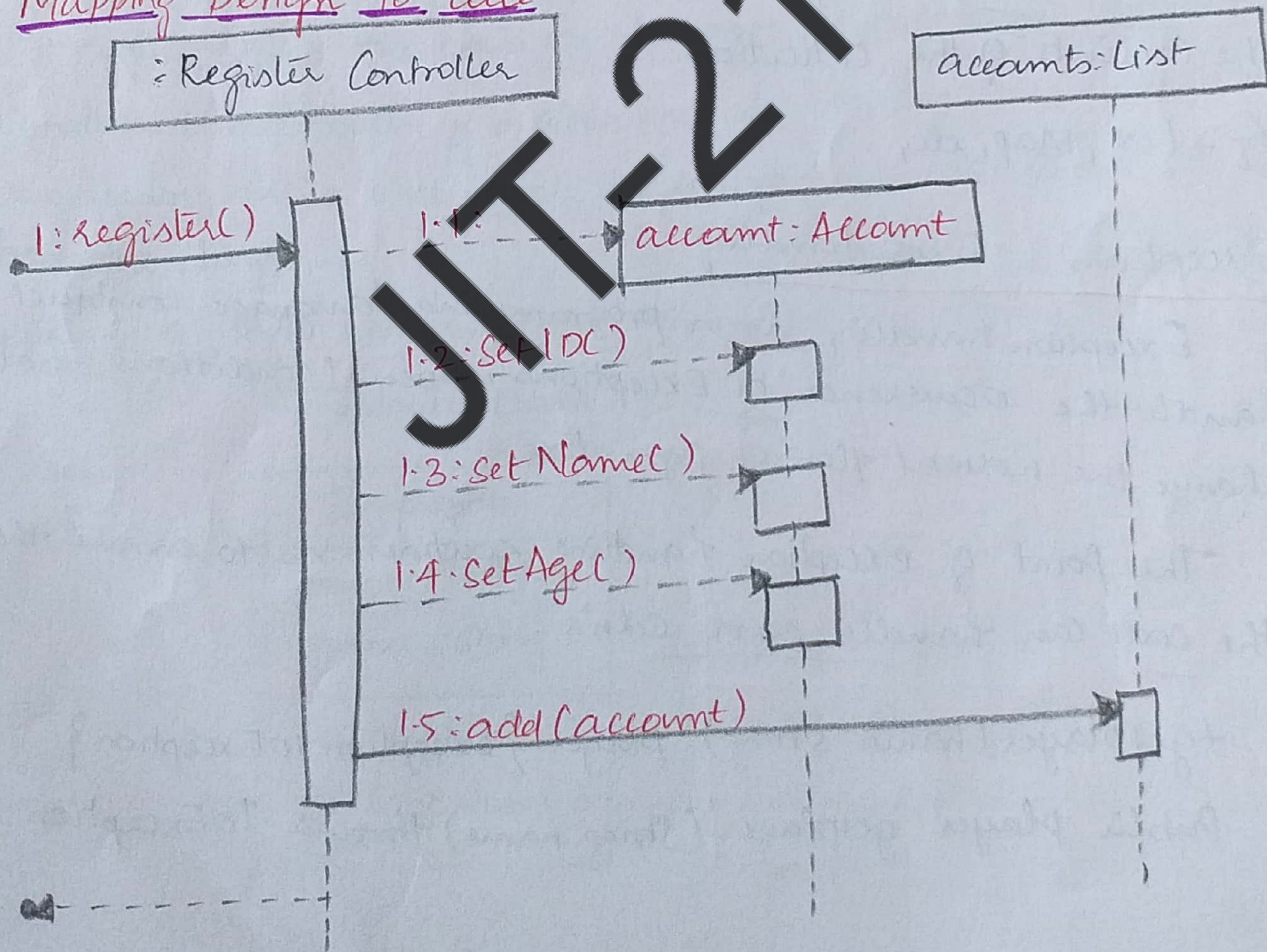
Scanned by CamScanner

Defining the Sale.MakeLineItem Method



Defn. for sale.makeLineItem method thru' collaboration diagram

Mapping Design to code



JIT-2106

For the sequence diagram drawn above, relevant Java code can be written as

Scanned by CamScanner

* A person invokes RegisterController's register method (message: 1)
    - ↳ it creates an acct. obj. (message: 1.1)
    - ↳ After that the Controller sets the id, name & age to the acct. obj. (message: 1.2, 1.3, 1.4) &
    - ↳ Adds itself to the acct. list (message: 1.5)

* The invocation ends with a return (message: 1.6)

<u>Java code for the above sequence diagram</u>

```
Public class RegisterController
{
private List _accounts = new ArrayList();
public void register(String name, int age)
 {
 Account account = new Account();
 account.setID(1);
 account.setName(name);
 account.setAge(Age);
 _accounts.add(account);
 }
public List getAccounts()
{
 return _accounts;
 }
}
```

JIT-2106

Scanned by CamScanner