A

***Course File Report***

**On**

***“DESIGN PATTERNS”***



*department of*

***Computer Science Engineering***

***CMR ENGINEERING COLLEGE***

**(Affiliated to J.N.T.U, HYDERABAD)**

Kandlakoya(v),Medchal -501 401

**(2016-17)**

**UNIT-I**

**INTRODUCTION**

**What is a Design Pattern?**

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

**Design Patterns in Smalltalk MVC:**

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

The following diagram shows a model and three views. The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.



Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. Another feature of MVC is that views can be nested. MVC also lets you change the way a view responds to user input without changing its visual presentation.

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input.

**Describing Design Patterns:**

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

**Pattern Name and Classification:**

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

**Intent:**

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

**Also Known As:**

Other well-known names for the pattern, if any.

**Motivation:**

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

**Applicability:**

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

**Structure:**

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). We also use interaction diagrams to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

**Participants:**

The classes and/or objects participating in the design pattern and their responsibilities.

**Collaborations:**

How the participants collaborate to carry out their responsibilities.

**Consequences:**

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

**Implementation:**

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

**Sample Code:**

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

**Known Uses:**

Examples of the pattern found in real systems. We include at least two examples from different domains.

**Related Patterns:**

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

**The Catalog of Design Patterns:**

**Abstract Factory**

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Adapter**

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Bridge**

Decouple an abstraction from its implementation so that the two can vary independently.

**Builder**

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Chain of Responsibility**

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**Command**

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**Composite**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Decorator**

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

**Facade**

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

**Factory Method**

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Flyweight**

Use sharing to support large numbers of fine-grained objects efficiently.

**Interpreter**

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**Iterator**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Mediator**

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Memento**

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

**Observer**

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

**Prototype**

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Proxy**

Provide a surrogate or placeholder for another object to control access to it.

**Singleton**

Ensure a class only has one instance, and provide a global point of access to it.

**State**

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**Strategy**

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

**Template Method**

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**Visitor**

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Organizing the Catalog**

We classify design patterns by two criteria. The first criterion, called **purpose**, reflects what a pattern does. Patterns can have **creational**, **structural**, or **behavioral** purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion, called **scope**, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships.

**How Design Patterns Solve Design Problems?**

Design patterns solve many of the day-to-day problems object-oriented designers face, and in any different ways. Here are several of these problems and how design patterns solve them.

* Finding Appropriate Objects
* Determining Object Granularity
* Specifying Object Interfaces
* Specifying Object Implementations
* Putting Reuse Mechanisms to Work
* Designing for Change

**How to Select a Design Pattern?**

Here are several different approaches to finding the design pattern that's right for your problem:

1. *Consider how design patterns solve design problems.*
2. *Scan Intent sections.*
3. *Study how patterns interrelate.*
4. *Study patterns of like purpose.*
5. *Examine a cause of redesign.*
6. *Consider what should be variable in your design.*

**How to Use a Design Pattern?**

Here's a step-by-step approach to applying a design pattern effectively:

1. *Read the pattern once through for an overview.* Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.

2. *Go back and study the Structure, Participants, and Collaborations sections.*

Make sure you understand the classes and objects in the pattern and how they relate to one another.

3. *Look at the Sample Code section to see a concrete example of the pattern in code.* Studying the code helps you learn how to implement the pattern.

4. *Choose names for pattern participants that are meaningful in the application context.* The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation.

5. *Define the classes.* Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.

6. *Define application-specific names for operations in the pattern.* Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.

7. *Implement the operations to carry out the responsibilities and collaborations in the pattern.* The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

**UNIT-II**

**A Case Study: Design a Document Editor**

This chapter presents a case study in the design of a "What-You-See-Is-What-You-Get" (or "WYSIWYG") document editor called **Lexi**. We'll see how design patterns capture solutions to design problems in Lexi and applications like it.

**Design Problems**

We will examine seven problems in Lexi's design:

1. *Document structure.* The choice of internal representation for the document affects nearly every aspect of Lexi's design. All editing, formatting, displaying, and textual analysis will require traversing the representation. The way we organize this information will impact the design of the rest of the application.

2. *Formatting.* How does Lexi actually arrange text and graphics into lines and columns? What objects are responsible for carrying out different formatting policies? How do these policies interact with the document's internal representation?

3. *Embellishing the user interface.* Lexi's user interface includes scroll bars, borders, and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexi's user interface evolves. Hence it's important to be able to add and remove embellishments easily without affecting the rest of the application.

4. *Supporting multiple look-and-feel standards.* Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.

5. *Supporting multiple window systems.* Different look-and-feel standards are usually implemented on different window systems. Lexi's design should be as independent of the window system as possible.

6. *User operations.* Users control Lexi through various user interfaces, including buttons and pull-down menus. The functionality behind these interfaces is scattered throughout the objects in the application. The challenge here is to provide a uniform mechanism both for accessing this scattered functionality and for undoing its effects.

7. *Spelling checking and hyphenation.* How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points? How can we minimize the number of classes we have to modify to add a new analytical operation?

**Document Structure**

A document is ultimately just an arrangement of basic graphical elements such as characters, lines, polygons, and other shapes. These elements capture the total information content of the document.

Lexi's user interface should let users manipulate these substructures directly. For example, a user should be able to treat a diagram as a unit rather than as a collection of individual graphical primitives. The user should be able to refer to a table as a whole, not as an unstructured mass of text and graphics. That helps make the interface simple and intuitive. To give Lexi's implementation similar qualities, we'll choose an internal representation that matches the document's physical structure.

In particular, the internal representation should support the following:

* Maintaining the document's physical structure, that is, the arrangement of text and graphics into lines, columns, tables, etc.
* Generating and presenting the document visually.
* Mapping positions on the display to elements in the internal representation. This lets Lexi determine what the user is referring to when he points to something in the visual representation.

**Formatting**

We've settled on a way to *represent* the document's physical structure. Next, we need to figure out how to construct a *particular* physical structure, one that corresponds to a properly formatted document. Representation and formatting are distinct: The ability to capture the document's physical structure doesn't tell us how to arrive at a particular structure. This responsibility rests mostly on Lexi. It must break text into lines, lines into columns, and so on, taking into account the user's higher-level desires. For example, the user might want to vary margin widths, indentation, and tabulation; single or double space; and probably many other formattingconstraints.6Lexi'sformatting algorithm must take all of these into account.

* **Encapsulating the Formatting Algorithm**

The formatting process, with all its constraints and details, isn't easy to automate. There are many approaches to the problem, and people have come up with a variety of formatting algorithms with different strengths and weaknesses. Because Lexi is a WYSIWYG editor, an important trade-off to consider is the balance between formatting quality and formatting speed. We want generally good response from the editor without sacrificing how good the document looks. This trade-off is subject to many factors, not all of which can be ascertained at compile-time.

* **Compositor and Composition**

We'll define a **Compositor** class for objects that can encapsulate a formatting algorithm. The interface lets the compositor know *what* glyphs to format and *when* to do the formatting. The glyphs it formats are the children of a special Glyph subclass called **Composition**. A composition gets an instance of a Compositor subclass (specialized for a particular line breaking algorithm) when it is created, and it tells the compositor to Compose its glyphs when necessary.



Figure: Composition and Compositor class relationships

**Embellishing the User Interface**

We consider two embellishments in Lexi's user interface. The first adds a border around the text editing area to demarcate the page of text. The second adds scroll bars that let the user view different parts of the page. To make it easy to add and remove these embellishments (especially at run-time), we shouldn't use inheritance to add them to the user interface. We achieve the most flexibility if other user interface objects don't even know the embellishments are there. That will let us add and remove the embellishments without changing other classes.

**Supporting Multiple Look-and-Feel Standards**

Achieving portability across hardware and software platforms is a major problem in system design. Retargeting Lexi to a new platform shouldn't require a major overhaul, or it wouldn't be worth retargeting. We should make porting as easy as possible.

One obstacle to portability is the diversity of look-and-feel standards, which are intended to enforce uniformity between applications. These standards define guidelines for how applications appear and react to the user. While existing standards aren't that different from each other, people certainly won't confuse one for the other—Motif applications don't look and feel exactly like their counterparts on other platforms, and vice versa. An application that runs on more than one platform must conform to the user interface style guide on each platform. Our design goals are to make Lexi conform to multiple existing look-and-feel standards and to make it easy to add support for new standards as they (invariably) emerge. We also want our design to support the ultimate in flexibility: changing Lexi's look and feel at run-time.

**Supporting Multiple Window Systems**

Look and feel is just one of many portability issues. Another is the windowing environment in which Lexi runs. A platform's window system creates the illusion of multiple overlapping windows on a bitmapped display. It manages screen space for windows and routes input to them from the keyboard and mouse. Several important and largely incompatible window systems exist today (e.g., Macintosh, Presentation Manager, Windows, X).We'd like Lexi to run on as many of them as possible for exactly the same reasons we support multiple look-and-feel standards.

**User Operations**

Some of Lexi's functionality is available through the document's WYSIWYG representation. You enter and delete text, move the insertion point, and select ranges of text by pointing, clicking, and typing directly in the document. Other functionality is accessed indirectly through user operations in Lexi's pull-down menus, buttons, and keyboard accelerators. The functionality includes operations for

* creating a new document,
* opening, saving, and printing an existing document,
* cutting selected text out of the document and pasting it back in,
* changing the font and style of selected text,
* changing the formatting of text, such as its alignment and justification,
* quitting the application,
* and on and on.

Lexi provides different user interfaces for these operations. But we don't want to associate a particular user operation with a particular user interface, because we may want multiple user interfaces to the same operation.

**Spelling Checking and Hyphenation**

The last design problem involves textual analysis, specifically checking for misspellings and introducing hyphenation points where needed for good formatting. As was the case for line breaking strategies, there's more than one way to check spelling and compute hyphenation points. So here too we want to support multiple algorithms. A diverse set of algorithms can provide a choice of space/time/quality trade-offs. We should make it easy to add new algorithms as well.

We also want to avoid wiring this functionality into the document structure. This goal is even more important here than it was in the formatting case; because spelling checking and hyphenation are just two of potentially many kinds of analyses we may want Lexi to support. Inevitably we'll want to expand Lexi's analytical abilities over time. We might add searching, word counting, a calculation facility for adding up tabular values, grammar checking, and so forth. But we don't want to change the Glyph class and all its subclasses every time we introduce new functionality of this sort.

**Creational Patterns**

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

**Abstract Factory**

**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Also Known As:** Kit

**Motivation:** Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.

**Structure:**



**Participants:**

* **AbstractFactory**
* declares an interface for operations that create abstract product objects.
* **ConcreteFactory**
* implements the operations to create concrete product objects.
* **AbstractProduct**
* declares an interface for a type of product object.
* **ConcreteProduct**
* defines a product object to be created by the corresponding concrete factory.
* implements the AbstractProduct interface.
* **Client**
* uses only interfaces declared by AbstractFactory and AbstractProduct classes.

**Consequences**

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

**Builder**

**Intent:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Motivation:** A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

A solution is to configure the RTFReader class with a TextConverter object that converts RTF to another textual representation. As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion. Whenever the RTFReader recognizes an RTF token (either plain text or an RTF control word), it issues a request to the TextConverter to convert the token. TextConverter objects are responsible both for performing the data conversion and for representing the token in a particular format.

**Structure:**



**Participants:**

* **Builder**
* specifies an abstract interface for creating parts of a Product object.
* **ConcreteBuilder**
* constructs and assembles parts of the product by implementing the Builder interface.
* defines and keeps track of the representation it creates.
* provides an interface for retrieving the product.
* **Director**
* constructs an object using the Builder interface.
* **Product**
* represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
* includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

**Consequences:**

* *It lets you vary a product's internal representation.*
* *It isolates code for construction and representation.*
* *It gives you finer control over the construction process.*

**Factory Method**

**Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Also Known As:** Virtual Constructor

**Motivation:** Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations.

**Structure:**



**Participants:**

* **Product**
* defines the interface of objects the factory method creates.
* **ConcreteProduct**
* implements the Product interface.
* **Creator**
* declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
* may call the factory method to create a Product object.
* **ConcreteCreator**
* overrides the factory method to return an instance of a ConcreteProduct.

**Consequences:**

Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes. A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Sub classing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

Here are two additional consequences of the Factory Method pattern:

* *Provides hooks for subclasses.* Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.
* *Connects parallel class hierarchies.*

**Prototype**

**Intent:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Motivation:** You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves. The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects. Users will click on the quarter-note tool and use it to add quarter notes to the score. Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.

**Structure:**



**Participants:**

* **Prototype**
	+ declares an interface for cloning itself.
* **ConcretePrototype**
	+ implements an operation for cloning itself.
* **Client**
	+ creates a new object by asking a prototype to clone itself.

**Consequences:**

Prototype has many of the same consequences that Abstract Factory and Builder have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about.

Additional benefits of the Prototype pattern are listed below.

* *Adding and removing products at run-time.*
* *Specifying new objects by varying values.*
* *Specifying new objects by varying structure.*
* *Reduced sub classing.*
* *Configuring an application with classes dynamically.*

**Singleton**

**Intent:** Ensure a class only has one instance, and provide a global point of access to it.

**Motivation:** It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

**Structure:**

****

**Participants:**

* **Singleton**
	+ defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
	+ may be responsible for creating its own unique instance.

**Consequences:**

The Singleton pattern has several benefits:

1. *Controlled access to sole instance.* Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. *Reduced name space.* The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. *Permits refinement of operations and representation.* The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.
4. *Permits a variable number of instances.* The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
5. *More flexible than class operations.* Another way to package a singleton's functionality is to use class operations.

**What to Expect from Design Patterns?**

Here are several ways in which the design patterns can affect the way you design object-oriented software, based on our day-to-day experience with them.

* A Common Design Vocabulary
* A Documentation and Learning Aid
* An Adjunct to Existing Methods
* A Target for Refactoring

**UNIT-III**

**Structural Patterns**

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural *class* patterns use inheritance to compose interfaces or implementations.

Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

**Adapter**

**Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Also Known As:** Wrapper

**Motivation:** Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

**Structure:**

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



**Participants:**

* **Target -** defines the domain-specific interface that Client uses.
* **Client -** collaborates with objects conforming to the Target interface.
* **Adaptee -** defines an existing interface that needs adapting.
* **Adapter -** adapts the interface of Adaptee to the Target interface.

**Consequences:**

Class and object adapters have different trade-offs. A class adapter

* adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
* lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
* introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

* lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
* makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

**Bridge**

**Intent:** Decouple an abstraction from its implementation so that the two can vary independently.

**Also Known As:** Handle/Body

**Motivation:** When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.

**Structure:**

****

**Participants:**

* **Abstraction** - defines the abstraction's interface, maintains a reference to an object of type Implementor.
* **RefinedAbstraction** - Extends the interface defined by Abstraction.
* **Implementor -** defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstract in defines higher-level operations based on these primitives.
* **ConcreteImplementor** - implements the Implementor interface and defines its concrete implementation.

**Consequences:**

The Bridge pattern has the following consequences:

1. *Decoupling interface and implementation.* An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.
2. *Improved extensibility.* You can extend the Abstraction and Implementor hierarchies independently.
3. *Hiding implementation details from clients.* You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).

**Composite**

**Intent:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Motivation:** Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.

**Structure:**



A typical Composite object structure might look like this:



**Participants:**

* **Component**
* declares the interface for objects in the composition.
* implements default behavior for the interface common to all classes, as appropriate.
* declares an interface for accessing and managing its child components.
* **Leaf**
* represents leaf objects in the composition. A leaf has no children.
* defines behavior for primitive objects in the composition.
* **Composite**
* defines behavior for components having children.
* stores child components.
* implements child-related operations in the Component interface.
* **Client**
* manipulates objects in the composition through the Component interface.

**Consequences:**

The Composite pattern

* defines class hierarchies consisting of primitive objects and composite objects.
* makes the client simple. Clients can treat composite structures and individual objects uniformly.
* makes it easier to add new kinds of components.
* can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite.

**Decorator**

**Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Also Known As:** Wrapper

**Motivation:**

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

**Structure:**



**Participants:**

* **Component**
* defines the interface for objects that can have responsibilities added to them dynamically.
* **ConcreteComponent**
* defines an object to which additional responsibilities can be attached.
* **Decorator**
* maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* **ConcreteDecorator**
* adds responsibilities to the component.

**Consequences:**

The Decorator pattern has at least two key benefits and two 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.*

**Façade**

**Intent:** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

**Motivation:** Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.



**Structure:**



**Participants:**

* **Facade**
* knows which subsystem classes are responsible for a request.
* delegates client requests to appropriate subsystem objects.
* **subsystem classes**
* implement subsystem functionality.
* handle work assigned by the Facade object.
* have no knowledge of the facade; that is, they keep no references to it.

**Consequences:**

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
2. It promotes weak coupling between the subsystem and its clients.
3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

**Flyweight**

**Intent:** Use sharing to support large numbers of fine-grained objects efficiently.

**Motivation:** Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive. The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.

A **flyweight** is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate. The key concept here is the distinction between **intrinsic** and **extrinsic** state. Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable. Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

**Structure:**



The following object diagram shows how flyweights are shared:



**Participants:**

* **Flyweight**
* declares an interface through which flyweights can receive and act on extrinsic state.
* **ConcreteFlyweight**
* implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
* **UnsharedConcreteFlyweight**
* not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure
* **FlyweightFactory**
* creates and manages flyweight objects.
* ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
* **Client**
* maintains a reference to flyweight(s).
* computes or stores the extrinsic state of flyweight(s).

**Consequences:**

Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.

**Proxy**

**Intent:** Provide a surrogate or placeholder for another object to control access to it.

**Also Known As:** Surrogate

**Motivation:**

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.

These constraints would suggest creating each expensive object *on demand*, which in this case occurs when an image becomes visible. But what do we put in the document in place of the image? And how can we hide the fact that the image is created on demand so that we don't complicate the editor's implementation? This optimization shouldn't impact the rendering and formatting code, for example. The solution is to use another object, an image **proxy** that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.

**Structure:**



Here's a possible object diagram of a proxy structure at run-time:



**Participants:**

* **Proxy**
* maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
* provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
* controls access to the real subject and may be responsible for creating and deleting it.
* **Subject**
* defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
* **RealSubject**
* defines the real object that the proxy represents.

**Consequences:**

The Proxy pattern introduces a level of indirection when accessing an object.

The additional indirection has many uses, depending on the kind of proxy:

1. A remote proxy can hide the fact that an object resides in a different address space.

2. A virtual proxy can perform optimizations such as creating an object on demand.

3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

**UNIT-IV**

**BEHAVIORAL PATTERNS**

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.

**Chain of Responsibility**

**Intent:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**Applicability:**

Use Chain of Responsibility when

* more than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
* you want to issue a request to one of several objects without specifying the receiver explicitly.
* the set of objects that can handle a request should be specified dynamically.

**Structure:**



A typical object structure might look like this:



**Participants:**

* **Handler**
* defines an interface for handling requests.
* **ConcreteHandler**
* handles requests it is responsible for.
* can access its successor.
* if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
* **Client**
* initiates the request to a ConcreteHandler object on the chain.

**Consequences:**

Chain of Responsibility has the following benefits and liabilities:

1. *Reduced coupling.* The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled “appropriately”. Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure.
2. *Added flexibility in assigning responsibilities to objects.*
3. *Receipt isn't guaranteed.* Since a request has no explicit receiver, there's no *guarantee* it'll be handled—the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

**Command**

**Intent:** Encapsulate a request as an object; thereby letting you parameterized clients with different requests, queue or log requests, and support undoable operations.

**Also Known As:** Action, Transaction

**Applicability:**

Use the Command pattern when you want to

* parameterize objects by an action to perform, as Menu Item objects did above. You can express such parameterization in a procedural language with a **callback** function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
* specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
* support undo. The Command's Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.
* support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation.
* structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support **transactions**. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

**Structure:**



**Participants:**

* **Command**
* declares an interface for executing an operation.
* **ConcreteCommand**
* defines a binding between a Receiver object and an action.
* implements Execute by invoking the corresponding operation(s) on Receiver.
* **Client**
* creates a ConcreteCommand object and sets its receiver.
* **Invoker**
* asks the command to carry out the request.
* **Receiver**
* knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

**Consequences:**

The Command pattern has the following consequences:

1. Command decouples the object that invokes the operation from the one that knows how to perform it.
2. Commands are first-class objects. They can be manipulated and extended like any other object.
3. You can assemble commands into a composite command. An example is the MacroCommand class described earlier. In general, composite commands are an instance of the Composite pattern.
4. It's easy to add new Commands, because you don't have to change existing classes.

**Interpreter**

**Intent:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**Applicability:**

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time efficiency, is not a critical concern. The most efficient interpreters are usually *not* implemented by interpreting parse trees directly but by first translating them into another form.

**Structure:**



**Participants**

* **AbstractExpression**
* declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
* **TerminalExpression**
* implements an Interpret operation associated with terminal symbols in the grammar.
* an instance is required for every terminal symbol in a sentence.
* **NonterminalExpression**
* one such class is required for every rule *R* ::= *R*1 *R*2 ... *R*n in the grammar.
* maintains instance variables of type AbstractExpression for each of the symbols *R*1 through *R*n.
* implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing *R*1 through *R*n.
* **Context**
* contains information that's global to the interpreter.
* **Client**
* builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
* invokes the Interpret operation.

**Consequences:**

The Interpreter pattern has the following benefits and liabilities:

1. *It's easy to change and extend the grammar.*
2. *Implementing the grammar is easy, too.*
3. *Complex grammars are hard to maintain.*
4. *Adding new ways to interpret expressions.*

**Iterator**

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

**Also Known As:** Cursor

**Applicability:**

Use the Iterator pattern

* to access an aggregate object's contents without exposing its Internal representation.
* to support multiple traversals of aggregate objects.
* to provide a uniform interface for traversing different Aggregate structures (that is, to support polymorphic iteration).

**Structure:**



**Participants:**

* **Iterator**
* defines an interface for accessing and traversing elements.
* **ConcreteIterator**
* implements the Iterator interface.
* keeps track of the current position in the traversal of the aggregate.
* **Aggregate**
* defines an interface for creating an Iterator object.
* **ConcreteAggregate**
* implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

**Consequences:**

The Iterator pattern has three important consequences:

1. *It supports variations in the traversal of an aggregate.* Complex aggregates may be traversed in many ways.
2. *Iterators simplify the Aggregate interface.* Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
3. *More than one traversal can be pending on an aggregate.* An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

**Mediator**

**Intent:** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Applicability:**

Use the Mediator pattern when

* a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
* reusing an object is difficult because it refers to and communicates with many other objects.
* a behavior that's distributed between several classes should Be customizable without a lot of subclassing.

**Structure:**



A typical object structure might look like this:



**Participants:**

* **Mediator**
* defines an interface for communicating with Colleague objects.
* **ConcreteMediator**
* implements cooperative behavior by coordinating Colleague objects.
* knows and maintains its colleagues.
* **Colleague classes**
* each Colleague class knows its Mediator object.
* each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

**Consequences:**

The Mediator pattern has the following benefits and drawbacks:

1. *It limits subclassing.* A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.

2. *It decouples colleagues.* A mediator promotes loose coupling between colleagues. You can vary and reuse Colleague and Mediator classes independently.

3. *It simplifies object protocols.* A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain, and extend.

4. *It abstracts how objects cooperate.* Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior. That can help clarify how objects interact in a system.

5. *It centralizes control.* The Mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.

**Memento**

**Intent:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

**Also Known As:** Token

**Applicability:**

Use the Memento pattern when

* a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
* a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

**Structure:**



**Participants:**

* **Memento**
* stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
* **Originator**
* creates a memento containing a snapshot of its current internal state.
* uses the memento to restore its internal state.
* **Caretaker**
* is responsible for the memento's safekeeping.
* never operates on or examines the contents of a memento.

**Consequences:**

The Memento pattern has several consequences:

1. *It simplifies Originator.*
2. *Preserving encapsulation boundaries.*
3. *Using mementos might be expensive.*
4. *Defining narrow and wide interfaces.*
5. *Hidden costs in caring for mementos.*

**Observer**

**Intent:**

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

**Also Known As:** Dependents, Publish-Subscribe

**Applicability:**

Use the Observer pattern in any of the following situations:

* When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
* When a change to one object requires changing others, and you don't know how many objects need to be changed.
* When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

**Structure:**



**Participants:**

* **Subject**
* knows its observers. Any number of Observer objects may observe a subject.
* provides an interface for attaching and detaching Observer objects.
* **Observer**
* defines an updating interface for objects that should be notified of changes in a subject.
* **ConcreteSubject**
* stores state of interest to ConcreteObserver objects.
* sends a notification to its observers when its state changes.
* **ConcreteObserver**
* maintains a reference to a ConcreteSubject object.
* stores state that should stay consistent with the subject's.
* implements the Observer updating interface to keep its state consistent with the subject's.

**Consequences:**

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

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

**UNIT-IV**

**BEHAVIORAL PATTERNS**

**State**

**Intent:** Allow an object to alter its behavior when it’s internal state changes. The object will appear to change its class.

**Also Known As:** Objects for States

**Applicability:**

Use the State pattern in either of the following cases:

* An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
* Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

**Structure:**

****

**Participants:**

* **Context**
* defines the interface of interest to clients.
* maintains an instance of a ConcreteState subclass that defines the current state.
* **State**
* defines an interface for encapsulating the behavior associated with a particular state of the Context.
* **ConcreteState subclasses**
* each subclass implements a behavior associated with a state of the Context.

**Consequences:**

The State pattern has the following consequences:

1. *It localizes state-specific behavior and partitions behavior for different states.*
2. *It makes state transitions explicit.*
3. *State objects can be shared.*

 **Strategy**

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

**Also Known As:** Policy

**Applicability:**

Use the Strategy pattern when

* many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
* you need different variants of an algorithm. For example, you might Define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
* an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
* a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

**Structure:**

****

**Participants**

* **Strategy**
* declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
* **ConcreteStrategy**
* implements the algorithm using the Strategy interface.
* **Context**
* is configured with a ConcreteStrategy object.
* maintains a reference to a Strategy object.
* may define an interface that lets Strategy access its data.

**Template Method**

**Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**Structure:**

****

**Participants:**

* **AbstractClass**
* defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
* implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
* **ConcreteClass**
* implements the primitive operations to carry out subclass-specific steps of the algorithm.

**Consequences:**

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes.

**Visitor**

**Intent:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Applicability:**

Use the Visitor pattern when

* an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
* many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.

**Structure:**



**Participants**

* **Visitor**
* declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
* **ConcreteVisitor**
* implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
* **Element**
* defines an Accept operation that takes a visitor as an argument.
* **ConcreteElement**
* implements an Accept operation that takes a visitor as an argument.
* **ObjectStructure**
* can enumerate its elements.
* may provide a high-level interface to allow the visitor to visit its elements.