assuresoft

Software Development Core

What is Software Architecture?

Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems.

Each structure comprises software elements, relations among them, and properties of both elements and relations.

Software Architecture characteristics

Multitude of stakeholders: software systems have to cater to a variety of stakeholders such as business managers, owners, users, and operators. These stakeholders all have their own concerns with respect to the system. Balancing these concerns and demonstrating that they are addressed is part of designing the system.

Separation of concerns: the established way for architects to reduce complexity is to separate the concerns that drive the design. Architecture documentation shows that all stakeholder concerns are addressed by modeling and describing the architecture from separate points of view associated with the various stakeholder concerns.

Software Architecture characteristics

Quality-driven: closely related to its quality attributes such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and others.

Recurring styles: like building architecture, the software architecture discipline has developed standard ways to address recurring concerns.

Conceptual integrity: represents an overall vision of what it should do and how it should do it. This vision should be separated from its implementation.

Software Architecture characteristics

Cognitive constraints: refers to organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

Code review

Importance and usage: Dissociate code from coder, how to provide constructive feedback, how to accept negative feedback, own the code and align with the product best interests.

Pair Programming

Importance and usage: When to use, benefits and disadvantages, wrong usage, exploit.

Swarming

Importance and usage: When to use, pros and cons.

UML (Unified Modeling Language) 2.0

Examples: https://webpt.atlassian.net/wiki/spaces/OUT/pages/740100583/OutcomesIntake+Diagram

https://webpt.atlassian.net/wiki/spaces/PROD/pages/1384218625/Architecture+Proposal+Fax+Consolidation+for+EMR+1.0

Tutorial Examples:

https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html

UML (Unified Modeling Language) 2.0

Sequence diagram is a time dependent view of the interaction between objects to accomplish a behavioral goal of the system. The time sequence is similar to the earlier version of sequence diagram. An interaction may be designed at any level of abstraction within the system design, from subsystem interactions to instancelevel.

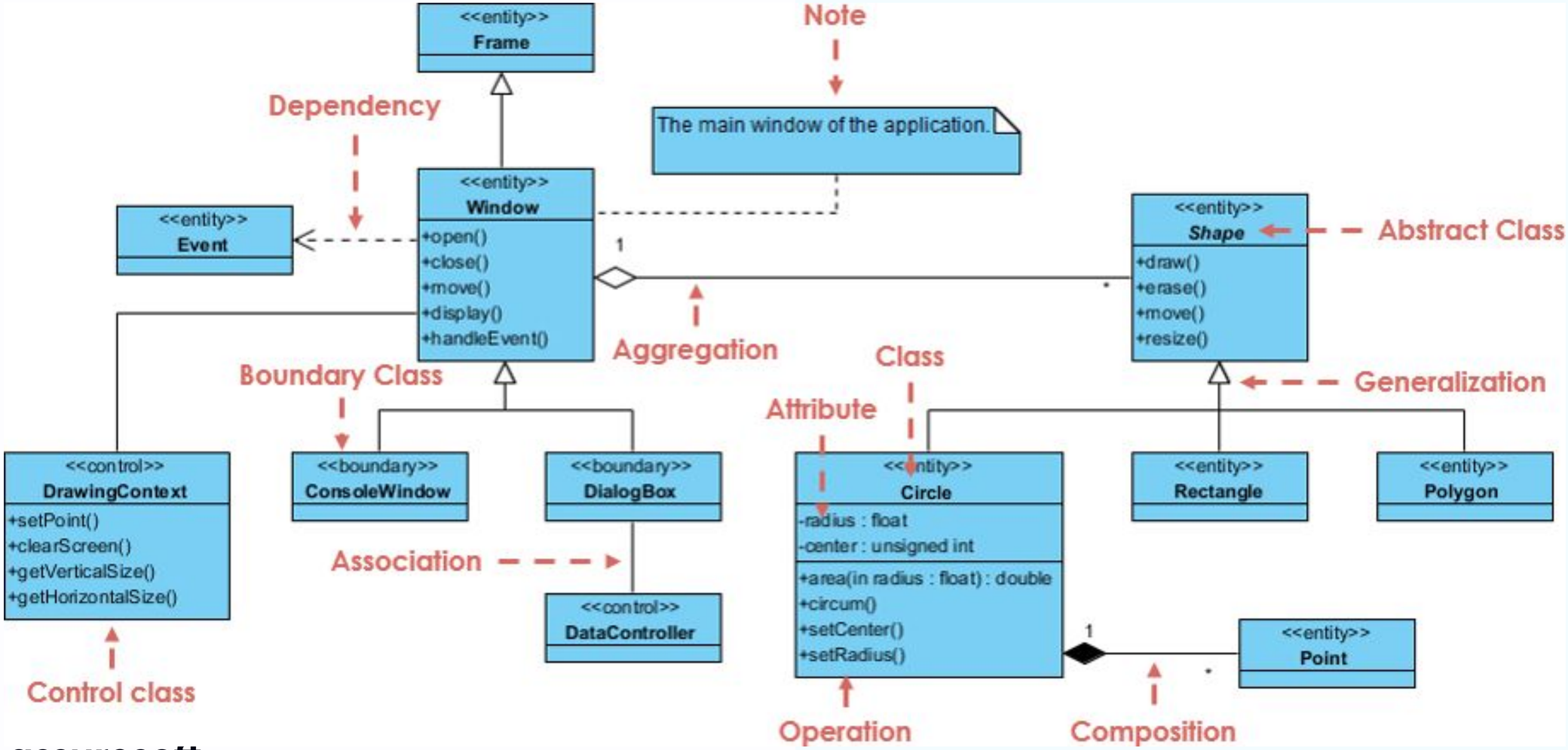
Communication diagram is a new name added in UML 2.0. Communication diagram is a structural view of the messaging between objects, taken from the Collaboration diagram concept of UML 1.4 and earlier versions. This can be defined as a modified version of collaboration diagram.

Interaction Overview diagram is also a new addition in UML 2.0. An Interaction Overview diagram describes a high-level view of a group of interactions combined into a logic sequence, including flow-control logic to navigate between the interactions.

Timing diagram is also added in UML 2.0. It is an optional diagram designed to specify the time constraints on the messages sent and received in the course of an interaction.

Specification: https://www.omg.org/spec/UML/2.5.1/About-UML/

UML (Unified Modeling Language) 2.0



assuresoft
Digital Academy

What are Design Patterns?

Design patterns are solutions to software design problems you find again and again in real-world application development.

Patterns are about reusable designs and interactions of objects.

The 23 Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral.

[Refactoring Guru - Design Patterns]

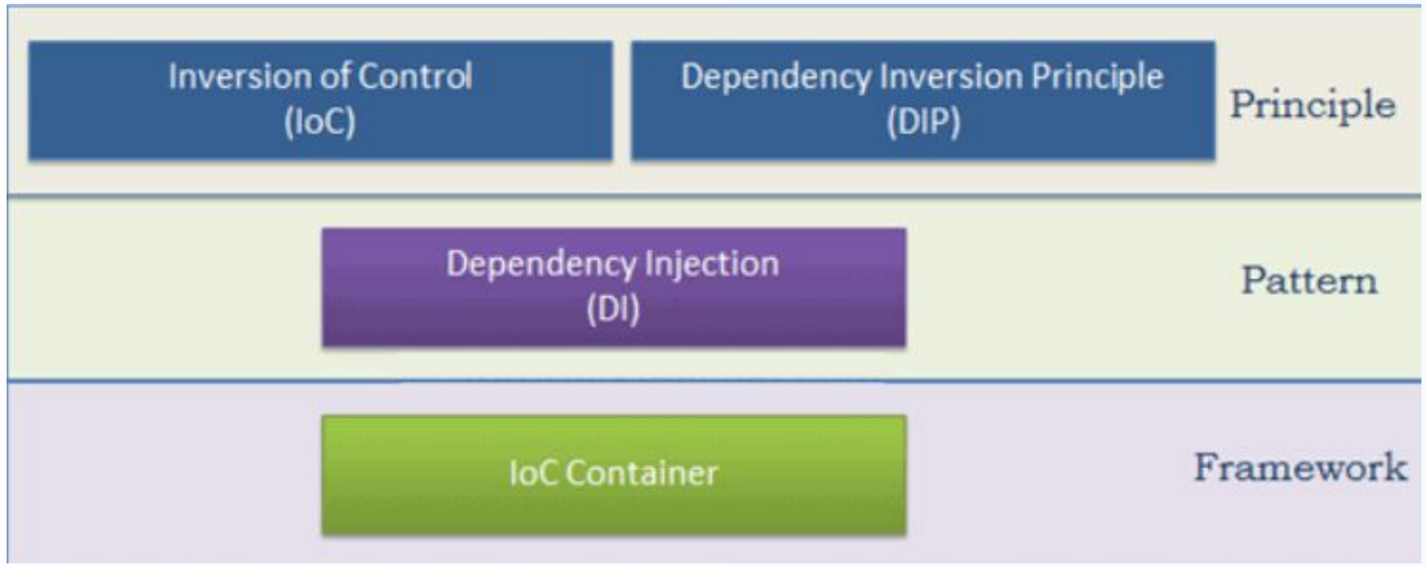
Difference between principle and pattern

Design Principle
Design principles provide high level guidelines to design better software applications. They do not provide implementation guidelines and are not bound to any programming language.
The SOLID (SRP, OCP, LSP, ISP, DIP) principles are one of the most popular sets of design principles.
For example, the Single Responsibility Principle (SRP) suggests that a class should have only one reason to change. This is a high-level statement which we can keep in mind while designing or creating classes for our application. SRP does not provide specific implementation steps but it's up to you how you implement SRP in your application.

Design Pattern
Design Pattern provides low-level solutions related to implementation, of commonly occurring object-oriented problems. In other words, design pattern suggests a specific implementation for the specific object-oriented programming problem. For example, if you want to create a class that can only have one object at a time, then you can use the Singleton design pattern which suggests the best way to create a class that can only have one object.
Design patterns are tested by others and are safe to follow, e.g. Abstract Factory, Factory, Singleton, Command, etc.

Dependency Injection.

You probably heard of Inversion of Control (IoC), Dependency Inversion Principle (DIP), Dependency Injection (DI), IoC containers or not at all, we will go through them.



assuresoft
Digital Academy

IoC, DIP, DI, IoC Container

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding (Remember SRP - Single Responsibility Principle).

The **DIP** principle also helps in achieving loose coupling between classes. It is highly recommended to use DIP and IoC together in order to achieve loose coupling.
DIP suggests that high-level modules should not depend on low level modules. Both should depend on abstraction.

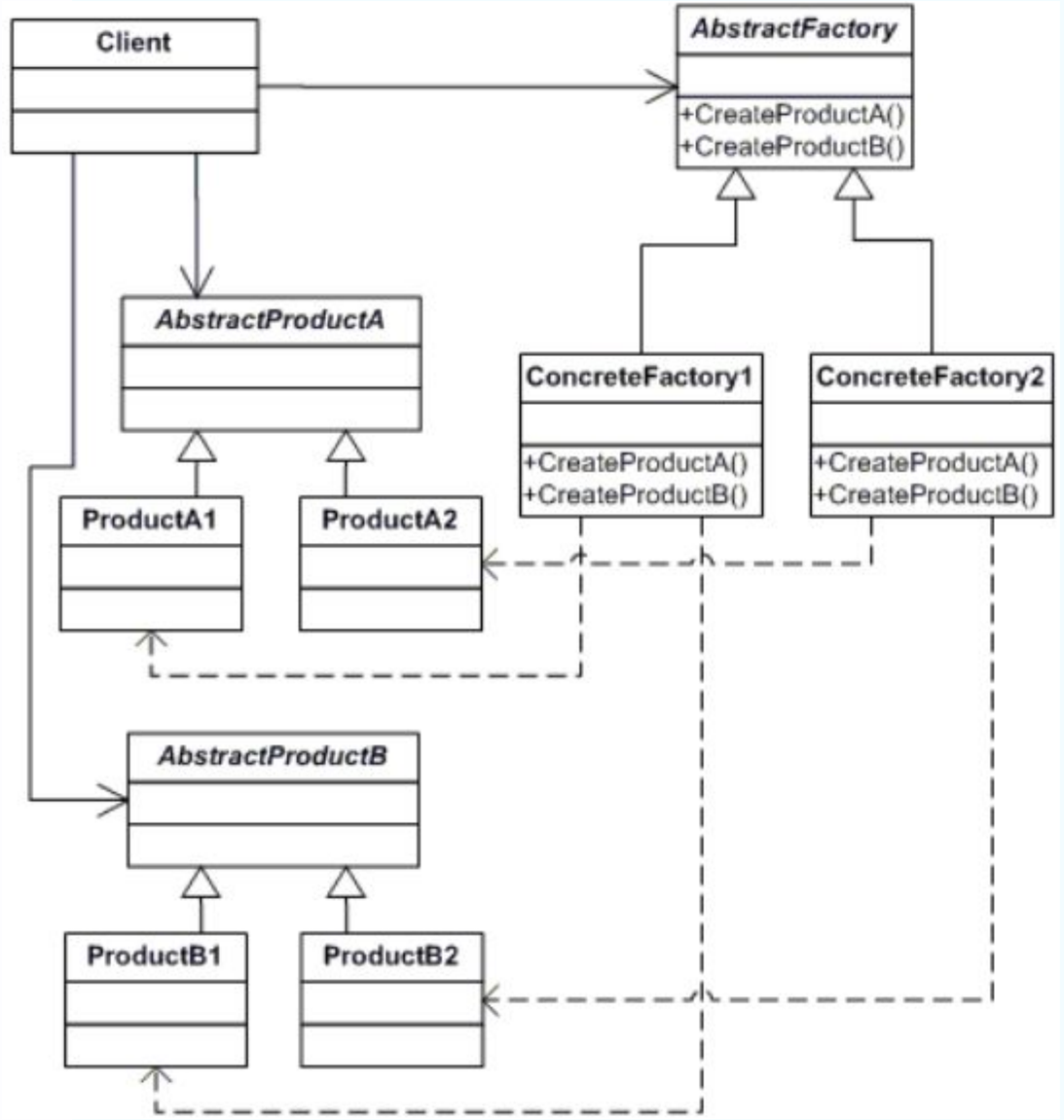
Dependency Injection **(DI)** is a design pattern which implements the IoC principle to invert the creation of dependent objects.

The **IoC container** is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it.

Creational Pattern: -Abstract Factory

The **Abstract Factory** design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Frequency of use:

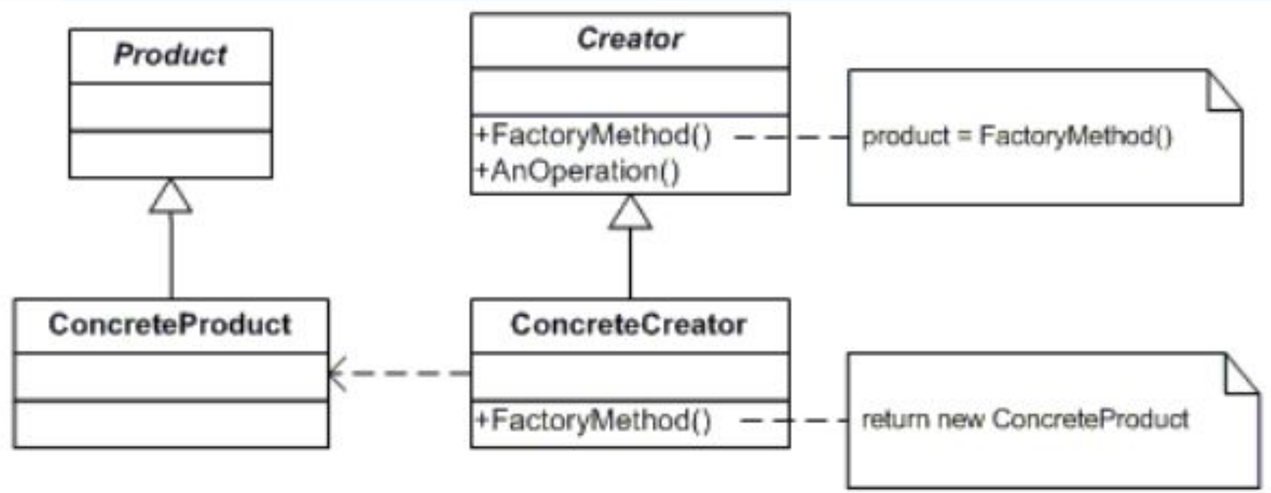


assuresoft
Digital Academy

Creational Pattern: -Factory method

The **Factory Method** design pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses.

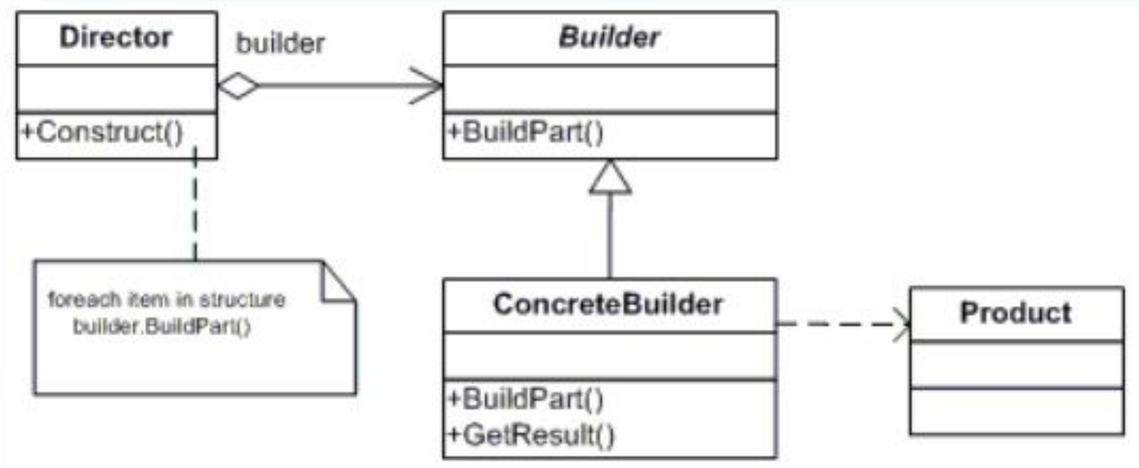
Frequency of use:



assuresoft
Digital Academy

Creational Pattern: -Builder

The **Builder** design pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.



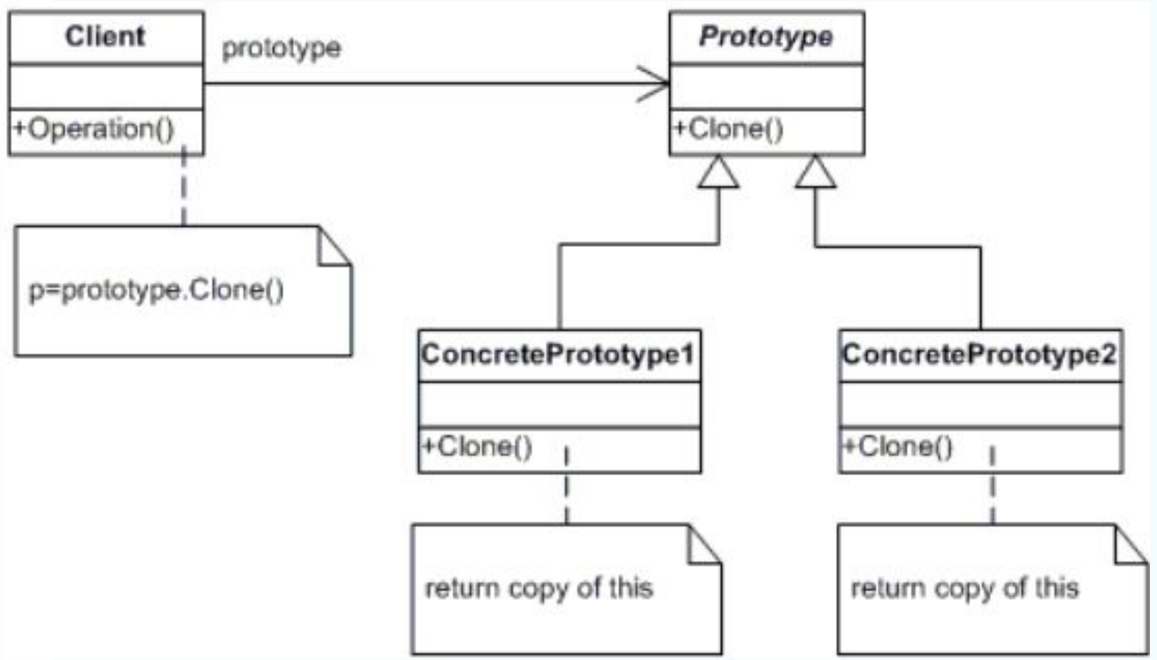
Frequency of use:

assuresoft
Digital Academy

Creational Pattern: -Prototype

The **Prototype** design pattern specifies the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

Frequency of use:



assuresoft
Digital Academy

Creational Pattern: -Singleton

The **Singleton** design pattern ensures a class has only one instance and provide a global point of access to it.

Frequency of use:

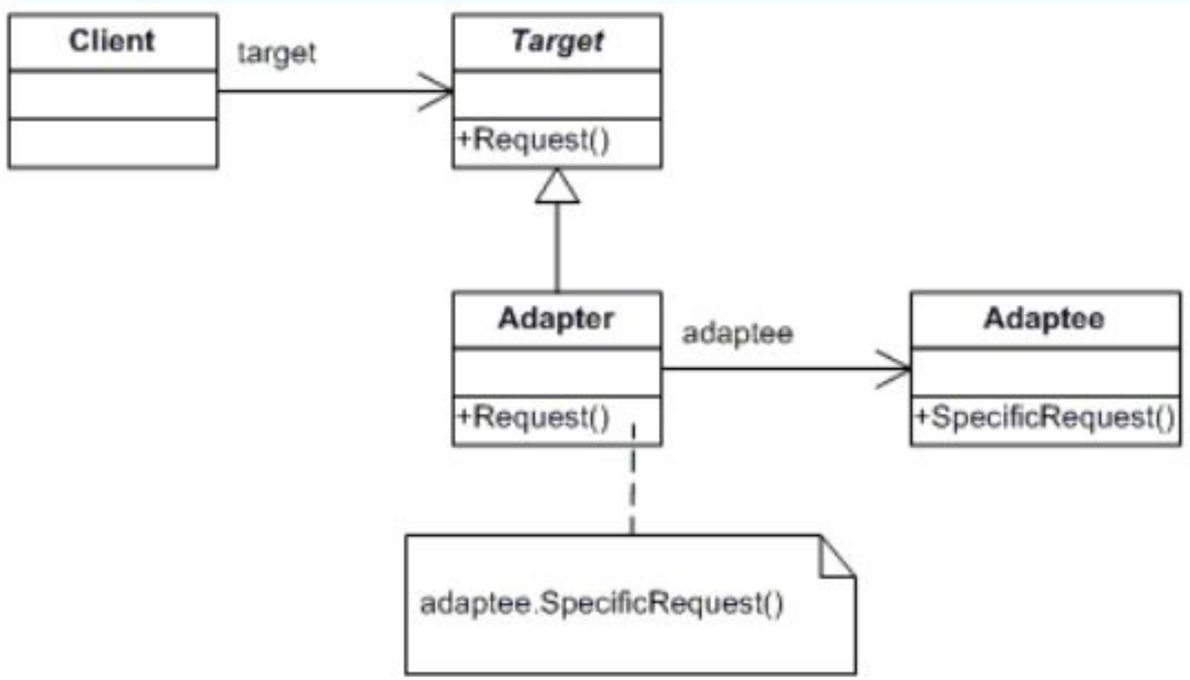
medium-high

assuresoft
Digital Academy

Structural Pattern: -Adapter

The **Adapter** design pattern converts the interface of a class into another interface clients expect. This design pattern lets classes work together that couldn‘t otherwise because of incompatible interfaces.

Frequency of use:

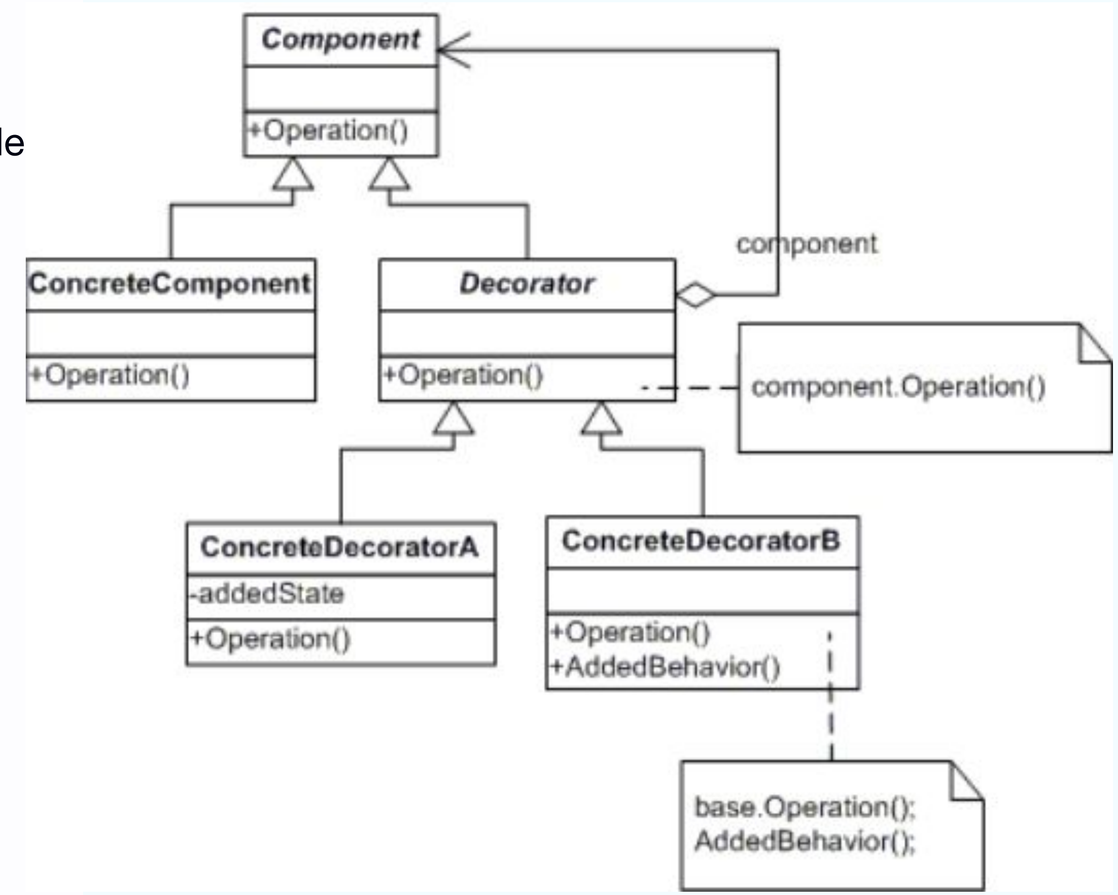


assuresoft
Digital Academy

Structural Pattern: -Decorator

The **Decorator** design pattern attaches additional responsibilities to an object dynamically. This pattern provide a flexible alternative to subclassing for extending functionality.

Frequency of use:

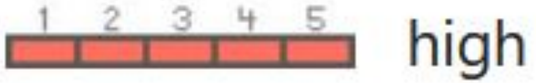


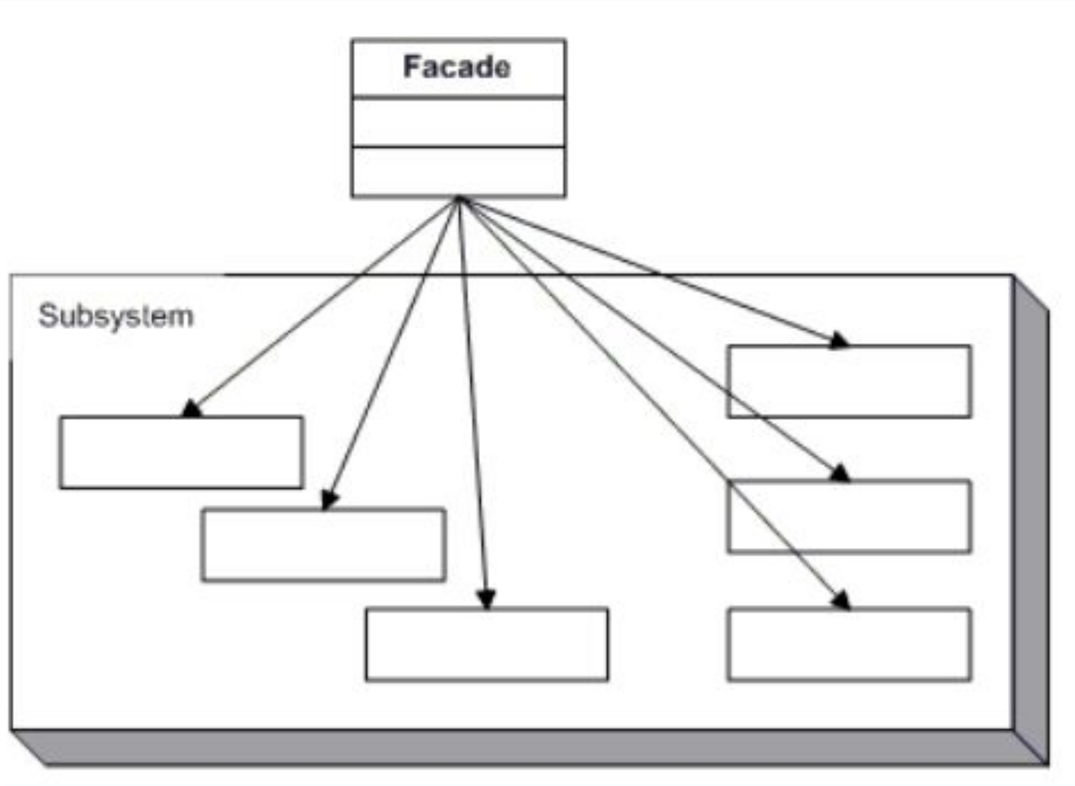
assuresoft
Digital Academy

Structural Pattern: -Facade

The **Facade** design pattern provides a unified interface to a set of interfaces in a subsystem. This pattern defines a higher-level interface that makes the subsystem easier to use.

Frequency of use:



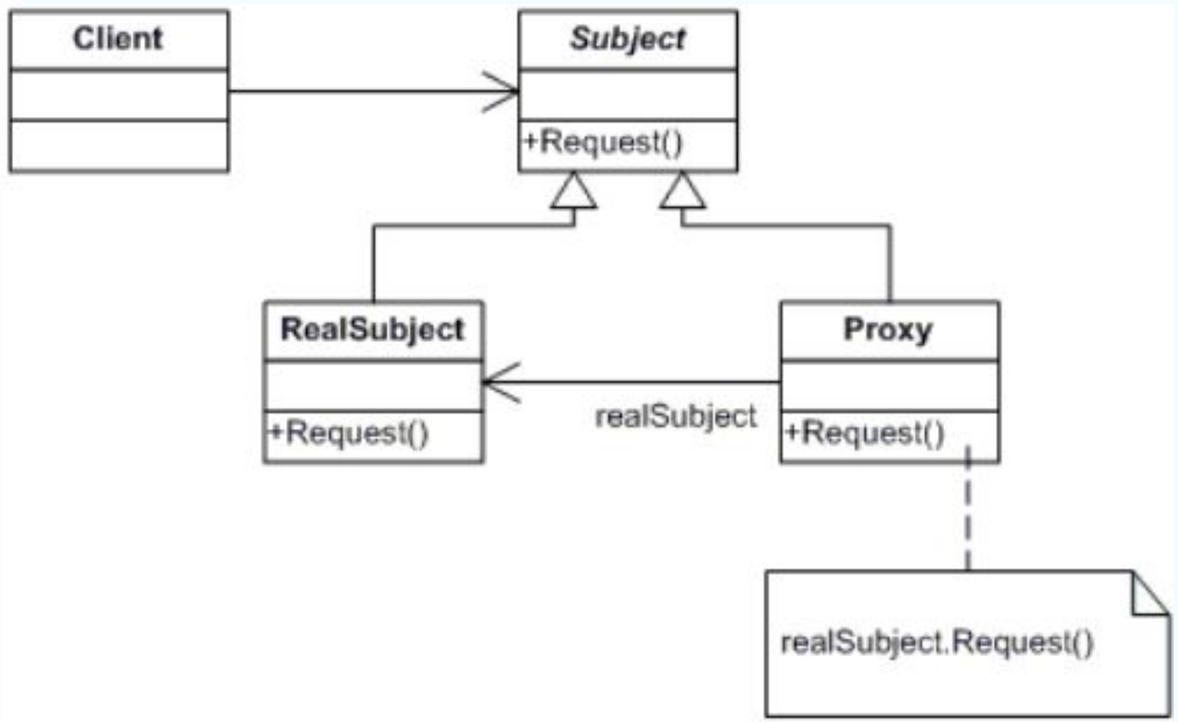


assuresoft
Digital Academy

Structural Pattern: -Proxy

The **Proxy** design pattern provides a surrogate or placeholder for another object to control access to it.

Frequency of use:

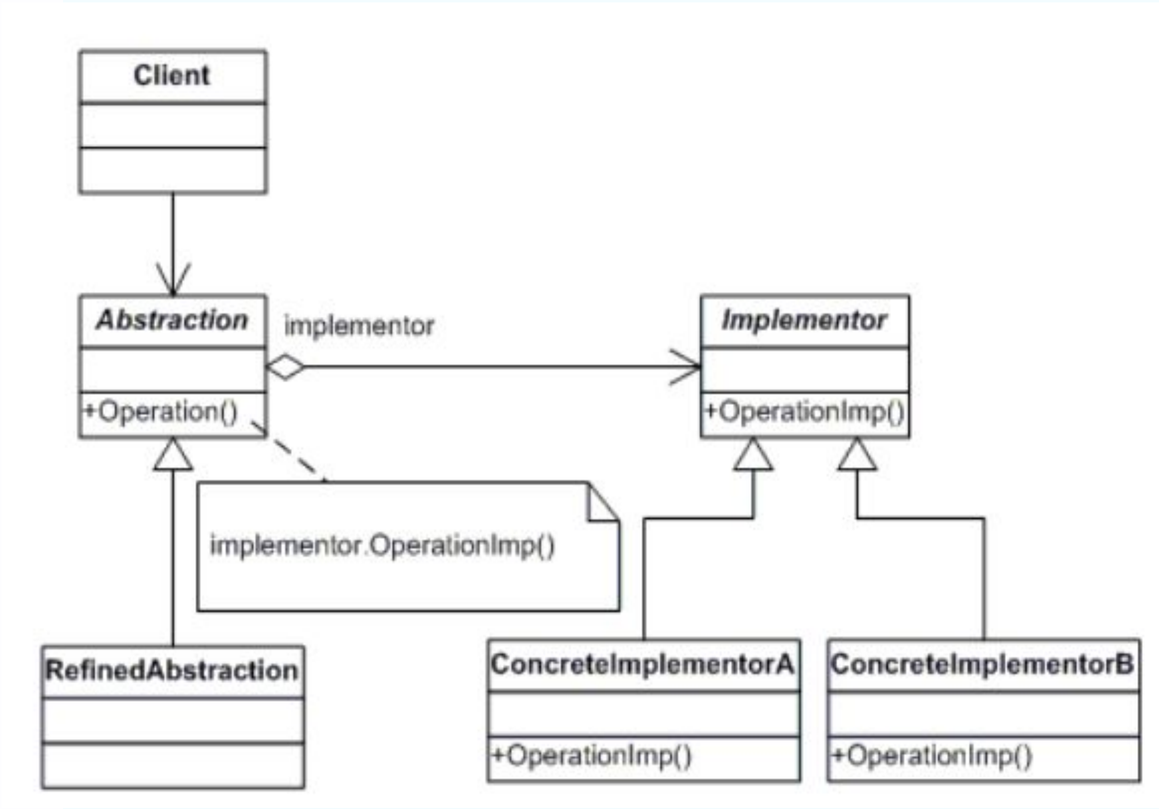


assuresoft
Digital Academy

Structural Pattern: -Bridge

The **Bridge** design pattern decouples an abstraction from its implementation so that the two can vary independently.

Frequency of use:

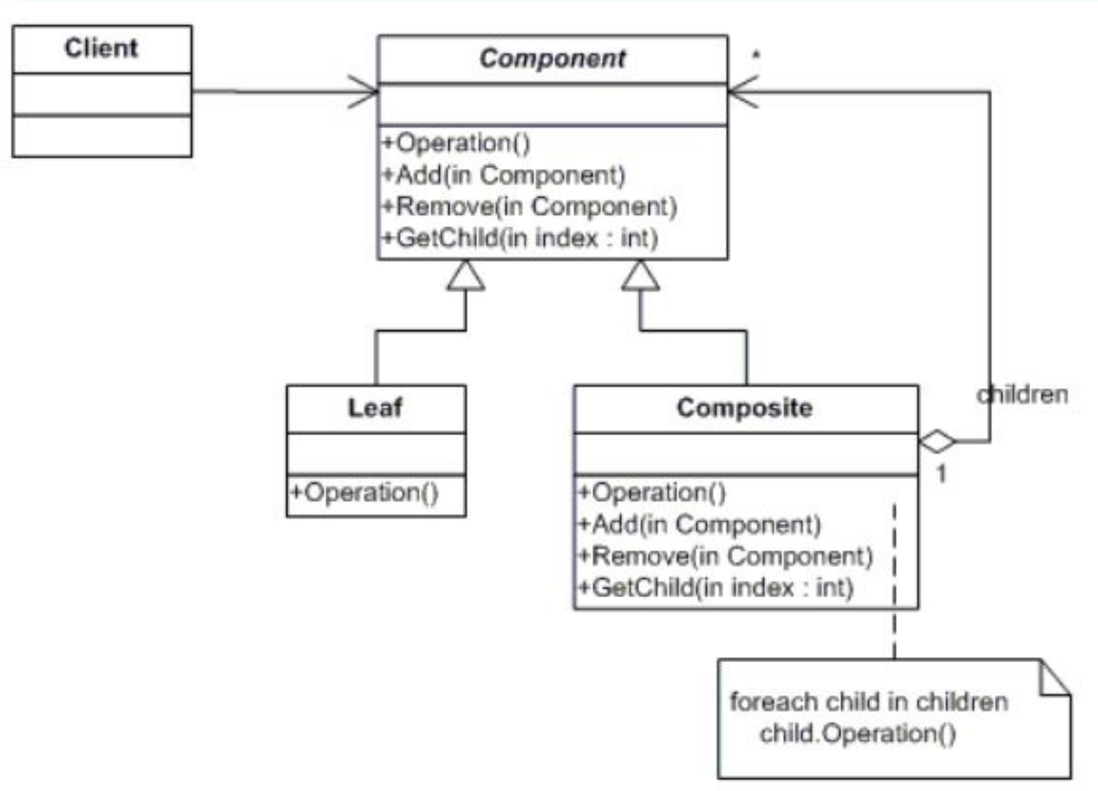


assuresoft
Digital Academy

Structural Pattern: -Composite

The **Composite** design pattern composes objects into tree structures to represent part-whole hierarchies. This pattern lets clients treat individual objects and compositions of objects uniformly.

Frequency of use:

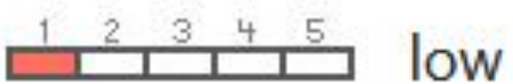


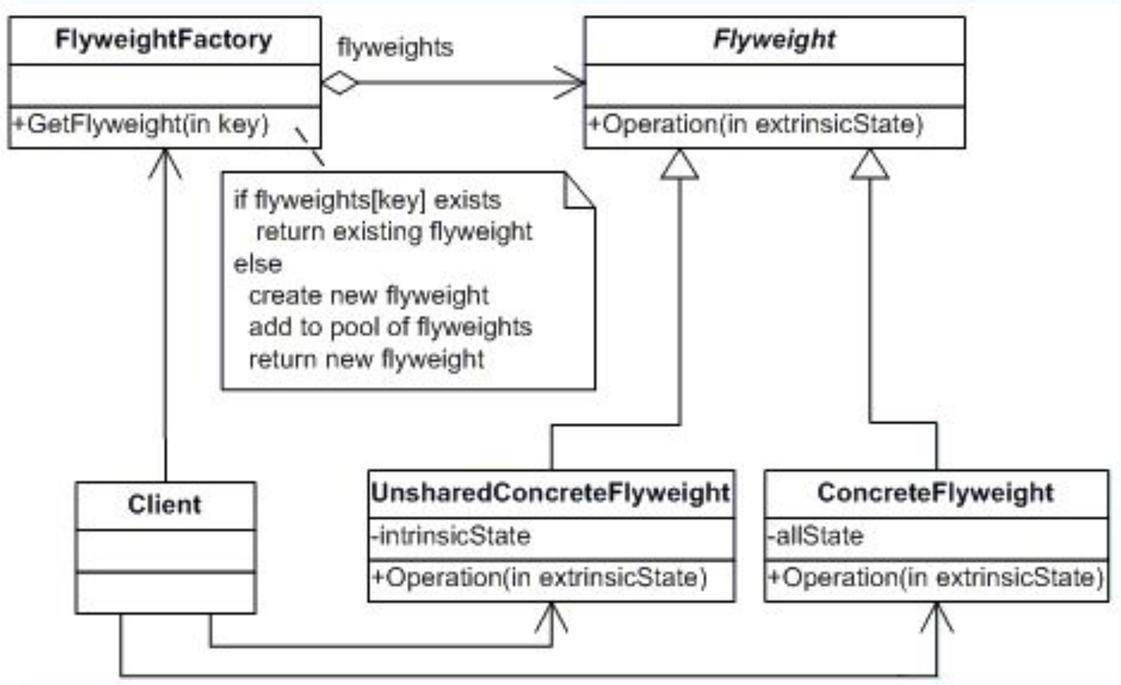
assuresoft
Digital Academy

Structural Pattern: -Flyweight

The **Flyweight** design pattern uses sharing to support large numbers of fine-grained objects efficiently.

Frequency of use:

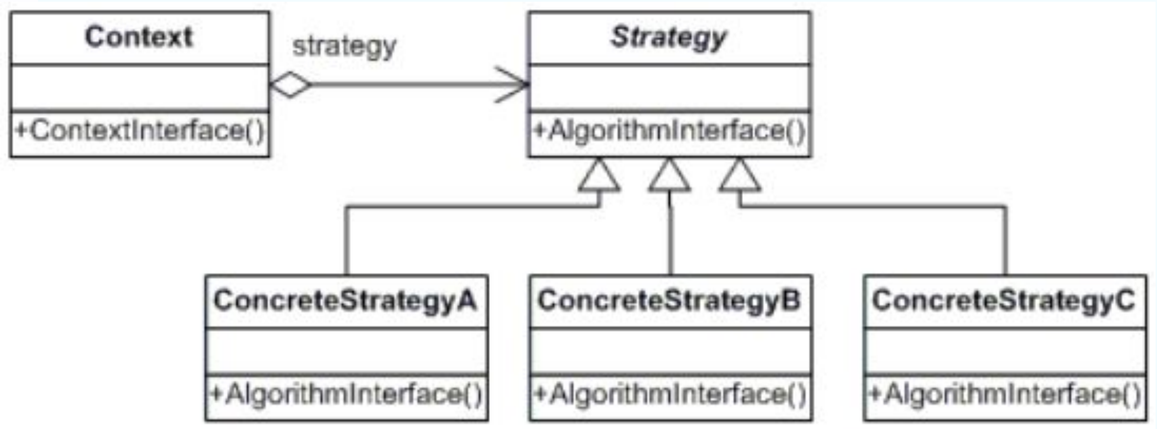




assuresoft
Digital Academy

Behavioral Pattern: -Strategy

The **Strategy** design pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. This pattern lets the algorithm vary independently from clients that use it.

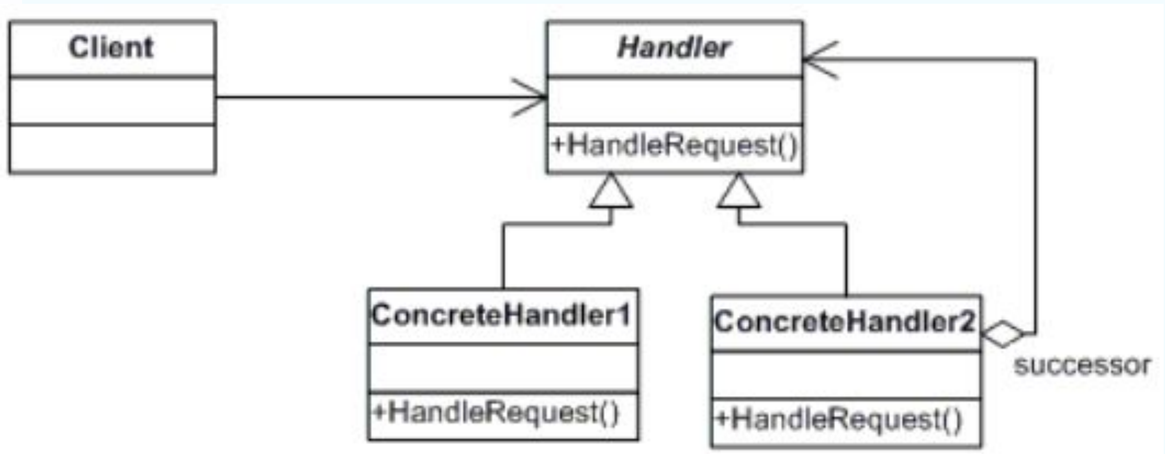


Frequency of use:

assuresoft
Digital Academy

Behavioral Pattern: -Chain of Responsibility

The **Chain of Responsibility** design pattern avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. This pattern chains the receiving objects and passes the request along the chain until an object handles it.



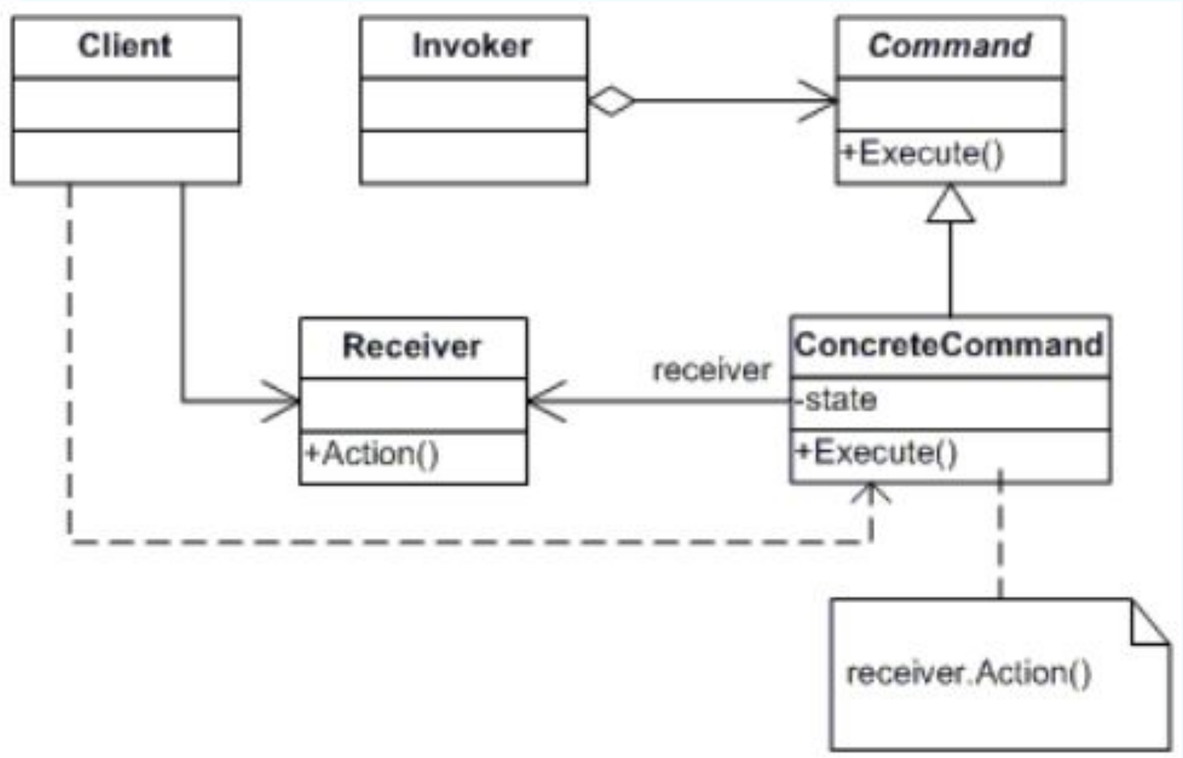
Frequency of use:

assuresoft
Digital Academy

Behavioral Pattern: -Command

The **Command** design pattern encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Frequency of use:

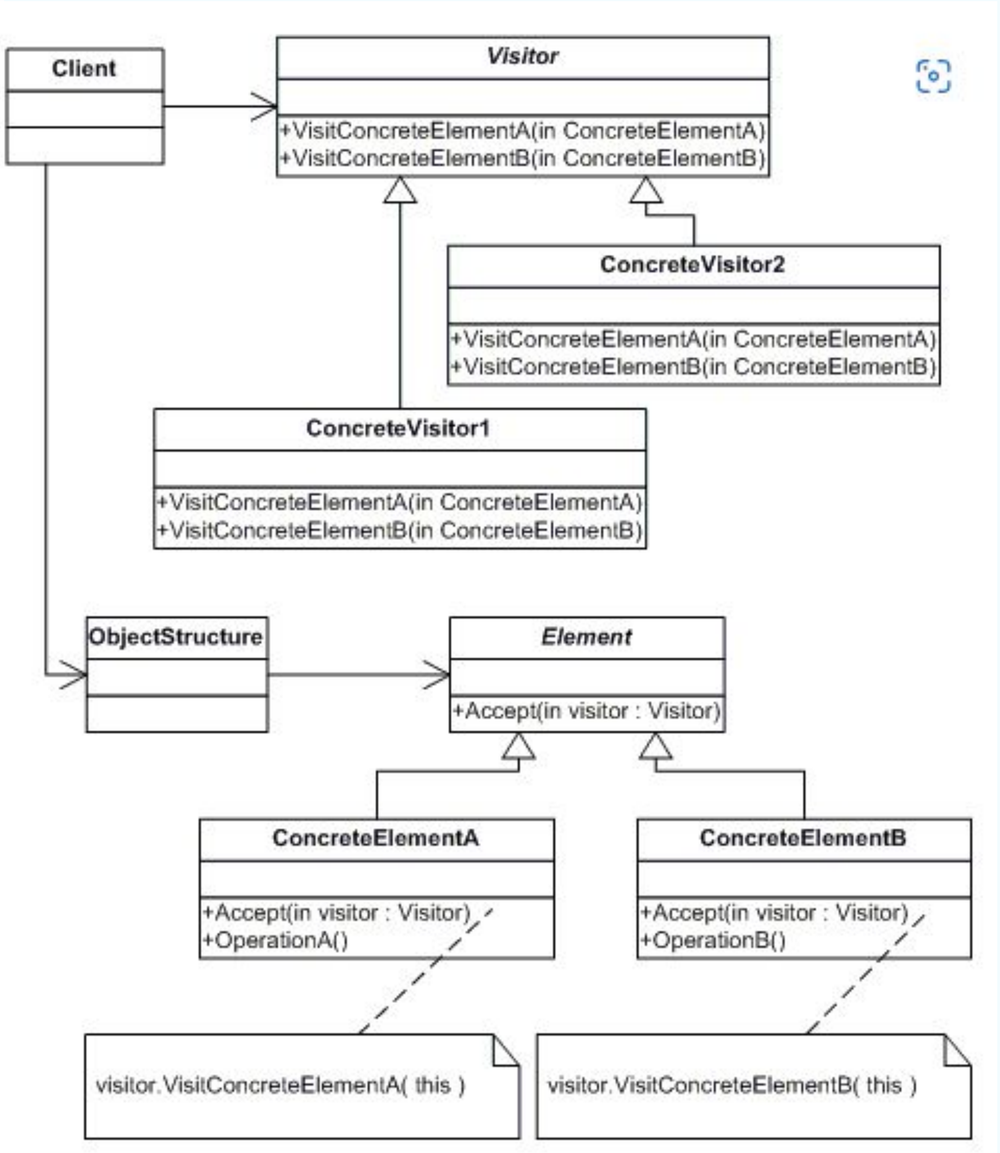


assuresoft
Digital Academy

Behavioral Pattern: -Visitor

The **Visitor** design pattern represents an operation to be performed on the elements of an object structure. This pattern lets you define a new operation without changing the classes of the elements on which it operates.

Frequency of use:

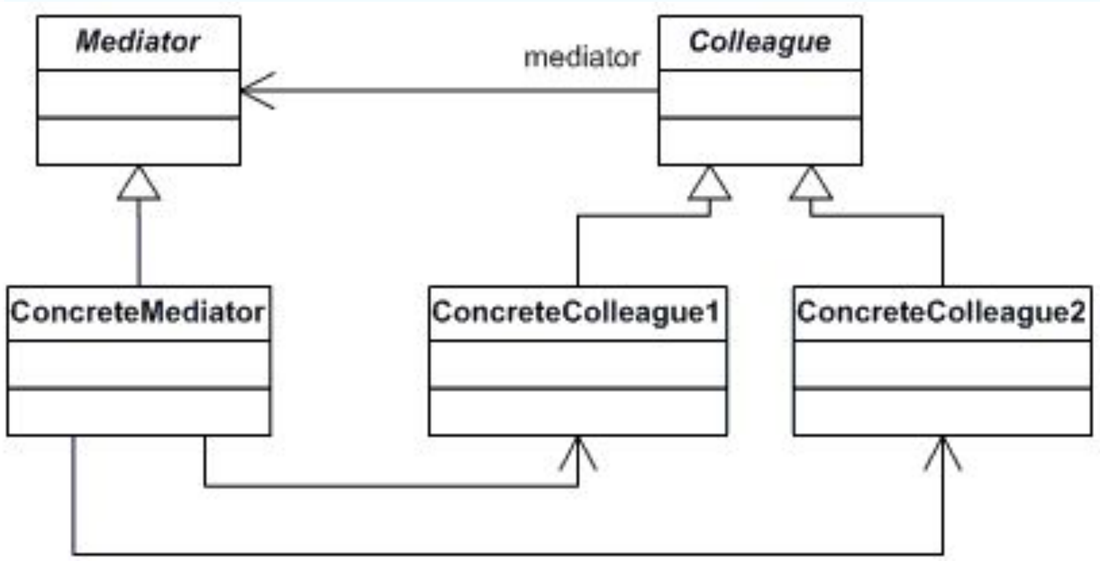


assuresoft
Digital Academy

Behavioral Pattern: -Mediator

The **Mediator** design pattern defines 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.

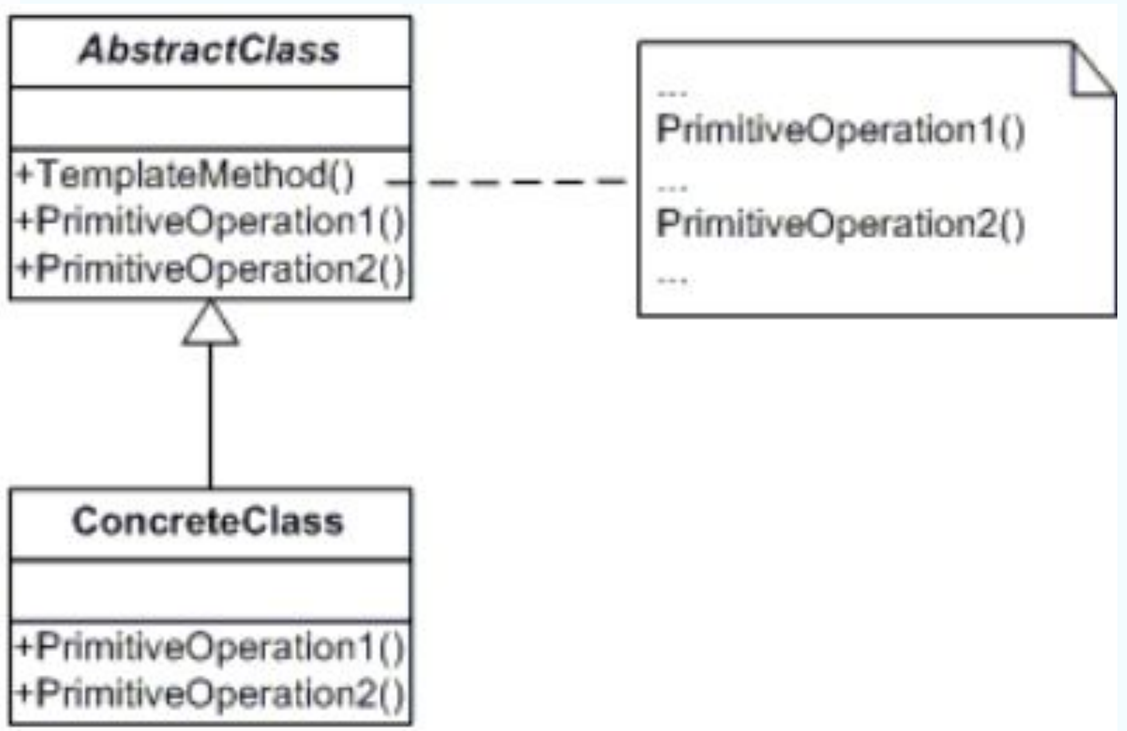
Frequency of use:



assuresoft
Digital Academy

Behavioral Pattern: -Template Method

The **Template Method** design pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. This pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm‘s structure.



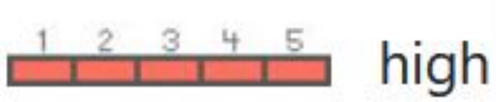
Frequency of use:

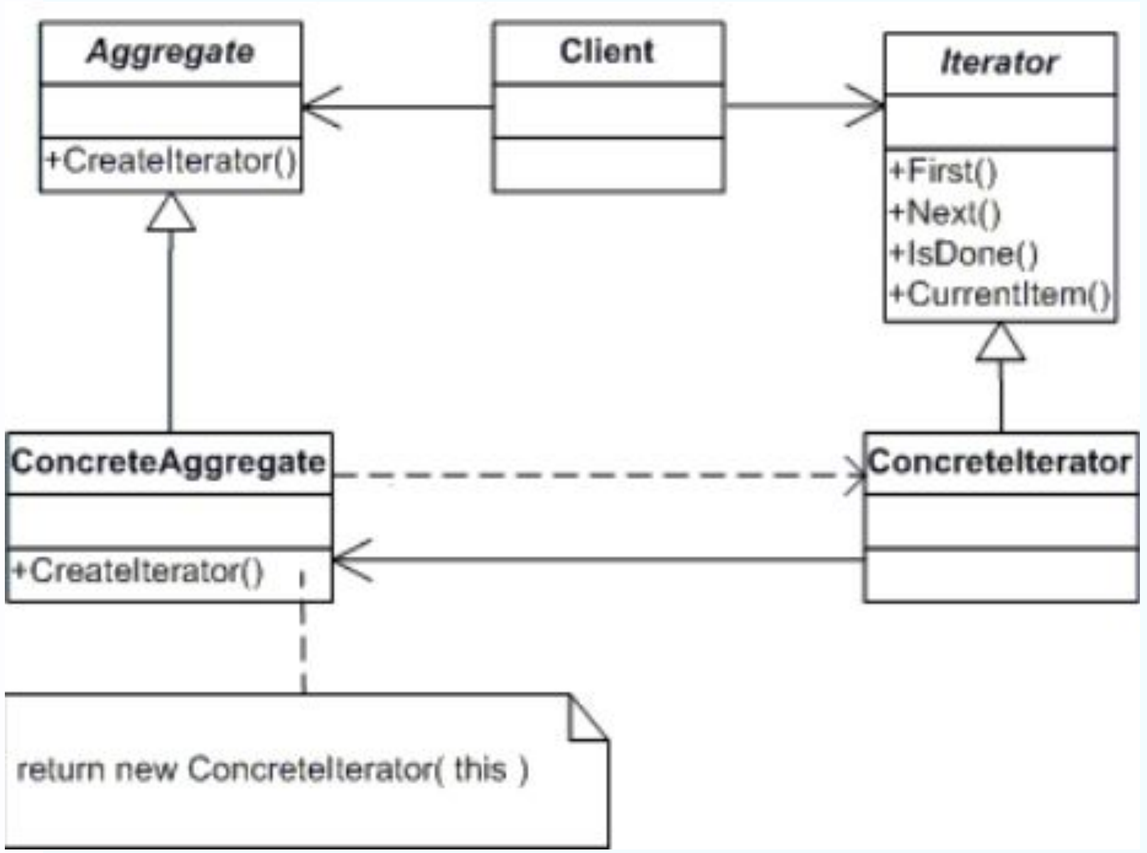
assuresoft
Digital Academy

Behavioral Pattern: -Iterator

The **Iterator** design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Frequency of use:



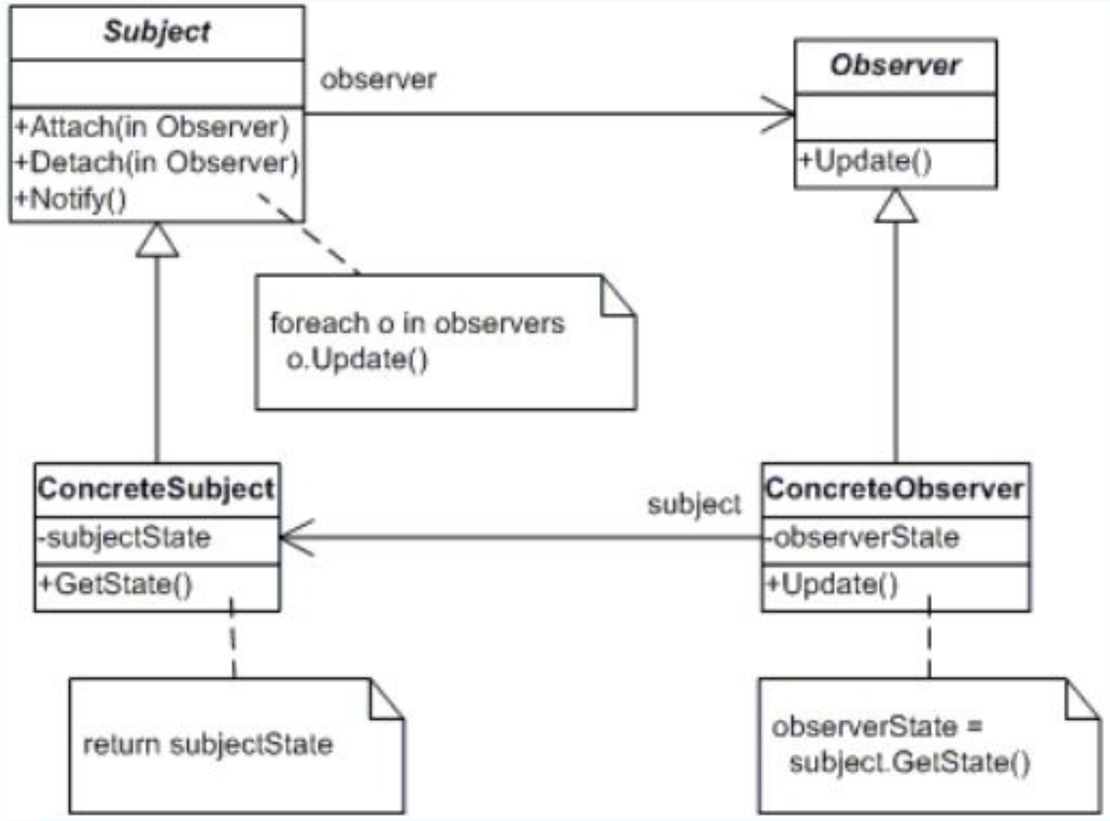


assuresoft
Digital Academy

Behavioral Pattern: -Observer

The **Observer** design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

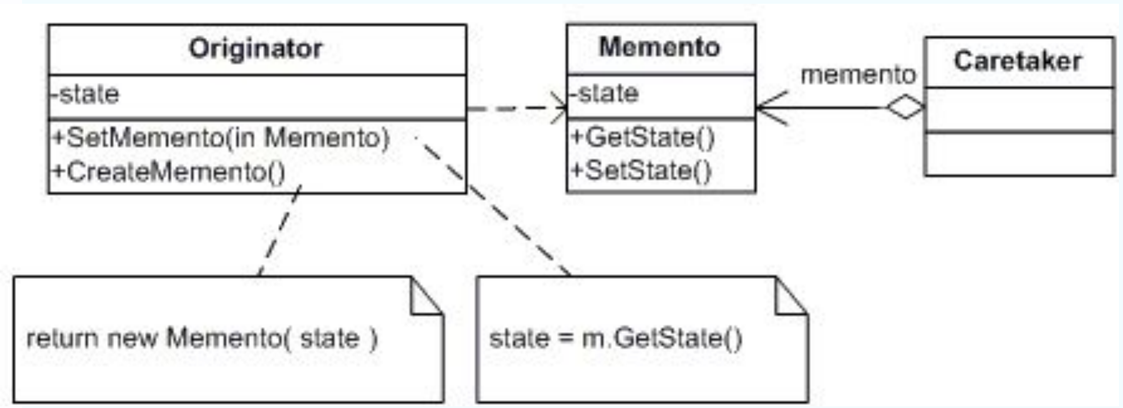
Frequency of use:



assuresoft
Digital Academy

Behavioral Pattern: - Memento

The **Memento** design pattern without violating encapsulation, captures and externalizes an object‘s internal state so that the object can be restored to this state later.



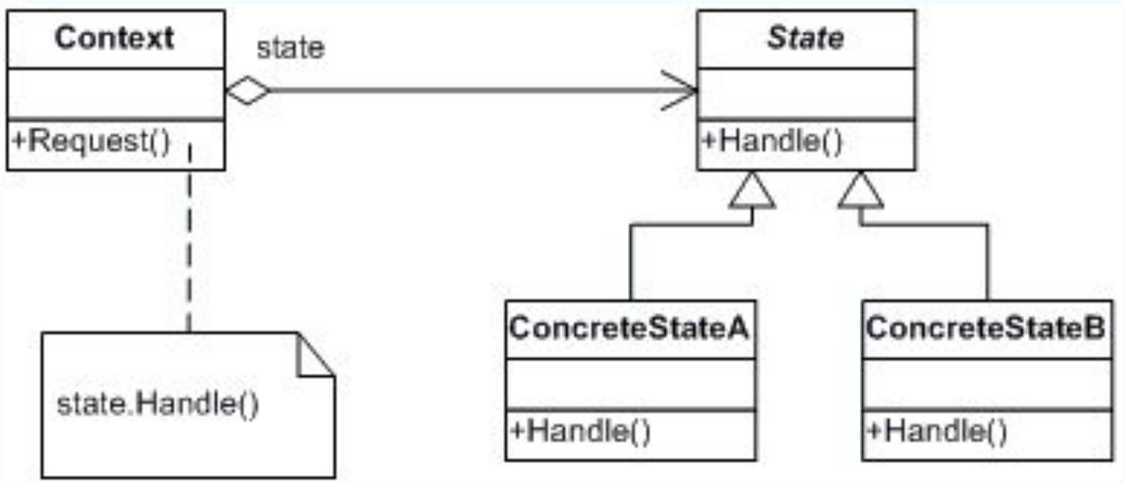
Frequency of use:

assuresoft
Digital Academy

Behavioral Pattern: -State

The **State** design pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Frequency of use:



assuresoft
Digital Academy

AntiPatterns - The Blob, Continuous Obsolescence

- AntiPattern Name: The Blob
- Also Known As: Winnebago and The God Class
- Most Frequent Scale: Application
- Refactored Solution Name: Refactoring of Responsibilities
- Refactored Solution Type: Software
- Root Causes: Sloth, Haste
- Unbalanced Forces: Management of Functionality, Performance, Complexity
- Anecdotal Evidence: "This is the class that is really the heart of our architecture."

Obsolescence Problem
Technology is changing so rapidly that developers have trouble keeping up with the current versions of software and finding combinations of product releases that work together.
Given that every commercial product line evolves through new product releases, the situation has become increasingly difficult for developers to cope with. Finding compatible releases of products that successfully interoperate is even harder.

AntiPatterns - Lava Flow, Functional Decomposition

- AntiPattern Name: Lava Flow
- Also Known As: Dead Code
- Most Frequent Scale: Application
- Refactored Solution Name: Architectural Configuration Management
- Refactored Solution Type: Process
- Root Causes: Avarice, Greed, Sloth
- Unbalanced Forces: Management of Functionality, Performance, Complexity

- AntiPattern Name: Functional Decomposition
- Also Known As: No Object-Oriented AntiPattern "No OO"
- Most Frequent Scale: Application
- Refactored Solution Name: Object-Oriented Reengineering
- Refactored Solution Type: Process
- Root Causes: Avarice, Greed, Sloth
- Unbalanced Forces: Management of Complexity, Change

AntiPatterns - Poltergeists, Golden Hammer

- AntiPattern Name: Poltergeists
- Also Known As: Gypsy , Proliferation of Classes , and Big DoIt Controller Class
- Most Frequent Scale: Application
- Refactored Solution Name: Ghostbusting
- Refactored Solution Type: Process
- Root Causes: Sloth, Ignorance
- Unbalanced Force: Management of Functionality, Complexity
- Anecdotal Evidence: "I'm not exactly sure what this class does, but it sure is important!"

- AntiPattern Name: Golden Hammer
- Also Known As: Old Yeller, Head-in-the sand
- Most Applicable Scale: Application
- Refactored Solution Name: Expand your horizons
- Refactored Solution Type: Process
- Root Causes: Ignorance, Pride, Narrow-Mindedness
- Unbalanced Forces: Management of Technology Transfer
- Anecdotal Evidence: "I have a hammer and everything else is a nail." "Our database is our architecture." "Maybe we shouldn't have used Excel macros for this job after all."

AntiPatterns - Dead End, Spaghetti Code

- A Dead End is reached by modifying a reusable component, if the modified component is no longer maintained and supported by the supplier. When these modifications are made, the support burden transfers to the application system developers and maintainers. Improvements in the reusable component cannot be easily integrated, and support problems may be blamed on the modification.

- AntiPattern Name: Spaghetti Code
- Most Applicable Scale: Application
- Refactored Solution Name: Software Refactoring, Code Cleanup
- Refactored Solution Type: Software
- Root Causes: Ignorance, Sloth
- Unbalanced Forces: Management of Complexity, Change
- Anecdotal Evidence: "Ugh! What a mess!" "You do realize that the language supports more than one function, right?" "It's easier to rewrite this code than to attempt to modify it." "Software engineers don't write spaghetti code." "The quality of your software structure is an investment for future modification and extension."

AntiPatterns - Copy and Paste, Mushroom management

- AntiPattern Name: Cut-and-Paste Programming
- Also Known As: Clipboard Coding, Software Cloning, Software Propagation
- Most Applicable Scale: Application
- Refactored Solution Name: Black Box Reuse
- Refactored Solution Type: Software
- Root Causes: Sloth
- Unbalanced Forces: Management of Resources, Technology Transfer
- Anecdotal Evidence: "Hey, I thought you fixed that bug already, so why is it doing this again?" "Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!"

Mushroom Management

Also known as Pseudo-Analysis and Blind Development, Mushroom Management is often described by this phrase: "Keep your developers in the dark and feed them fertilizer."
An experienced system architect recently stated, "Never let software developers talk to end users." Furthermore, without end-user participation, "The risk is that you end up building the wrong system."

Partial Solution:
Risk-driven development is a spiral development process based upon prototyping and user feedback. Risk-driven development is a specialization of iterative-incremental development process (see the Analysis Paralysis AntiPattern). In this case, every increment is an external iteration.

Source: https://sourcemaking.com/antipatterns

Refactoring

Martin Fowler: “Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.”

We should try to refactor code whenever possible to follow SOLID principles and good practices in general.

- Refactor effort should be lower than value in doing so.
  - If a refactor is going to take several hours and will just order some code, is not worthy.
  - If a refactor will leave us in a better position than before allowing us to perform our tasks then is worthy.

- Refactor risks should be diminished as much as possible.
  - If a refactor is going to be extensive, place unit tests before performing it or harden its related unit tests if has any.
  - Avoid refactoring common resources unless strictly necessary
  - If there are means to add a feature flip (flag) in the portion refactored, do so.
  - Have a fallback plan if something gets broken and you have to rollback changes.

Refactoring
Method extraction

Problem
You have a code fragment that can be grouped together.
Solution
Move this code to a separate new method (or function) and replace the old code with a call to the method.

The more lines found in a method, the harder it’s to figure out what the method does. This is the main reason for this refactoring.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.