5. Study of Design Patterns

5.1. Creational Design Patterns

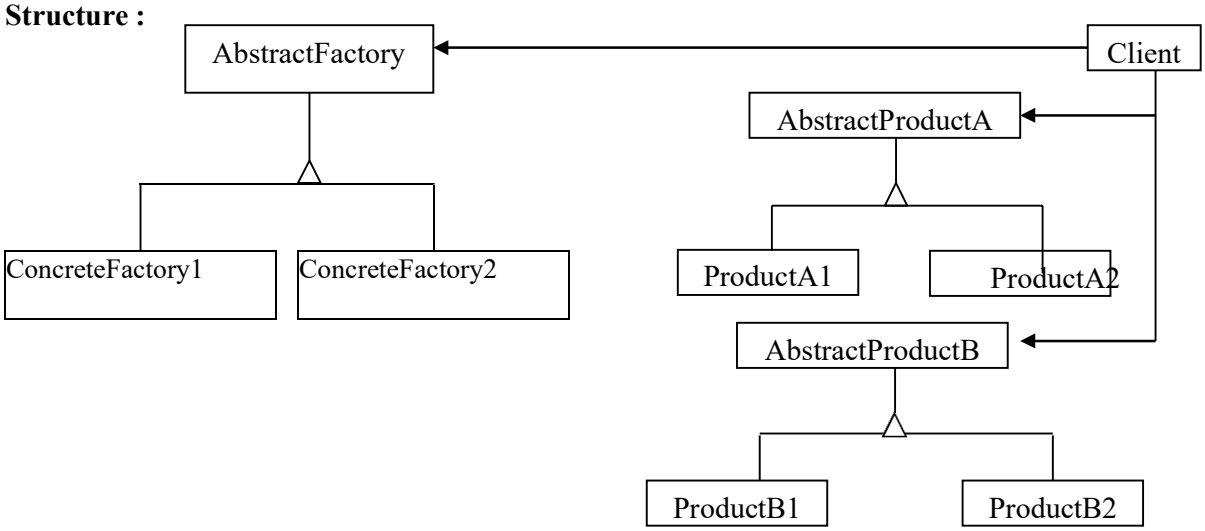
Creational design patterns abstract instantiation process.

5.1.1. Abstract Factory

Scope : Object
Purpose : Creational
Intent : Provide an interface for creating families of related or dependentobjects without specifying their concrete classes.
Also Known As : Kit
Motivation : A user interface toolkit that supports multiple look-and-feelstandards.
Applicability : *Use the Abstract Factory when*

1. A system should be independent of how its product are created,composed, and represented.
2. A system should be configured with one of multiple families ofproducts.
3. A family of related product objects is designed to be used together,and you need to enforce this constraint.
4. You want to provide a class library of products, and you want toreveal just their interfaces, not their implementations.

Structure :



Participants :

1. AbstractFactory : Declares an interface for operations that createabstract product objects.
2. ConcreteFactory : Implements the operations to create concreteproduct objects.
3. AbstractProduct : Declares an interface for a type of productobject.
4. ConcreteProduct : Defines a product object to be created by the corresponding concrete factory. Implements the AbstractProductinterface.
5. Client : Uses only interfaces declared by AbstractFactory andAbstractProduct classes.

Collaborations :

1. Normally a single instance of a ConcreteFactory class is created atrun-time.
2. AbstractFactory defers creation of product objects to itsConcreteFactory subclass.

Consequences :

1. It isolates concrete classes.
2. It makes exchanging product families easy.
3. It promotes consistency among products.
4. Supporting new kinds of products is difficult.

Implementation :

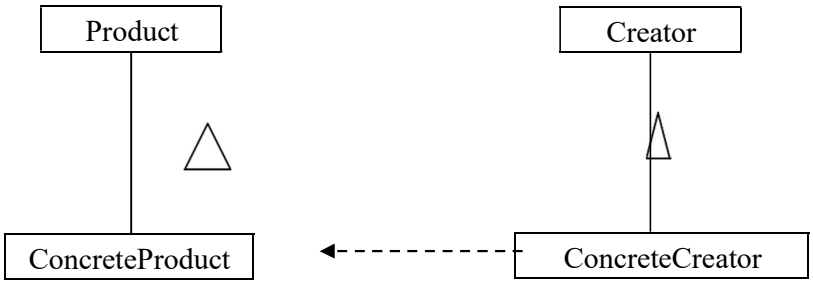
1. Factories as singleton.
2. Creating the product.
3. Defining extensible factories.

5.1.2. Factory Method

Scope : Class.
Purpose : Creational.
Intent : Define an interface for creating an object, but let subclassesdecide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Also Known As : Virtual Constructor.
Motivation : A framework for applications that can present multipledocuments to the user.
Applicability : Use the Factory Method when

1. A class can't anticipate the class of objects it must create.
2. A class wants its subclasses to specify the objects it's created.
3. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass isthe delegate.

Structure :



Participants :

1. Product : Defines the interface of objects the factory methodcreates.
2. ConcreteProduct : Implements the Product interface.
3. Creator : Declares the factory method, which returns an object oftype Product. May call the factory method to create a Product object.
4. ConcreteCreator : Overrides the factory method to return aninstance of a ConcreteProduct.

Collaborations : Creator relies on its subclasses to define the factory method so that it returns as instance of the appropriate ConcreteProduct.
Consequences :

1. Provides hooks for subclasses.
2. Connects parallel class hierarchies.

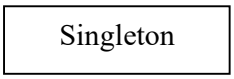
Implementation :

1. Two major varieties.
2. Parameterized factory methods.
3. Language-specific variants and issues.
4. Using template to avoid subclassing.
5. Naming conventions.

Singleton Scope : Object **Purpose :** Creational
Intent : Ensure a class has one instance, and provide a global point ofaccess to it.
Motivation : A digital filter will have one A/D Converter.
Applicability : Use the Singleton Pattern when

4. There must be exactly one instance of a class, and it must beaccessible to clients from a well-known access point.
5. When the sole instance should be extensible by subclassing, andclients should be able to use an extended instance without modifying their code.

Structure :
\



Participants :

Singleton : Defines an Instance operation that lets clients access its unique instance. May be responsible for creating its own unique instance.

Collaborations :

1. Controlled access to sole instance.
2. Reduced name space.
3. Permits refinement of operations and representations.
4. Permits a variable number of instances.
5. More flexible that class operations.

Implementation :

1. Ensuring one instance.
2. Subclassing the Singleton class.

5.2. Structural Design Patterns

Structural Patterns are concerns with how classes and objects arecomposed to form larger structures.

5.2.1. Adapter **Scope :** Class, Object**Purpose :** Structural

Intent : Convert the interface of a class into another clients expects.

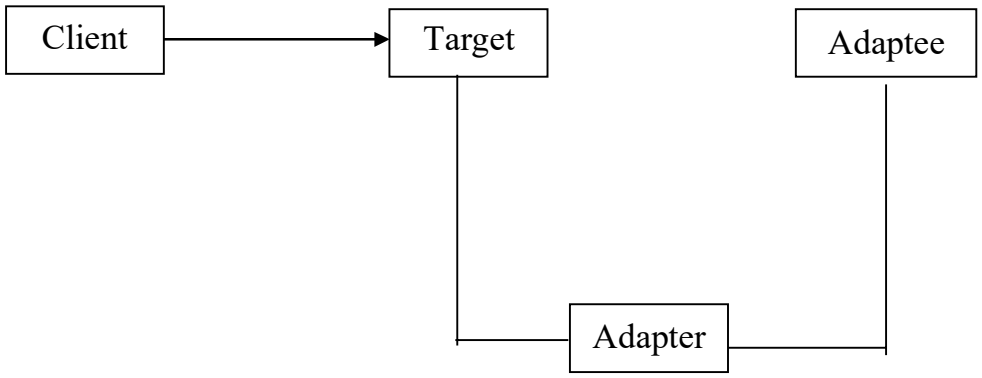
Also Known as : Wrapper.

Motivation : Consider for example a drawing editor that lets user drawand arrange graphical elements into pictures and diagrams.

Applicability : Use the Adapter pattern when

1. You want to use existing class, and its interface does not match theone you need.
2. You want to create a reusable class that cooperates with unrelatedof unforeseen classes.
3. You need to use several existing subclasses, but it's impractical toadapt their interface by sub classing every one.

Structure :



Participants :

1. Target : Defines the domain-specific interface that client uses.
2. Client : Collaborates with objects conforming to the targetinterface.
3. Adaptee : Defines an existing interface that needs adapting.
4. Adapter : Adapts the interface of Adaptee to the Target interface.

Collaborations :

Clients call operations on an Adaptee instance. In turn, the adaptercalls adaptee operations that carry out request.

Consequences :

1. How much adapting does Adapter do?
2. Pluggable adapters.
3. Using two-way adapters to provide transparency.

Implementation :

1. Implementing class adapters in C++.
2. Pluggable adapters : Uses three approaches

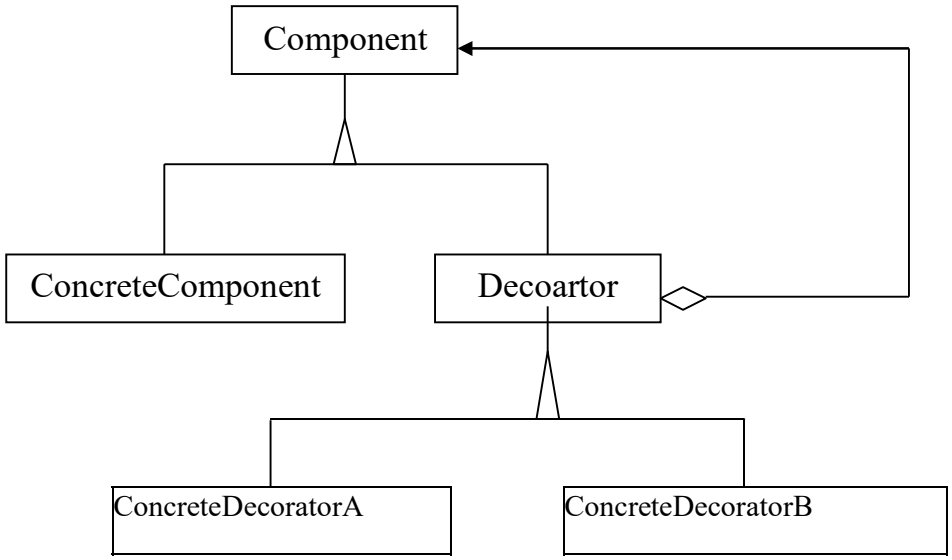
a. Using abstract operations.
b. Using delegate objects.
c. Parameterized adapters.

5.2.2. Decorator Scope : Object Purpose : Structural

Intent : Attach additional responsibilities to an object dynamically.
Also Known As : Wrapper.
Motivation : A graphical user interface toolkit, for example, should letyou add properties like borders or behaviors like scrolling to any user interface component.
Applicability : Use decorator

1. To add responsibilities to individual objects dynamically andtransparently, that is without affecting other objects.
2. For responsibilities that can be withdrawn.
3. When extension by subclassing is impractical.

Structure :



Participants :

1. Components : Defines an interface for objects that can haveresponsibilities added them to dynamically.
2. ConcreteComponent : Defines an object to which additionalresponsibilities can be attached.
3. Decorator : Maintains a reference to a Component object anddefines an interface that conforms to Component's interface.
4. ConcreteDecorator : Adds responsibilities to the component.

Collaborations : Decorator forwards requests to its Component object. Itmay optionally perform additional operations before and after forwardingthe request.
Consequences : The Decorator pattern has at least two key benefits andtwo liabilities

1. More flexibility than static inheritance.
2. Avoids feature-laden classes high up in the hierarchy.
3. A decorator and its component aren't identical.
4. Lots of little objects.

Implementation : Several issues should be considered when applying theDecorator pattern

1. Interface conformance.
2. Omitting the abstract decorator class.
3. Keeping Component class lightweight.
4. Changing the skin of an object versus changing its guts.

5.2.3. Façade Scope : Object Purpose : Structural

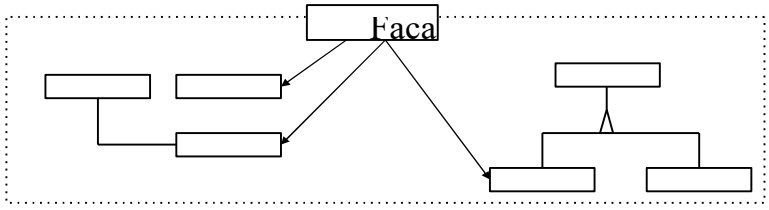
Intent : Provide a unified interface to a set of interface in a subsystem.
Motivation : Consider for example a programming environment thatgives application access to its

compiler subsystem.

Applicability : Use the Façade pattern when

1. You want to provide a simple interface to a complex subsystem.
2. There are many dependencies between clients and theimplementation classes of an abstraction.
3. You want to layer your subsystem.

Structure



Participants :

Façade :

a. Knows which subsystem classes are responsible for arequest.
b. Delegate's client's requests to appropriate subsystemobjects.

2. Subsystem classes :

a. Implement subsystem functionality
b. Handle work assigned by the Façade object.
c. Have no knowledge of the façade.

Collaborations :

1. Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem objects.
2. Clients that use the façade don't have to access its subsystemobjects directly.

Consequences : The Façade pattern offers following benefits

1. It shields clients from subsystem components.
2. It promotes weak coupling between the subsystem and its clients.
3. It doesn't prevent application from using subsystem classes if theyneed to.

Implementation : Consider the following issues when implementing afaçade :

1. Reducing client-subsystem coupling.
2. Public versus private subsystem classes.

5.3. Behavioral Design Patterns

Behavioral Pattern are concerned with algorithms and the assignment ofresponsibility between objects.

5.3.1 Command Scope : Object Purpose : Behavioral

Intent : Encapsulate a request as an object, therby letting you parameterize clients with different requests, queue or log requests, andsupport undoable operations.

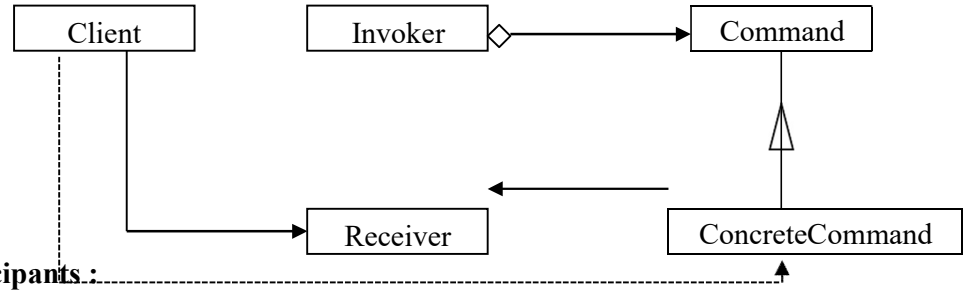
Also Known As : Action, Transition.

Motivation : Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested or thereceiver of the request.

Applicability : Use the Command Pattern when you want to

1. Parameterize objects by an action to perform.
2. Specify, queue, and execute requests at different times.
3. Supports undo.
4. Support logging changes so that they can be reapplied in case of asystem crash.
5. Structure a system around high-level operations built on primitivesoperations.

Structure :



Participants :

1. Command : Declares an interface for executing operations.

a. ConcreteCommand : Defines a binding between a Receiver object and an action.

b. Implements Execute by invoking the correspondingoperations on receiver.

2. Client : Creates a ConcreteCommand object and sets its receiver.
3. Invoker : Asks the command to carry out the request.
4. Receiver : Knows how to perform the operations associated withcarrying out a request. Any class serves as a Receiver.

Collabrations :

The client creates a ConcreteCommand object and specifies itsreceiver.

1. An invoker object stores the ConcreteCommand object.
2. The invoker issues a request by calling Execute on the command.
3. The ConcreteCommand object invokes operations on its receiver tocarry our request.

Consequences : The command pattern has the following consequences :

1. Command decouples the object that invokes the operation from theone that know how to perform it.
2. Commands are first class objects.
3. You can assemble commands into a composite command.
4. It is easy to add new commands.

Implementation : Consider following issues when implementing theCommand Pattern :

1. How intelligent should a command be?
2. Supporting redo and undo.
3. Avoiding error accumulation in the undo process.
4. Using C++ templates.

5.3.2. Iterator **Scope :** Object **Purpose :** Behavioral

Intent : Provide a way to access the elements of an aggregate objectsequentially without exposing its underlying representation.

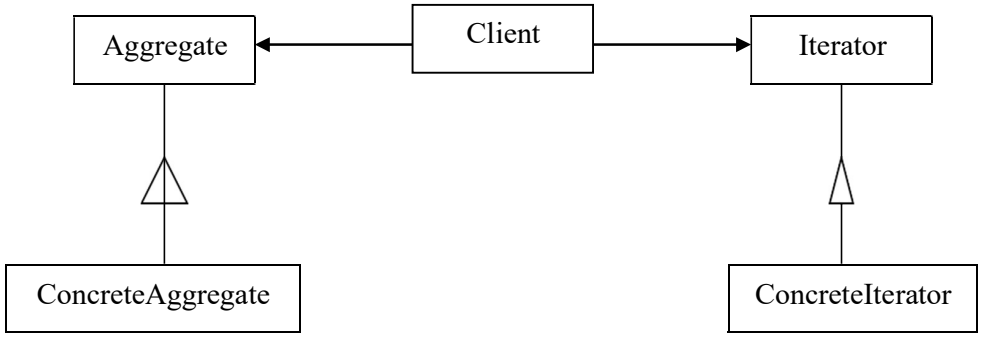
Also Known As : Cursor

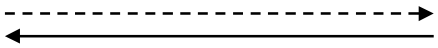
Motivation : An aggregate object such as a list should give you a way toaccess its elements without exposing its internal structure.

Applicability : Use the iterator pattern

1. To access an aggregate object's contents without exposing itsinternal representation.
2. To support multiple traversals of aggregate objects.
3. To provide a uniform interface for traversing different aggregatestructures.

Structure :





Participants :

1. Iterator : Defines an interface for accessing and traversingelements.
2. ConcreteIterator :

a. Implements the Iterator interface.
b. Keep track of the current position in the traversal of theaggregate.

3. Aggregate : Defines an interface for creating an Iterator object.
4. ConcreteAggregate : Implements the Iterator creation to return aninstance of the proper ConcreteIterator.

Collaborations : A ConcreteIterator keeps track of the current object inthe aggregate and can compute the succeeding object in traversal.

Consequences : The Iterator pattern has the three importantconsequences :

1. It supports variation in the traversal of an aggregate.
2. Iterators simplify the Aggregate interface,
3. More than one traversal can be pending on the aggregate.

Implementation : Iterator has many implementation variants andalternatives :

1. Who controls the iteration?
2. Who defines the traversal algorithm?
3. How robust is the iterator?
4. Additional Iterator operations.
5. Using polymorphism iterators in C++.
6. Iterators may have privileged access.
7. Iterators for composites.
8. Null operators.

5.3.3. Observer **Scope :** Object **Purpose :** Behavioral

Intent : Define a one-to-many dependency between objects so that whenone-object changes state, all its dependents are notified and updated automatically.

Also Known As : Dependents, Publish-Subscribe

Motivation : Many graphical user interface toolkits separate thepresentational aspects of the user interface from the underlying application data.

Applicability : Use the Observer pattern in one of the followingsituations :

1. When an abstraction has two aspects, one dependent on the other.
2. When a change to one object requires changing others, and youdon't know how many objects need to be changed.
3. When an object should be able to notify other objects withoutmaking assumptions about which these objects are.

Structure :

Participants :

1. Subject : Knows its observer. Provide an interface for attaching anddetaching Observer objects.
2. Observer : Defines an updating interface for objects that should benotified of changes in subjects.
3. ConcreteSubject : Stores data of interest to a ConcreteObserverobjects.
4. ConcreteObserver : Maintains a reference to a ConcreteSubjectobject. Stores state that should stay

consistent with subject's. Implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations :

1. ConcreteSubject notifies its observers whenever a change occursthat could make its observer's state inconsistent with its own.
2. After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information.

Consequences :

1. Abstract coupling between Subject and Observer.
2. Support for broadcast communication.
3. Unexpected updates.

Implementation :

1. Mapping subjects to their observers.
2. Observing more than one subject.
3. Who triggers the update?
4. Dangling references to deleted subjects.
5. Making sure Subject state is self-consistent before notification.
6. Avoiding observer specific update protocol.
7. Specifying modifications of interest explicitly.
8. Encapsulating complex update semantics.

5.3.4. State

Scope : Object

Purpose : Behavioral

Intent : Allow an object to alter its behavior when its internal statechanges. The object will appear to change its class.

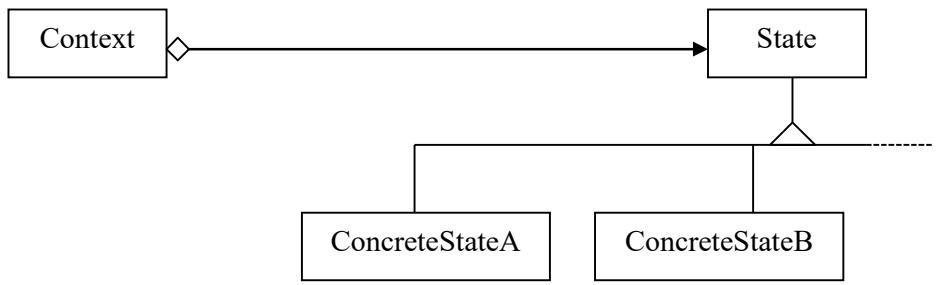
Also Known As : Objects for states.

Motivation : A TCPConnection object can be in one of several differentstates : Established, Listening, Closed.

Applicability : Use the State pattern when

1. An object's behavior depends on its state, and it must change itsbehavior at run-time depending on that state.
2. Operations have large, multipart conditional statements that dependon the object's state.

Structure :



Participants :

1. Context : Define an interface of interest to clients. Maintains an instance of a ConcreteState subclass that defines the current state.
2. State : Defines an interface for encapsulating the behaviorassociated with a particular state of the context.
3. ConcreteState Subclasses : Each subclass implements a behaviorassociated with a state of the Context.

Collaborations :

1. Context delegates state-specific requests to the currentConcreteState object.

2. A context may pass itself as an argument to the State objecthandling the request.
3. Context is the primary interface for clients.
4. Either Context or the ConcreteState subclasses can decide whichstate succeeds another and under what circumstances.

Consequences :

1. It localizes state-specific behavior and pattern behavior fordifferent states.
2. It makes state transitions explicit.
3. Sate objects can be shared.

Implementation :

1. Who defines the state transition?
2. A table based alternative.
3. Creating and destroying State objects.
4. Using dynamic inheritance.

5.3.4. Strategy Scope : Object Purpose : Behavioral

Intent : Define a family of algorithms, encapsulate each one, and makethem interchangeable. Strategy lets the algorithm vary independently from clients that use it.

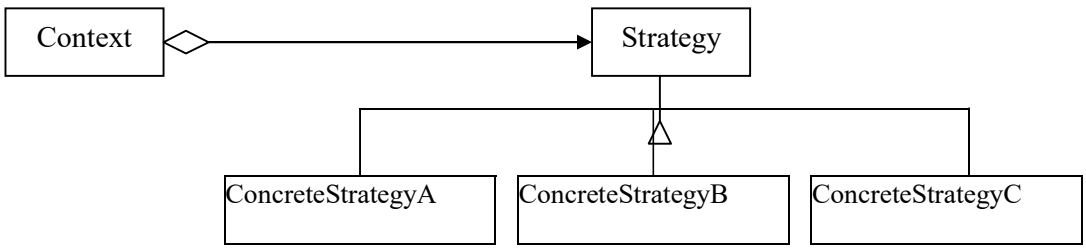
Also Known As : Policy.

Motivation : Many algorithms exist for breaking a stream of text intolines.

Applicability : Use the Strategy pattern when

1. Many related classes differ only in their behavior.
2. You need different variants of an algorithm.
3. An algorithm uses data that clients shouldn't know about.
4. A class defines many behaviors, and these appear as multipleconditional statements in its operations.

Structure :



Participants :

1. Strategy : Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm definedby a ConcreteStrategy.
2. ConcreteStrategy : Implements the algorithm using the Strategyinterfaces.
3. Context : Is configured with a ConcreteStrategy object. Maintains areference to a strategy object. May define an interface that lets Strategy access its data.

Collaborations :

1. Strategy and Context interact to implement the chosen algorithm.
2. A context forwards requests from its client to its strategy.

Consequences :

1. Families of related algorithm.
2. An alternative to subclassing.
3. Strategy eliminates conditional statements.
4. A choice of implementation.
5. Client must be aware of different Strategies.
6. Communication overhead between Strategy and Context.
7. Increased number of objects.

Implementation :

1. Defining the Strategy and Context interface.

2. Strategies as template parameters.
3. Making Strategies objects optional.

GRASP

1. Responsibilities

The UML defines a responsibility as "a contract or obligation of a classifier." Responsibilities are related to the obligations or behavior of anobject in terms of its role.

Types of Responsibilities : Basically, responsibilities are of the twotypes :

1. **Doing :** Responsibilities of an object include :

i. Doing something itself, such as creating an object ordoing a calculation.
ii. Initiating action in other objects.
iii. Controlling and coordinating activities in other products.

2. **Knowing :** Responsibilities of an object include :

i. Knowing about private encapsulated data.
ii. Knowing about related objects.
iii. Knowing about things that can derive or calculate.

5.4 GRASP

Stands for General Responsibility Assignment Software Patterns. The name was chosen to suggest the importance of grasping these principlesto the successful design object-oriented software.

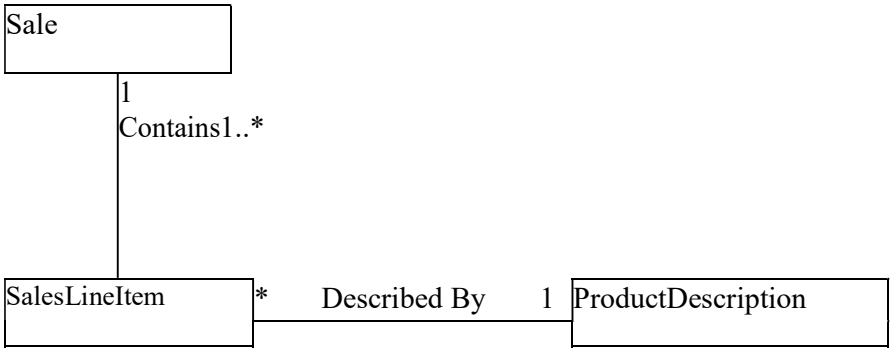
1. **Creator GRASP**

Problem : Who should be responsible for creating a new instance ofsome class?

Solution : Assign class B the responsibility to create an instance of classA if one of the these is true :

i. B "contains" or compositely aggregates A.
ii. B records A.
iii. B closely uses A.
iv. B has the initiating data for A.B is a creator of A objects.

Example : Who should be responsible for creating an instance ofSalesLineItem Instanc



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

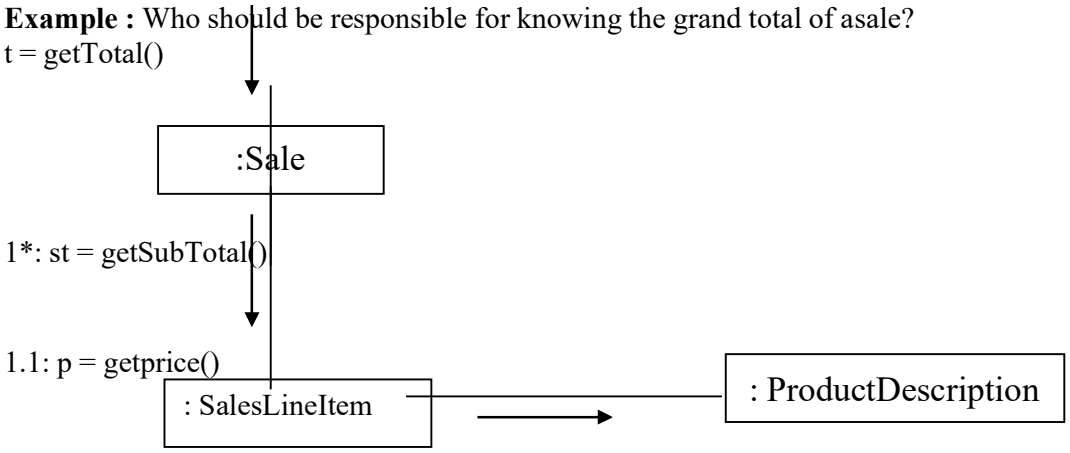
Benefits : Low coupling is supported.

2. **Information Expert(or Expert) GRASP.**

Problem : What is a general principle of assigning responsibilities toobjects?

Solution : Assign a responsibility to the information expert-the class thathas the information necessary to fulfill the responsibility.

Example : Who should be responsible for knowing the grand total of asale?



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

i. Sale knows sale total.
ii. SalesLineItem knows line item subtotal.
iii. ProductDescription knows product price.

Benefits :

1. Information encapsulation is maintained since objects use theirown information to fulfill tasks.
2. Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" classdefinitions that are easier to understand and maintain.
3. Support both Low Coupling and High Cohesion.

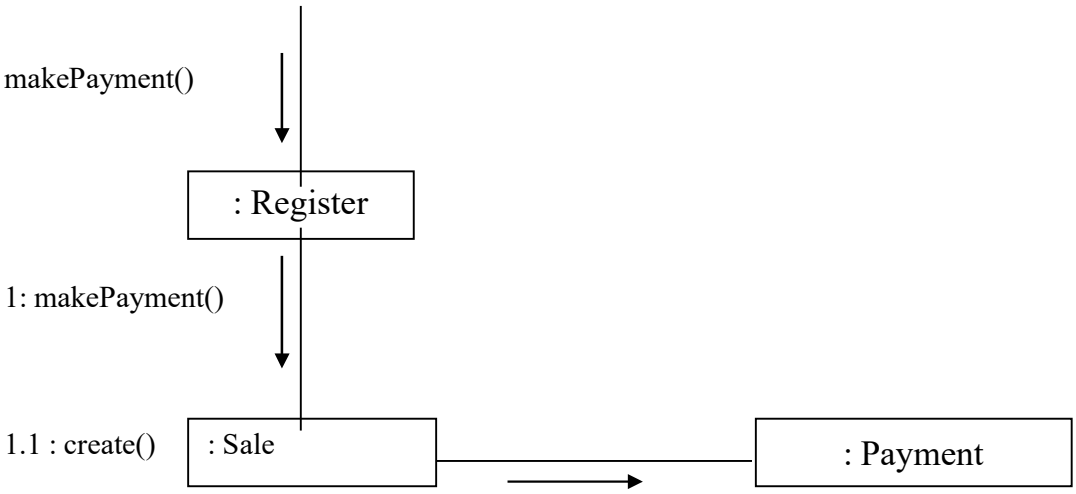
3. Low Coupling GRASP.

Problem : How to support low dependency, low change impact, andincreased reuse

Coupling : Is a measure of how strongly one element is connected to, hasknowledge of, or relies on other elements.

Solution : Assign a responsibility so that coupling remains low. Use thisprinciple to evaluate alternatives.

Example : Assume we need to create a Payment instance and associate itwith the Sale. What class should be responsible for this?



Create a Payment instance and associate it with Sale.

Common forms of coupling : Common forms of coupling from TypeXto TypeY include the following :

i. TypeX has an attribute that refers to a TypeY instance, orTypeY itself.
ii. A TypeX object calls on services of a TypeY object.
iii. TypeX has a method that references an instance of TypeY, orTypeY itself, by any means.
iv. TypeX is a direct or indirect subclass of TypeY.

v. TypeY is an interface, and TypeX implements that interface.

Benefits :

1. Not affected by changes in other components.
2. Simple to understand in isolation.
3. Convenient to reuse.

4. Controller GRASP

Problem : What first object beyond the UI layer received and coordinatessystem operations.

Controller : Is the first object beyond the UI layer that is responsible forreceiving or handling operation message.

Solution : Assign the responsibility to a class representing one of thefollowing choices :

i. Represent the overall "system", a "root object", a device that thesoftware is running within, or a major subsystem.

ii. Represent a use case scenario within which the system eventoccurs.

Benefits :

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

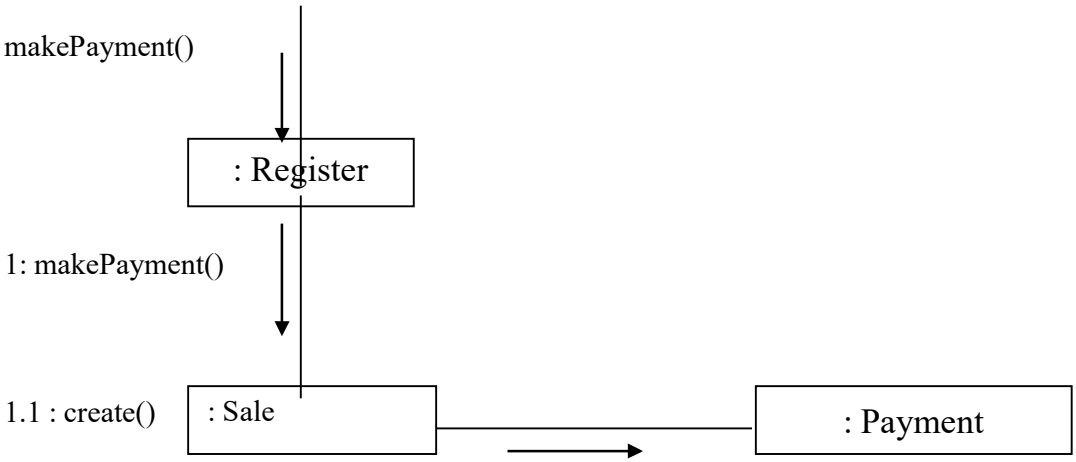
5. High Cohesion GRASP

Problem : How to keep objects focused, understandable, andmanageable, support Low Coupling?

Cohesion : In terms of object design, cohesion is a measure of howstrongly related and focused the responsibilities of an element are.

Solution : Assign a responsibility so that cohesion remains high. Use thisto evaluate alternatives.

Example : Assume we need to create a Payment instance and associatewith it Sale. What class should be responsible for this?



Delegate the Payment creation responsibility to the Sale supportshigher cohesion in the Register.

Degrees of Functional Cohesion : Here are some scenarios that illustratevarying degrees of functional cohesion :

1. **Very Low Cohesion :** A class is solely responsible for manythings in very different functional areas.
2. **Low Cohesion :** A class has sole responsibility for a complex taskin one functional area.
3. **High Cohesion :** A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill task.
4. **Moderate Cohesion :** A class has lightweight and sole responsibilities in few different areas that logically related to theclass concept but not to each other.

Benefits :

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

6. Polymorphism GRASP

Problem : How handle alternatives based on types? How to createpluggable software components?

Solution : When related alternatives or behaviors vary by type(class), assign responsibility for the behavior-using polymorphic operations-tothe types for which the behavior varies.
Example : What objects should be responsible for handling these varyingexternal tax calculator interfaces?

Since the behavior of calculator adaptation varies by the type of calculator, by Polymorphism we should assign the responsibility for adaptation to different calculator objects themselves, implemented withpolymorphic getTaxes operation.

Benefits :

1. Extension required for new variations are easy to add.
2. New implementation can be introduced without affecting clients.

7. Pure Fabrication GRASP

Problem : What object should have the responsibility, when you do notwant to violate High Cohesion and Low Coupling, or other goals, but solution offered by Expert are not appropriate?
Solution : Assign a highly cohesive set of responsibilities to an artificialor convenience class that does not represent a problem domain concept –something made up, to support high cohesion, low coupling, and reuse.
Example : Suppose that support is needed to save Sale instances in thedatabase.

A reasonable solution is to create a new class that is solely responsible for saving objects in some kind of persistent storage medium,such as a relational database; call it the PesistentStorage. This class is a Pure Fabrication.

Benefits :

1. High Cohesion is supported because responsibilities are factored into a fine-grained class that only focuses on a very specific set ofrelated tasks.
2. Reuse potential may increase because of the presence of fine-grained Pure Fabrication classes whose responsibilities have applicability in other applications.

8. Indirection GRASP

Problem : Where to assign a responsibility, to avoid direct couplingbetween two(or more) things? How to de-couple objects so that lowcoupling is supported and reuse potential remains higher?
Solution : Assign the responsibility to an intermediate object to mediatebetween other components or services so that they are not directly coupled.

The intermediary creates an indirection between the othercomponents.

Example : TaxCalculatorAdaptor
These objects act as intermediaries to the external tax calculators.
Via polymorphism, they provide a consistent interface to the inner objectsand hide the variations in the external API's.

By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the externalinterfaces.

Benefit : Lower Coupling between components.

9. Protected Variations GRASP

Problem : How to design objects, subsystems, and systems so that thevariations or instability in these elements does not have an undesirableimpact on other elements?
Solution : Identify points of predicted variation or instability; assignresponsibilities to create a stable interface around them.
Example : What objects should be responsible for handling these varyingexternal tax calculator interfaces?

The point of instability or variation is the different interfaces or API's of external tax calculators. The POS system needs to be able to integrate with many existing tax calculator systems, and also with futurethird-party calculators not yet in existence.

Benefits :

1. Extensions required for new variations are easy to add.
2. New implementations can be introduced without affecting clients.
3. Coupling is lowered.
4. The impact or cost of changes can be lowered.

University Paper Questions

1. Which two design patterns are known as wrapper?
2. “Object Inheritance is white box reuse”. Justify.
3. What is the intent of Observer Pattern?
4. Give the definition, uses and structure of Decorator Design Pattern.
5. What are the main participant’s classes in the Strategy Pattern?
6. Explain applicability, structure of Strategy Design Pattern.
7. What is coupling? “Low Coupling is desirable” comment. Discussthe problem of assigning responsibility to reduce coupling.
8. Describe the participants of Decorator Pattern.
9. Why composition should be favored over inheritance? Give thedesign pattern using the above principle.
10. Give the definition, participants and structure for adapter designpattern.
11. What is coupling? What are problems with high coupling?
12. Which four types of classes, according to GRASP pattern, shouldbe assigned responsibility of handling system events?
13. Give the participants and collaboration in state design pattern.
14. Describe each of the three creational design patterns and state howall the three are related.
15. Define facade design pattern. Discuss its advantages.
16. State the benefits using Singleton Pattern.
17. What is decorator pattern? State its participants.
18. Describe the patterns assigning responsibility in GRASP.
19. What are drawbacks of Strategy Pattern?
20. What are consequences of Facade Design Pattern?
21. What are two types of responsibilities? Give examples of eachtype.
22. Give applicability and consequences of Singleton Design Pattern.
23. Discuss structure and participants of Observer Design Pattern.’
24. Explain intent, participants and consequences of Decorator DesignPattern.
25. “Assign responsibilities so that cohesion remains high”. Justify.

4.3 Describing Design Patterns

. • Pattern Name and Classification: A descriptive and unique name that helps in identifying and referring to the pattern. • Intent: A description of the goal behind the pattern and the reason for using it. • Also Known As: Other names for the pattern. • Motivation (Forces): A scenario consisting of a problem and a context in which this pattern can be used. • Applicability: Situations in which this pattern is usable; the context for the pattern. • Structure: A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose. • Participants: A listing of the classes and objects used in the pattern and their roles in the design. • Collaboration: A description of how classes and objects used in the pattern interact with each other • Consequences: A description of the results, side effects, and tradeoffs caused by using the pattern. • Implementation: A description of an implementation of the pattern; the solution part of the pattern. • Sample Code: An illustration of how the pattern can be used in a programming language. • Known Uses: Examples of real usages of the pattern. • Related Patterns: Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.