OBJECT ORIENTED SOFTWARE DEVELOPMENT

Subash Manandhar

Chapter 3 – Design Patterns
3.1 Introduction to Design Patterns

- **Introduction**:
  - A design pattern is a general repeatable solution to a commonly occurring problem in software design.
  - A design pattern isn't a finished design that can be transformed directly into code.
  - It is a description or template for how to solve a problem that can be used in many different situations.
  - The pattern is not a specific piece of code, but a general concept for solving a particular problem.
  - We can follow the pattern details and implement a solution that suits the realities of our own program.
  - Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems.
  - While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution.

Chapter 3 – Design Patterns
3.1 Introduction to Design Patterns

- **Introduction**:
  - The code of the same pattern applied to two different programs may be different.
  - An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.
  - The sections that are usually present in a pattern description are:
    - **Intent** of the pattern briefly describes both the problem and the solution.
    - **Motivation** further explains the problem and the solution the pattern makes possible.
    - **Structure** of classes shows each part of the pattern and how they are related.
    - **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Chapter 3 – Design Patterns
3.1 Introduction to Design Patterns

- **History**:
  - Patterns originated as an architectural concept by Christopher Alexander (1977/79).
  - In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year.
  - In the following years, Beck, Cunningham and others followed up on this work.
  - Design patterns gained popularity in computer science after the book Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994 by the so-called “Gang of Four” (Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm.).
  - That same year, the first Pattern Languages of Programming Conference was held and the following year, the Portland Pattern Repository was set up for documentation of design patterns.

Chapter 3 – Design Patterns
3.2 Programming paradigm versus Design patterns

- Programming paradigm is a method, a way, a principle of programming.
  - It describes the programming process, which is the way programs are made.
  - It explains core structure of program written in certain paradigm, everything that program consists of and its components.
  - Some of programming paradigms: Procedural paradigm, functional paradigm, object-oriented paradigm, modular programming, and structured paradigm etc…
- According to GoF definition of design patterns:
  - “In software engineering, a software **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.”
- Design patterns are formalized solutions to common programming problems. They mostly refer to object oriented programming, but some of solutions can be applied in various paradigms.

Chapter 3 – Design Patterns
3.3 Importance of Design Pattern

- Design patterns are a toolkit of **tried and tested solutions** to common problems in software design.
  - Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.
- Design patterns define a common language that you and your teammates can use to **communicate** more efficiently.
  - You can say, "Oh, just use a Singleton for that," and everyone will understand the idea behind your suggestion.
  - No need to explain what a singleton is if you know the pattern and its name.
- Design patterns can **speed up** the development process by providing tested, proven development paradigms.
  - Effective software design requires considering issues that may not become visible until later in the implementation.
  - Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.
- .

Chapter 3 – Design Patterns
3.3 Importance of Design Pattern

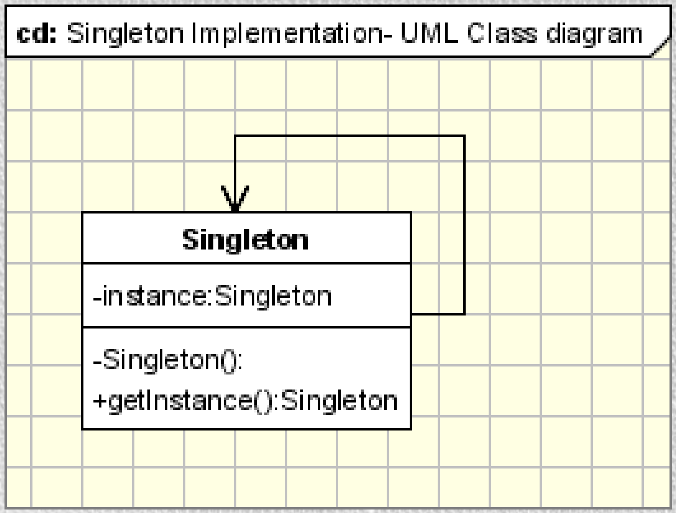
- Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems.
  - Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
- In addition, patterns allow developers to communicate using well-known, well understood names for software interactions.
- Design pattern provides a general reusable solution for the common problems occurs in software design.
  - The patterns typically show relationships and interactions between classes or objects.
  - The idea is to speed up the development process by providing well tested, proven development/design paradigm.
- Design patterns are not meant for project development. Design patterns are meant for the common problem solving.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Creational Pattern**
  - provide object creation mechanisms that increase flexibility and reuse of existing code.
  - e.g. **Singleton Pattern** : *A class of which only a single instance can exist*
- **Structural Pattern**
  - explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
  - e.g. **Adapter Pattern** : *Match interfaces of different classes*
- **Behavioral Pattern**
  - take care of effective communication and the assignment of responsibilities between objects.
  - e.g. **Chain of Responsibility** : *A way of passing a request between a chain of objects*

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **1 Creational Pattern**
  - **1.1 Singleton Pattern**
    - It is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
    - The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients.
getInstance() is responsible for creating its class unique instance in case it is not created yet and to return that instance.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

```
public class SingletonClass {

        private static SingletonClass instance = new SingletonClass();

        private SingletonClass() {}

        public static SingletonClass getInstance() {
                return instance;
        }

        public void showMessage() {
                System.out.println("I'm a singleton object!");
        }
}
```

Here
This class is creating a static object of itself, which represents the global instance.
By providing a private constructor, the class cannot be instantiated.
A static method getInstance() is used as a global access point for the rest of the application.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

```
public class Main {
        public static void main(String[] args) {
                SingletonClass singletonClass = SingletonClass.getInstance();
                singletonClass.showMessage();
        }
}
```

Output: I'm a singleton object!

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

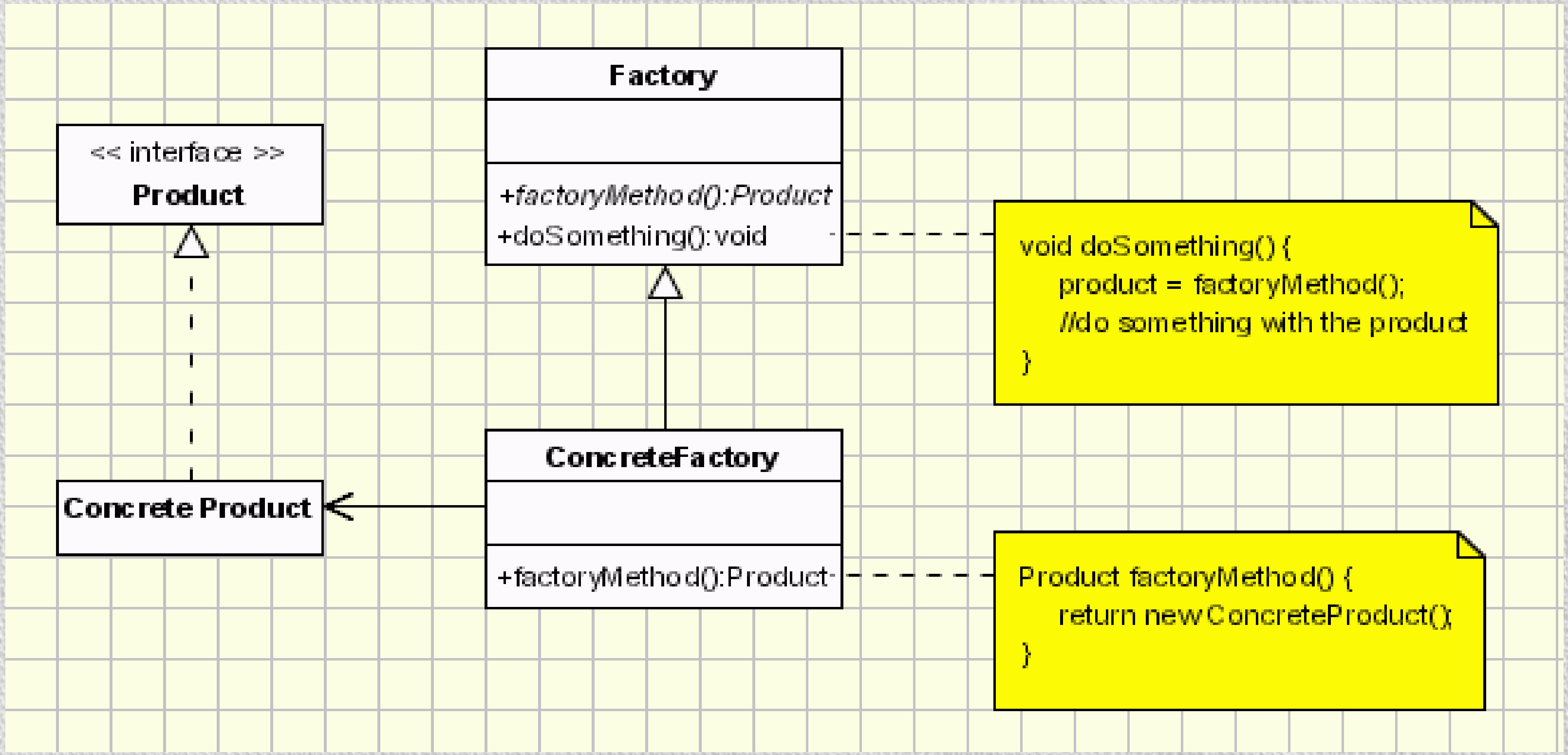
- **Applicability & Examples**
  - **Example 1 - Logger Classes**
    - The Singleton pattern is used in the design of logger classes.
    - This classes are usualy implemented as a singletons, and provides a global logging access point in all the application components without being necessary to create an object each time a logging operations is performed.
  - **Example 2 - Configuration Classes**
    - The Singleton pattern is used to design the classes which provides the configuration settings for an application.
    - By implementing configuration classes as Singleton not only that we provide a global access point, but we also keep the instance we use as a cache object.
    - When the class is instantiated(or when a value is read) the singleton will keep the values in its internal structure.
    - If the values are read from the database or from files this avoids the reloading the values each time the configuration parameters are used.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Example 3 - Accessing resources in shared mode**
  - It can be used in the design of an application that needs to work with the serial port.
  - Let's say that there are many classes in the application, working in an multi-threading environment, which needs to operate actions on the serial port.
  - In this case a singleton with synchronized methods could be used to be used to manage all the operations on the serial port.
- **Example 4 - Factories implemented as Singletons**
  - Let's assume that we design an application with a factory to generate new objects(Account, Customer, Site, Address objects) with their ids, in an multithreading environment.
  - If the factory is instantiated twice in 2 different threads then is possible to have 2 overlapping ids for 2 different objects.
  - If we implement the Factory as a singleton we avoid this problem. Combining Abstract Factory or Factory Method and Singleton design patterns is a common practice.
-

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **1.2 Factory Method:**
  - It is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

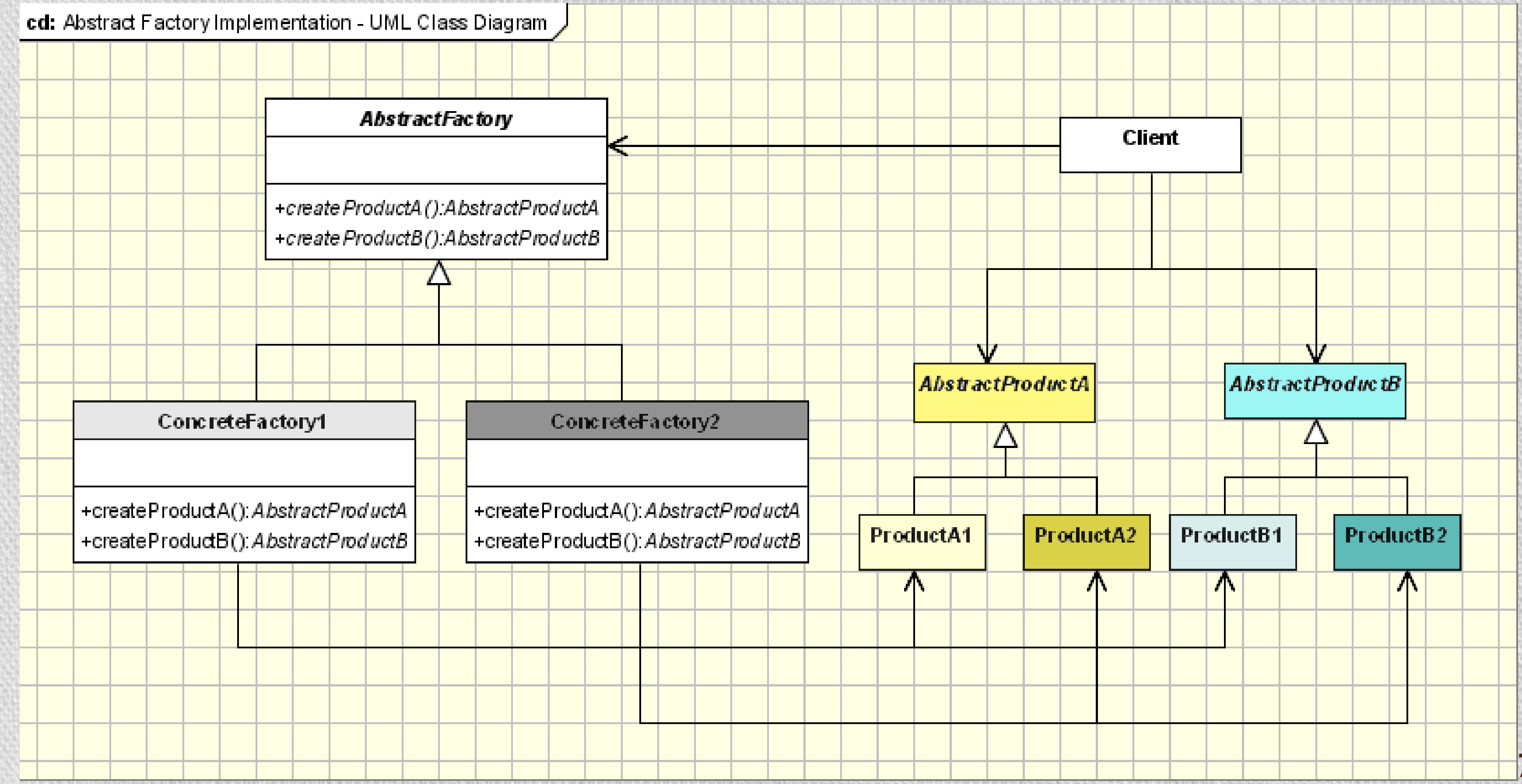
- **Factory Method:**
  - The participants classes in this pattern are:
  - **Product** defines the interface for objects the factory method creates.
  - **ConcreteProduct** implements the Product interface.
  - **Creator**(also refered as **Factory** because it creates the Product objects) declares the method **FactoryMethod**, which returns a Product object. May call the generating method for creating Product objects
  - **ConcreteCreator** overrides the generating method for creating ConcreteProduct objects
  - All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent.
  - The Creator class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteCreator.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Factory Method:**
  - The need for implementing the Factory Method is very frequent. The cases are the ones below:
    - when a class can't anticipate the type of the objects it is supposed to create
    - when a class wants its subclasses to be the ones to specific the type of a newly created object

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **1.3 Abstract Factory Method:**
  - It offers the interface for creating a family of related objects, without explicitly specifying their classes.



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Abstract Factory Method:**
  - The classes that participate to the Abstract Factory pattern are:
  - **AbstractFactory** - declares a interface for operations that create abstract products.
  - **ConcreteFactory** - implements operations to create concrete products.
  - **AbstractProduct** - declares an interface for a type of product objects.
  - **Product** - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
  - **Client** - uses the interfaces declared by the AbstractFactory and AbstractProduct classes.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

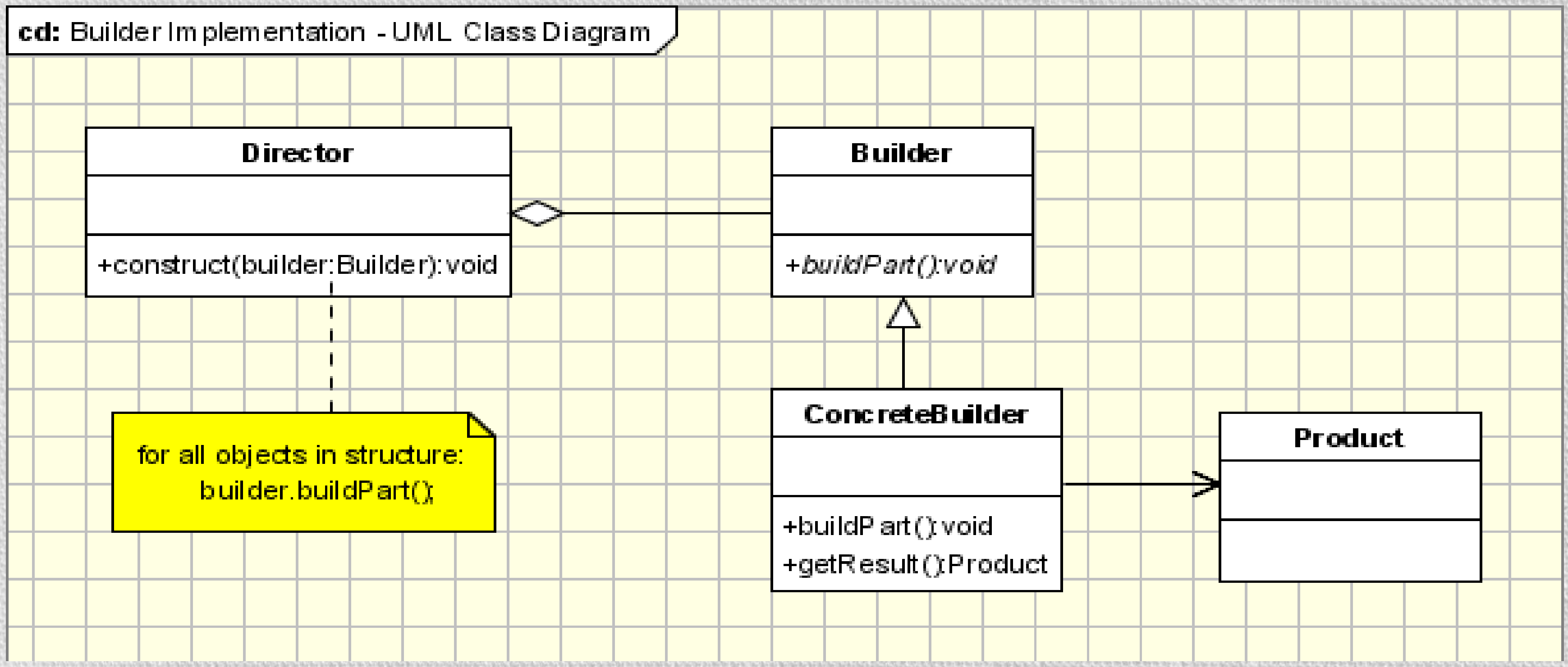
- **Abstract Factory Method:**
  - The AbstractFactory class is the one that determines the actual type of the concrete object and creates it, but it returns an abstract pointer to the concrete object just created.
  - This determines the behavior of the client that asks the factory to create an object of a certain abstract type and to return the abstract pointer to it, keeping the client from knowing anything about the actual creation of the object.
  - The fact that the factory returns an abstract pointer to the created object means that the client doesn't have knowledge of the object's type.
  - This implies that there is no need for including any class declarations relating to the concrete type, the client dealing at all times with the abstract type.
  - The objects of the concrete type, created by the factory, are accessed by the client only through the abstract interface.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Abstract Factory Method:**
  - The second implication of this way of creating objects is that when the adding new concrete types is needed, all we have to do is modify the client code and make it use a different factory, which is far easier than instantiating a new type, which requires changing the code wherever a new object is created.
  - **Applicability**
  - We should use the Abstract Factory design pattern when:
    - the system needs to be independent from the way the products it works with are created.
    - the system is or should be configured to work with multiple families of products.
    - a family of products is designed to work only all together.
    - the creation of a library of products is needed, for which is relevant only the interface, not the implementation, too.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **1.4 Builder Pattern:**
  - It is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Builder Pattern:**
- The participants classes in this pattern are:
  - The **Builder** class specifies an abstract interface for creating parts of a Product object.
  - The **ConcreteBuilder** constructs and puts together parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for saving the product.
  - The **Director** class constructs the complex object using the Builder interface.
  - The **Product** represents the complex object that is being built.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

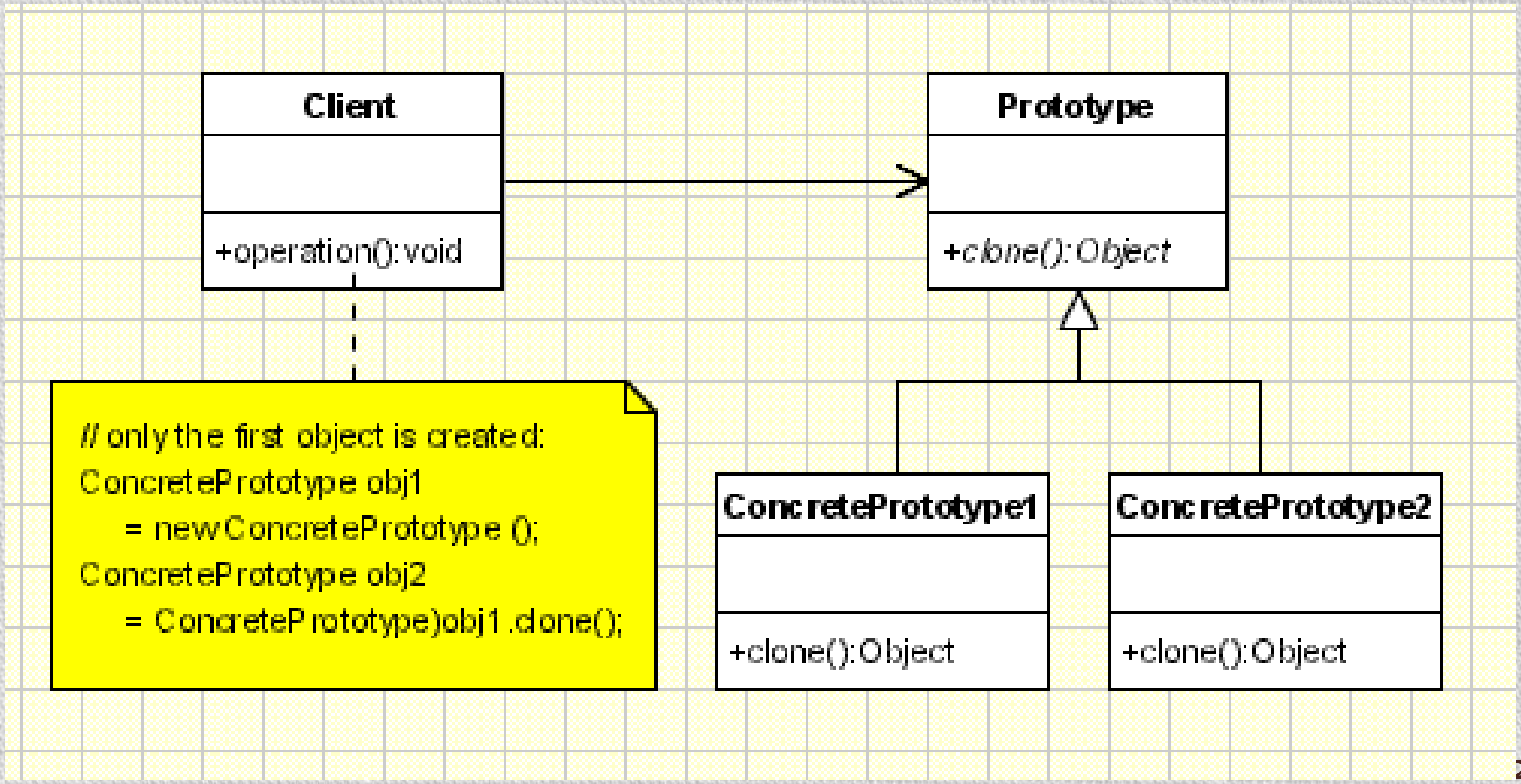
- **Builder Pattern:**
- The client, that may be either another object or the actual client that calls the main() method of the application, initiates the Builder and Director class.
- The Builder represents the complex object that needs to be built in terms of simpler objects and types.
- The constructor in the Director class receives a Builder object as a parameter from the Client and is responsible for calling the appropriate methods of the Builder class.
- In order to provide the Client with an interface for all concrete Builders, the Builder class should be an abstract one.
- This way you can add new types of complex objects by only defining the structure and reusing the logic for the actual construction process.
- The Client is the only one that needs to know about the new types, the Director needing to know which methods of the Builder to call.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Builder Pattern:**
- **Applicability**
- Builder Pattern is used when:
  - the creation algorithm of a complex object is independent from the parts that actually compose the object
  - the system needs to allow different representations for the objects that are being built

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **1.5 Prototype Pattern:**
- It is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **Prototype Pattern:**
- The classes participating to the Prototype Pattern are:
  - **Client** - creates a new object by asking a prototype to clone itself.
  - **Prototype** - declares an interface for cloning itself.
  - **ConcretePrototype** - implements the operation for cloning itself.
- The process of cloning starts with an initialized and instantiated class.
- The Client asks for a new object of that type and sends the request to the Prototype class.
- A ConcretePrototype, depending of the type of object is needed, will handle the cloning through the Clone() method, making a new instance of itself.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

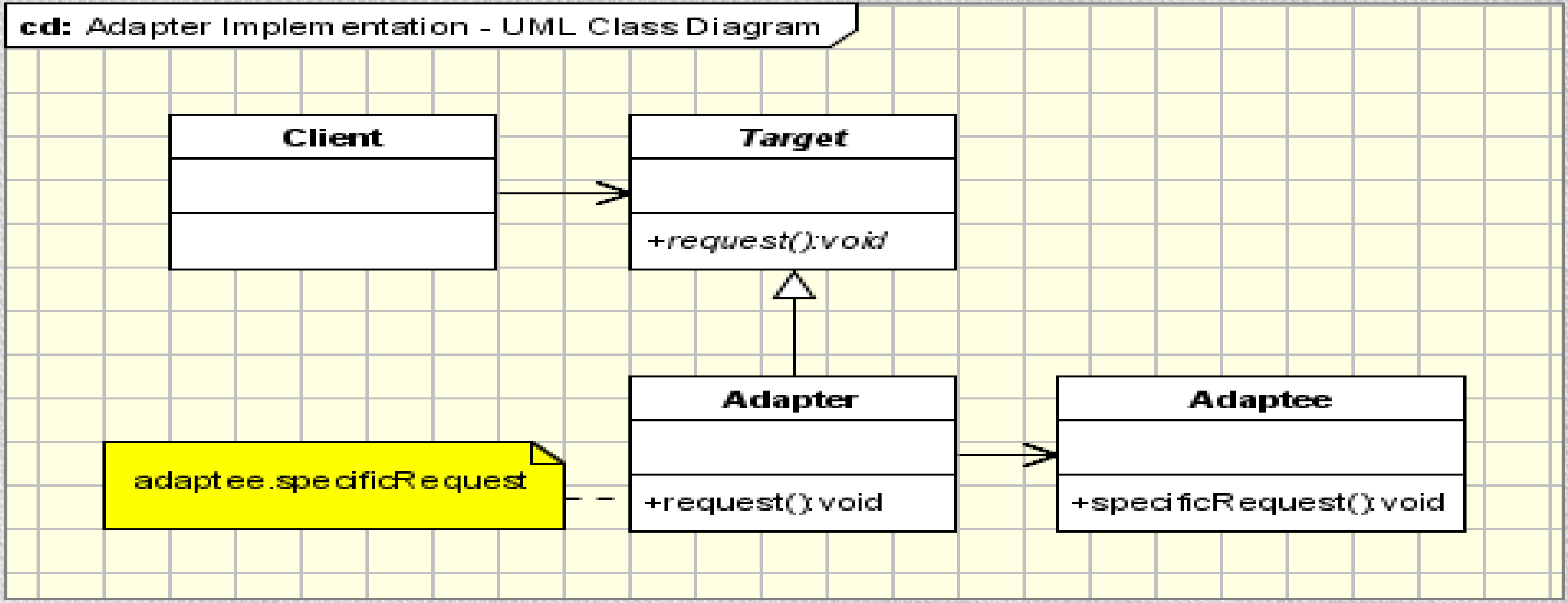
- **Prototype Pattern:**
- You should use a Prototype when:
  - You need to remove overhead for initializing your object.
  - You need multiple objects, from a similar object that already exists.
  - You need to ‘save’ an object at a certain state, then add/change parameters on the new object.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2 Structural Pattern**
- concerned with how classes and objects can be composed, to form larger structures.
- simplifies the structure by identifying the relationships.
- focus on, how the classes inherit from each other and how they are composed from other classes.
- **2.1 Adapter Pattern:**
- converts the interface of a class into another interface that a client wants i.e. to provide the interface according to client requirement while using the services of a class with a different interface.
- **Advantages**
  - It allows two or more previously incompatible objects to interact.
  - It allows reusability of existing functionality.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.1 Adapter Pattern:**
- **Uses**
  - When an object needs to utilize an existing class with an incompatible interface.
  - When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.1 Adapter Pattern:**
- The classes/objects participating in adapter pattern:
  - **Target** - defines the domain-specific interface that Client uses.
  - **Adapter** - adapts the interface Adaptee to the Target interface.
  - **Adaptee** - defines an existing interface that needs adapting.
  - **Client** - collaborates with objects conforming to the Target interface.

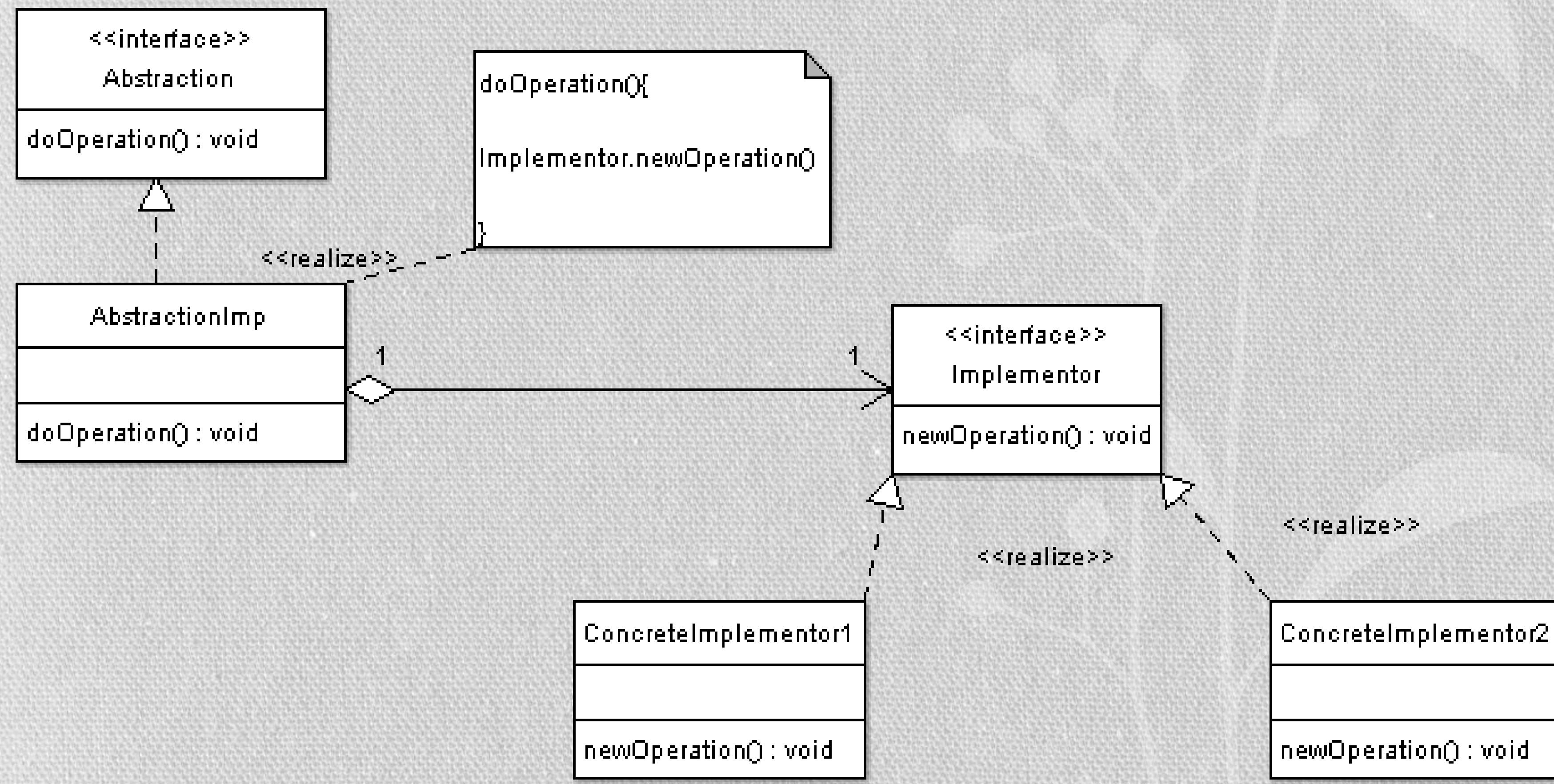
- **2.2 Bridge Pattern**
- decouple the functional abstraction from the implementation so that the two can vary independently.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.2 Bridge Pattern**
- **Advantages**
  - It enables the separation of implementation from the interface.
  - It improves the extensibility.
  - It allows the hiding of implementation details from the client.
- **Uses**
  - When you don't want a permanent binding between the functional abstraction and its implementation.
  - When both the functional abstraction and its implementation need to extended using sub-classes.
  - It is mostly used in those places where changes are made in the implementation does not affect the clients.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.2 Bridge Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.2 Bridge Pattern**
- The participants classes in the bridge pattern are:
  - **Abstraction** - Abstraction defines abstraction interface.
  - **AbstractionImpl** - Implements the abstraction interface using a reference to an object of type Implementor.
  - **Implementor** - Implementor defines the interface for implementation classes. This interface does not need to correspond directly to abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementor interface.
  - **ConcreteImplementor1, ConcreteImplementor2** - Implements the Implementor interface.
- An Abstraction can be implemented by an abstraction implementation, and this implementation does not depend on any concrete implementers of the Implementor interface.
- Extending the abstraction does not affect the Implementor. Also extending the Implementor has no effect on the Abstraction.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.3 Composite Pattern**
  - allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects
  - The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.

Advantages

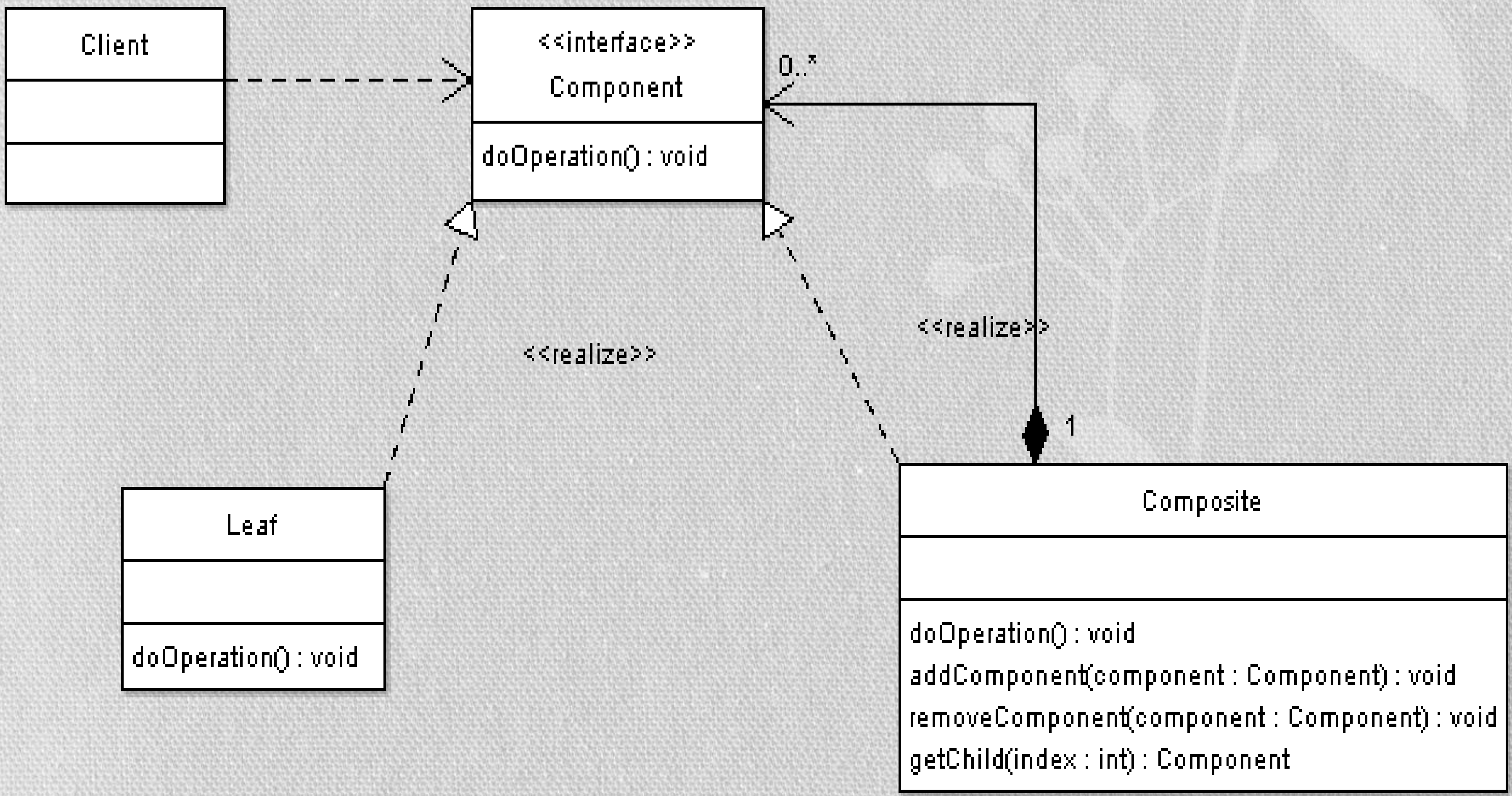
- It defines class hierarchies that contain primitive and complex objects.
- It makes easier to you to add new kinds of components.
- It provides flexibility of structure with manageable class or interface.

Uses

- When you want to represent a full or partial hierarchy of objects.
- When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.3 Composite Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.3 Composite Pattern**
- Elements of composite patterns:
  - **Component** - Component is the abstraction for leafs and composites. It defines the interface that must be implemented by the objects in the composition. For example a file system resource defines move, copy, rename, and getSize methods for files and folders.
  - **Leaf** - Leafs are objects that have no children. They implement services described by the Component interface. For example a file object implements move, copy, rename, as well as getSize methods which are related to the Component interface.
  - **Composite** - A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components. In addition composites provide additional methods for adding, removing, as well as getting components.
  - **Client** - The client manipulates objects in the hierarchy using the component interface.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.3 Composite Pattern**
- A client has a reference to a tree data structure and needs to perform operations on all nodes independent of the fact that a node might be a branch or a leaf.
- The client simply obtains reference to the required node using the component interface, and deals with the node using this interface; it doesn’t matter if the node is a composite or a leaf.

- **2.4 Decorator Pattern**
- attach a flexible additional responsibilities to an object dynamically
- The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.4 Decorator Pattern**
  - attach a flexible additional responsibilities to an object dynamically
  - The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

Advantages

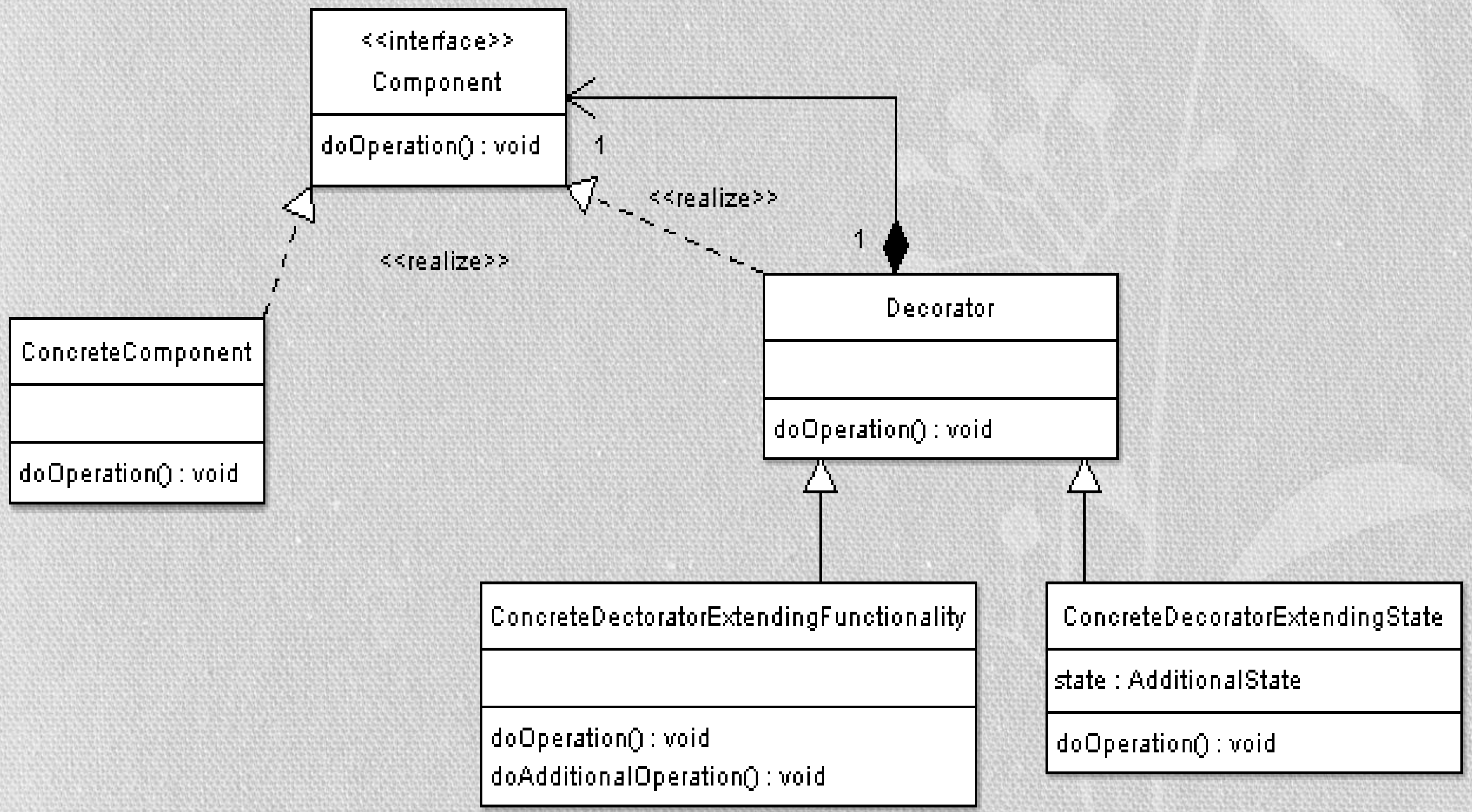
- It provides greater flexibility than static inheritance.
- It enhances the extensibility of the object, because changes are made by coding new classes.
- It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Uses

- When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- When you want to add responsibilities to an object that you may want to change in future.
- Extending functionality by sub-classing is no longer practical.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.4 Decorator Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.4 Decorator Pattern**
- The participants in this pattern are:
  - **Component** - Interface for objects that can have responsibilities added to them dynamically.
  - **ConcreteComponent** - Defines an object to which additional responsibilities can be added.
  - **Decorator** - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
  - **Concrete Decorators** - Concrete Decorators extend the functionality of the component by adding state or adding behavior.
- The decorator pattern applies when there is a need to dynamically add as well as remove responsibilities to a class, and when subclassing would be impossible due to the large number of subclasses that could result.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.5 Proxy Pattern**
  - provide a Placeholder for an object to control references to it
  - provides the control for accessing the original object
  - like hiding the information of original object, on demand loading etc.

Advantages

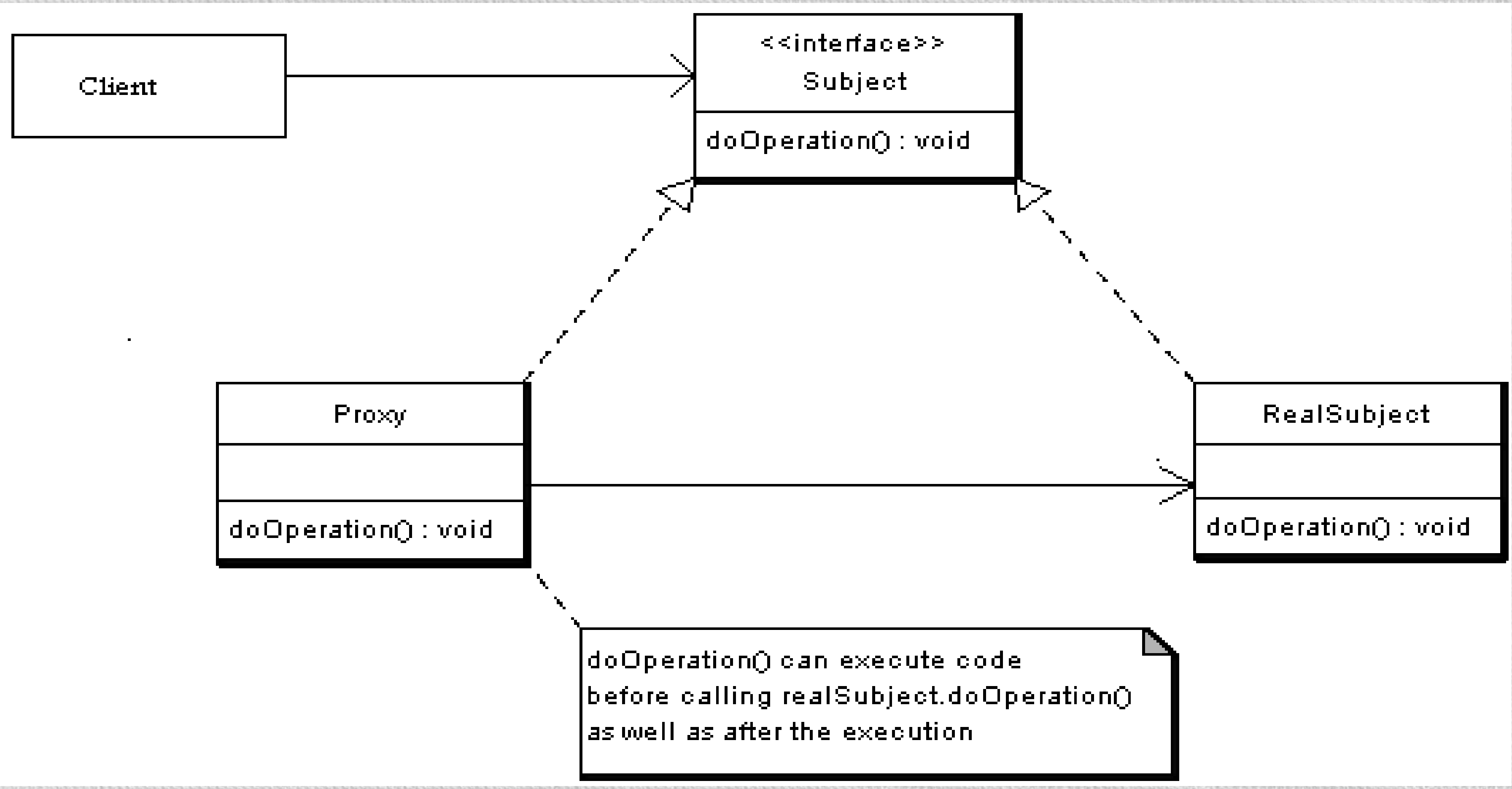
- It provides the protection to the original object from the outside world.

Uses

- It can be used in **Virtual Proxy** scenario
- It can be used in **Protective Proxy** scenario
- It can be used in **Remote Proxy** scenario
- It can be used in **Smart Proxy** scenario

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.5 Proxy Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.5 Proxy Pattern**
- The participants classes in the proxy pattern are:
  - **Subject** - Interface implemented by the RealSubject and representing its services. The interface must be implemented by the proxy as well so that the proxy can be used in any location where the RealSubject can be used.
  - **Proxy**
    - Maintains a reference that allows the Proxy to access the RealSubject.
    - Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.
    - Controls access to the RealSubject and may be responsible for its creation and deletion.
    - Other responsibilities depend on the kind of proxy.
  - **RealSubject** - the real object that the proxy represents.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.5 Proxy Pattern**
  - A client obtains a reference to a Proxy, the client then handles the proxy in the same way it handles RealSubject and thus invoking the method doSomething().
  - At that point the proxy can do different things prior to invoking RealSubject's doSomething() method.
  - The client might create a RealSubject object at that point, perform initialization, check permissions of the client to invoke the method, and then invoke the method on the object.
  - The client can also do additional tasks after invoking the doSomething() method, such as incrementing the number of references to the object.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **2.6 Façade Pattern**
  - just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client
  - Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Advantages

- It shields the clients from the complexities of the sub-system components.
- It promotes loose coupling between subsystems and its clients.

Uses

- When you want to provide simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3 Behavioral Pattern**
  - concerned with algorithms and the assignment of responsibilities between objects.
  - In these design patterns,the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.
- **3.1 Chain of Responsibility**
  - A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".**
  - In other words, we can say that normally each receiver contains reference of another receiver.
  - If one object cannot handle the request then it passes the same to the next receiver and so on.
  - *For example, an ATM uses the Chain of Responsibility design pattern in money giving process.*

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.1 Chain of Responsibility**

Advantages

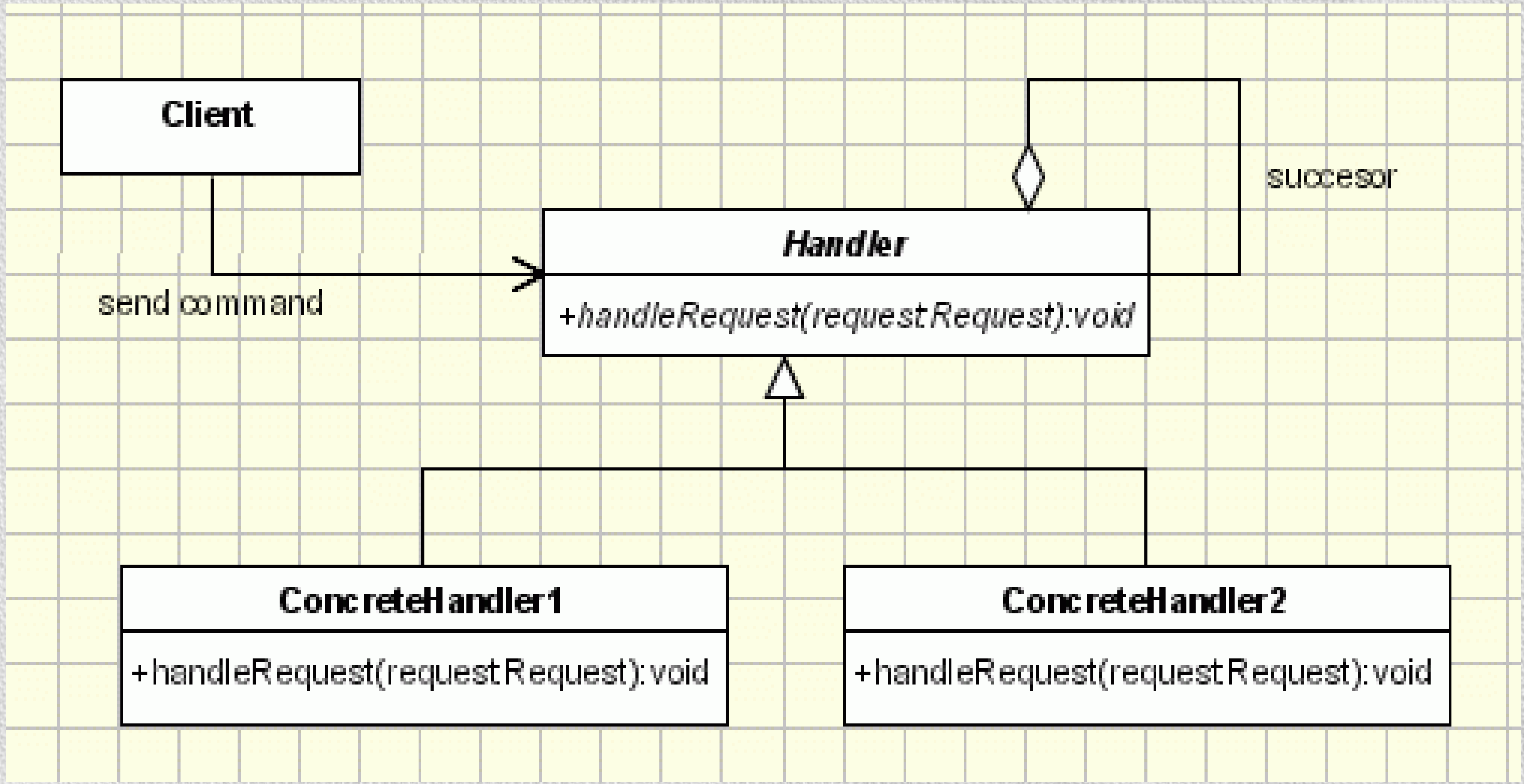
- It reduces the coupling.
- It adds flexibility while assigning the responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

Uses

- When more than one object can handle a request and the handler is unknown.
- When the group of objects that can handle the request must be specified in dynamic way.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.1 Chain of Responsibility**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.1 Chain of Responsibility**
  - The participants are
  - **Handler** - defines an interface for handling requests
  - **ConcreteHandler** - handles the requests it is responsible for . If it can handle the request it does so, otherwise it sends the request to its successor
  - **Client** - sends commands to the first object in the chain that may handle the command
  - the Client in need of a request to be handled sends it to the chain of handlers, which are classes that extend the Handler class.
  - Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client.
  - If ConcreteHandler_i can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler_i+1, the next one in the chain.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.2 Command Pattern**
- A Command Pattern says that "encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command".

Advantages

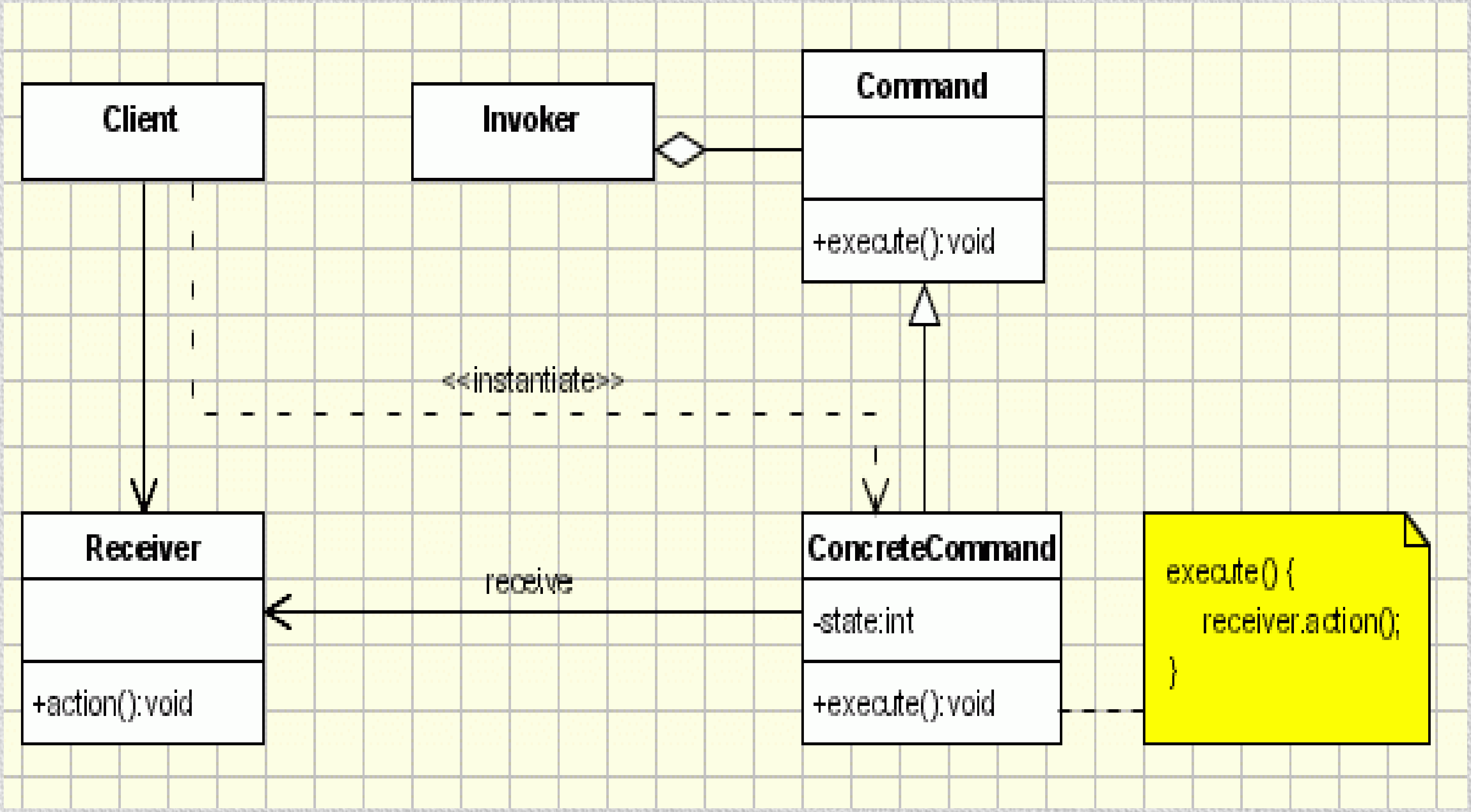
- It separates the object that invokes the operation from the object that actually performs the operation.
- It makes easy to add new commands, because existing classes remain unchanged.

Uses

- When you need parameterize objects according to an action perform.
- When you need to create and execute requests at different times.
- When you need to support rollback, logging or transaction functionality.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.2 Command Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.2 Command Pattern**
  - The participants are
  - **Command** - declares an interface for executing an operation;
  - **ConcreteCommand** - extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action.
  - **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;
  - The Client asks for a command to be executed.
  - The Invoker takes the command, encapsulates it and places it in a queue, in case there is something else to do first, and the ConcreteCommand that is in charge of the requested command, sending its result to the Receiver.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.3 Observer Pattern**
  - An Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

Advantages

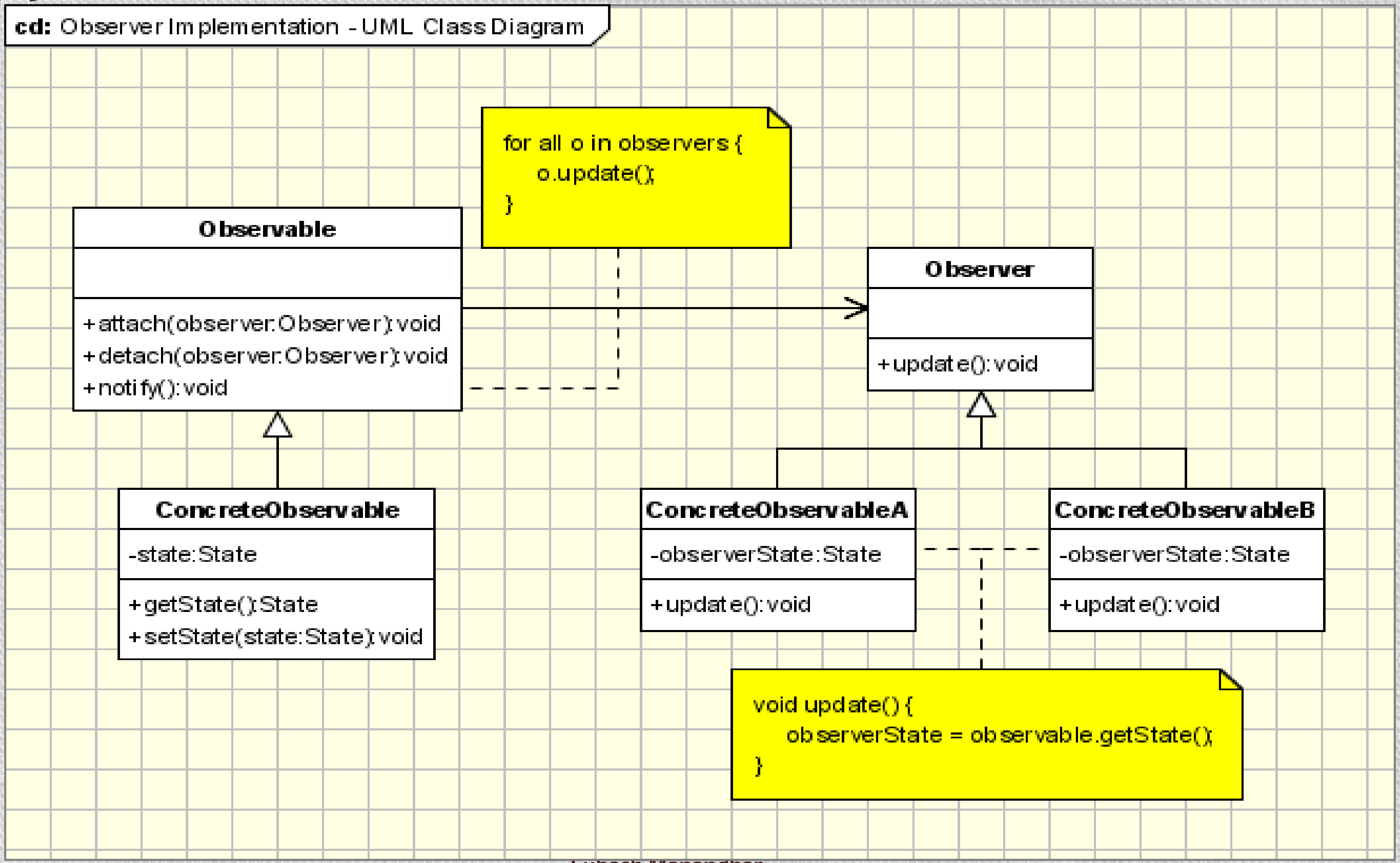
- It describes the coupling between the objects and the observer.
- It provides the support for broadcast-type communication.

Uses

- When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- When the framework we writes and needs to be enhanced in future with new observers with minimal changes.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.3 Observer Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.3 Observer Pattern**
- The participants classes in this pattern are:
  - **Observable** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as **Subject**.
  - **ConcreteObservable** - concrete Observable class. It maintain the state of the object and when a change in the state occurs it notifies the attached **Observers**.
  - **Observer** - interface or abstract class defining the operations to be used to notify this object.
  - **ConcreteObserverA, ConcreteObserver2** - concrete **Observer** implementations.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.3 Observer Pattern**
  - The main framework instantiate the ConcreteObservable object.
  - Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface.
  - Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface.
  - When a new Observer is added to the application, all we need to do is to instantiate it in the main framework and to add attach it to the Observable object.
  - The classes already created will remain unchanged.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.4 Memento Pattern**
  - The intent of this pattern is to capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

Advantages

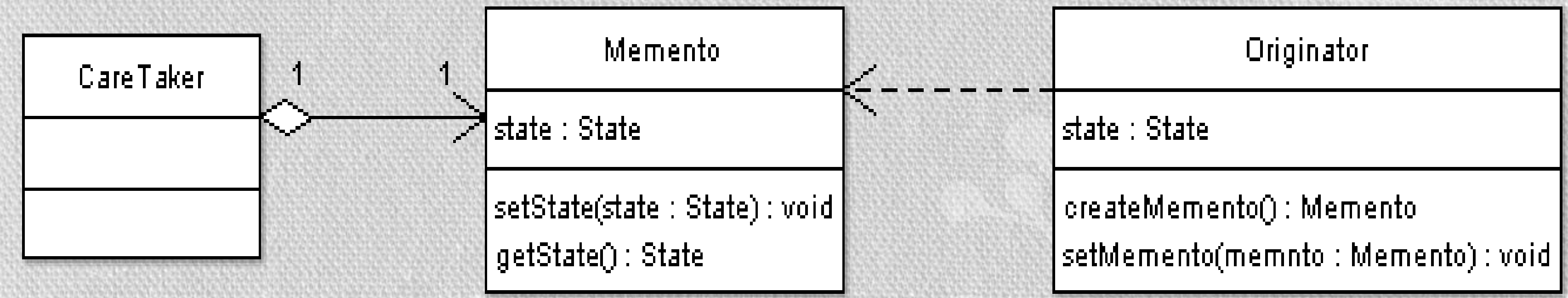
- It preserves encapsulation boundaries.
- It simplifies the originator.

Uses

- It is used in Undo and Redo operations in most software.
- It is also used in database transactions.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.4 Memento Pattern**



- The participants are
- **Memento**
  - Stores internal state of the Originator object. The state can include any number of state variables.
  - The Memento must have two interfaces, an interface to the caretaker. This interface must not allow any operations or any access to internal state stored by the memento and thus honors encapsulation. The other interface is to the originator and allows the originator to access any state variables necessary to for the originator to restore previous state.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.4 Memento Pattern**
  - **Originator**
    - Creates a memento object capturing the originators internal state.
    - Use the memento object to restore its previous state.
  - **Caretaker**
    - Responsible for keeping the memento.
    - The memento is opaque to the caretaker, and the caretaker must not operate on it.
- A Caretaker would like to perform an operation on the Originator while having the possibility to rollback.
- The caretaker calls the createMemento() method on the originator asking the originator to pass it a memento object.
- At this point the originator creates a memento object saving its internal state and passes the memento to the caretaker.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.4 Memento Pattern**
- The caretaker maintains the memento object and performs the operation.
- In case of the need to undo the operation, the caretaker calls the setMemento() method on the originator passing the maintained memento object.
- The originator would accept the memento, using it to restore its previous state.
- **3.5 Mediator Pattern**
  - says that "to define an object that encapsulates how a set of objects interact".
  - used to reduce communication complexity between multiple objects or classes.
  - provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.5 Mediator Pattern**

Advantages

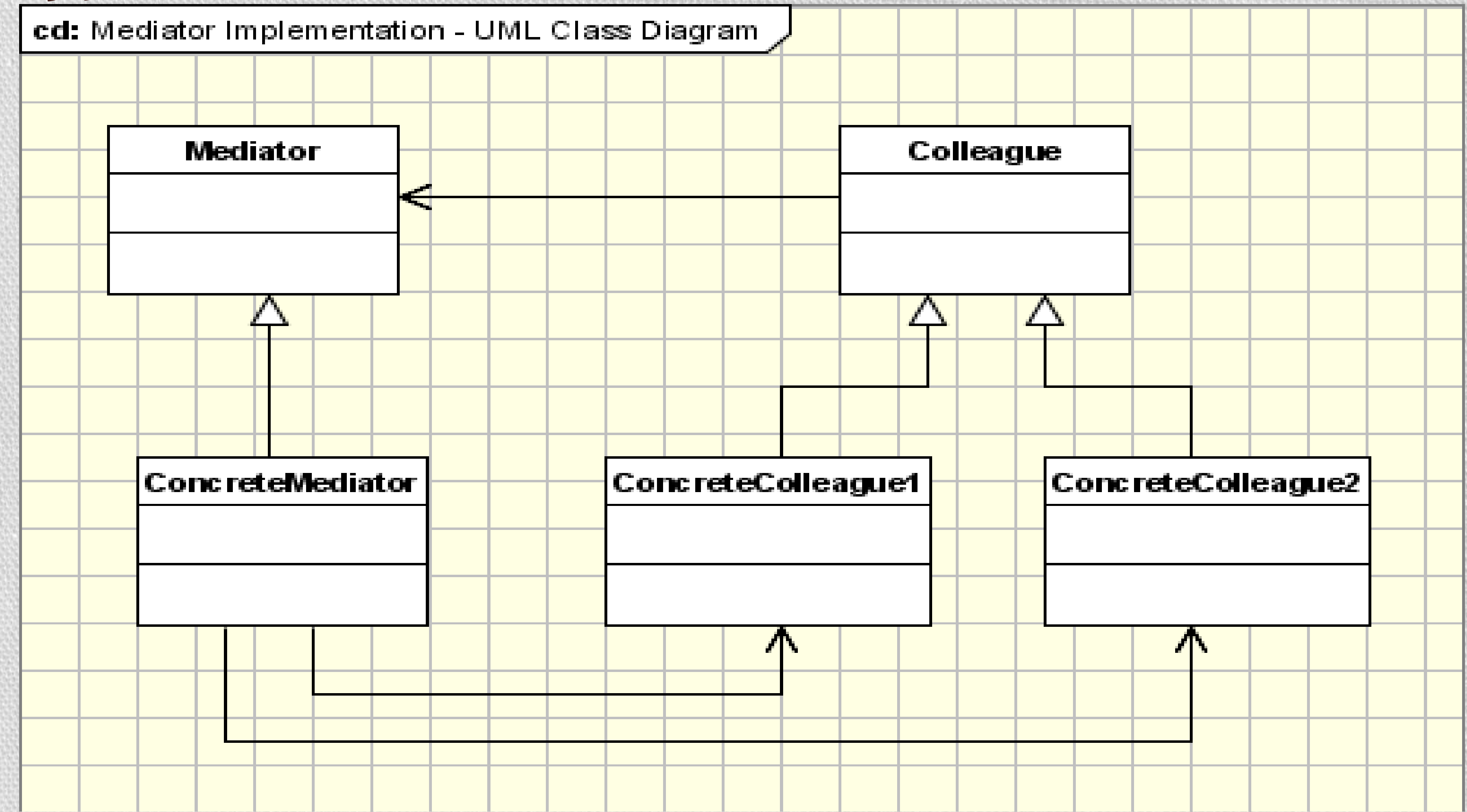
- It decouples the number of classes.
- It simplifies object protocols.
- It centralizes the control.
- The individual components become simpler and much easier to deal with because they don't need to pass messages to one another. The components don't need to contain logic to deal with their intercommunication and therefore, they are more generic.

Uses

- It is commonly used in message-based systems likewise chat applications.
- When the set of objects communicate in complex but in well-defined ways.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.5 Mediator Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.5 Mediator Pattern**
  - The participants classes in this pattern are:
  - **Mediator** - defines an interface for communicating with Colleague objects.
  - **ConcreteMediator** - knows the colleague classes and keep a reference to the colleague objects.
    - - implements the communication and transfer the messages between the colleague classes
  - **Colleague classes** - keep a reference to its Mediator object
    - - communicates with the Mediator whenever it would have otherwise communicated with another Colleague

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.6 Template Pattern**
  - says that "just define the skeleton of a function in an operation, deferring some steps to its subclasses".
  - lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

Advantage

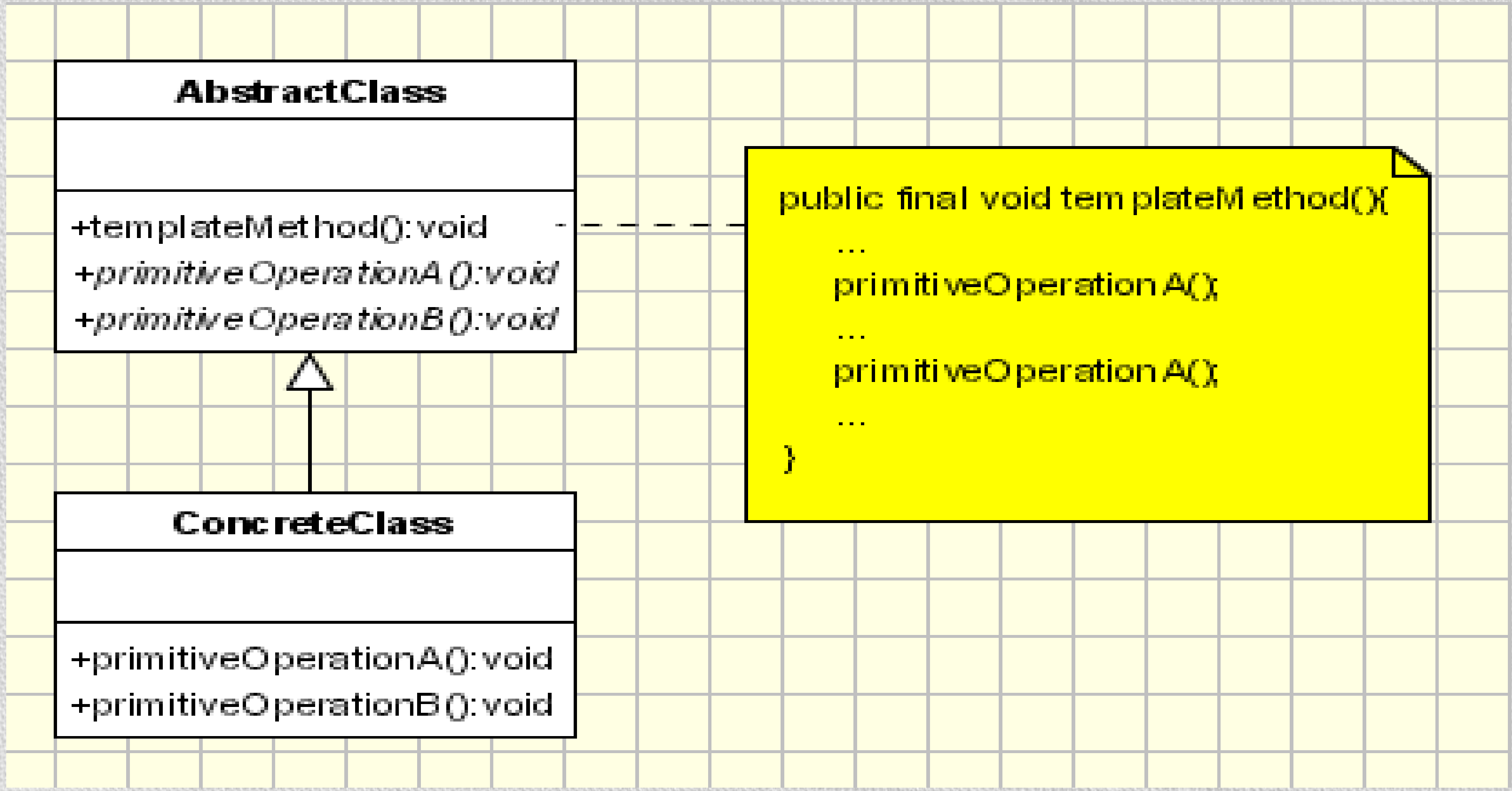
- It is very common technique for reusing the code.

Uses

- It is used when the common behavior among sub-classes should be moved to a single common class by avoiding the duplication.

3.4 Classification of Design patterns

- **3.6 Template Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.6 Template Pattern**
  - The Participants are:
  - **AbstractClass** - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
    - - implements a template method which defines 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.
  - When a concrete class is called the template method code will be executed from the base class while for each method used inside the template method will be called the implementation from the derived class.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.7 Strategy Pattern**
  - says that "defines a family of functionality, encapsulate each one, and make them interchangeable".
  - lets the algorithm vary independently from clients that use it.

Advantages

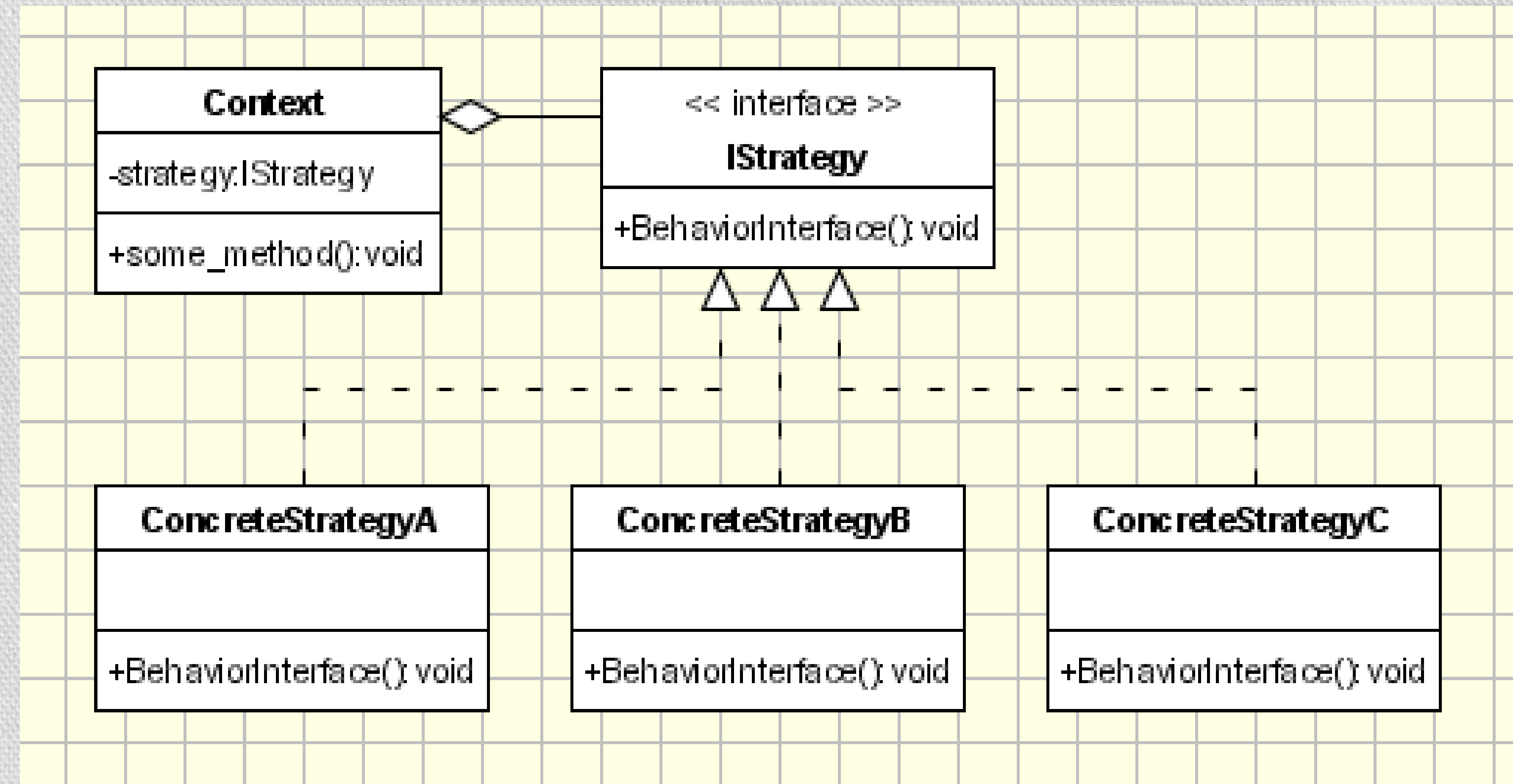
- It provides a substitute to subclassing.
- It defines each behavior within its own class, eliminating the need for conditional statements.
- It makes it easier to extend and incorporate new behavior without changing the application.

Uses

- When the multiple classes differ only in their behaviors.e.g. Servlet API.
- It is used when you need different variations of an algorithm.

Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.7 Strategy Pattern**



Chapter 3 – Design Patterns
3.4 Classification of Design patterns

- **3.7 Strategy Pattern**
  - The Participants are:
    - **Strategy** - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
    - **ConcreteStrategy** - each concrete strategy implements an algorithm.
    - **Context** contains a reference to a strategy object. may define an interface that lets strategy accessing its data.
  - The Context objects contains a reference to the ConcreteStrategy that should be used.
  - When an operation is required then the algorithm is run from the strategy object.
  - The Context is not aware of the strategy implementation.
  - If necessary, addition objects can be defined to pass data from context object to strategy.
  - The context object receives requests from the client and delegates them to the strategy object.
  - Usually the ConcreteStartegy is created by the client and passed to the context.
  - From this point the clients interacts only with the context.

Chapter 3 – Design Patterns
3.5 Documenting and Describing Patterns

- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.

Chapter 3 – Design Patterns
3.5 Documenting and Describing Patterns

- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Chapter 3 – Design Patterns
3.6 Criticism

- The concept of design patterns has been criticized by some in the field of computer science.
- **<u>Does not differ significantly from other abstractions</u>**
  - Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary.
  - The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

Chapter 3 – Design Patterns
3.6 Criticism

- **Leads to inefficient solutions**
  - The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.
- **Lacks formal foundations**
  - The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

Chapter 3 – Design Patterns
3.6 Criticism

- **Targets the wrong problem**
  - The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay Revenge of the Nerds
  - Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.
- **Inefficient solutions**
  - Patterns try to systematize approaches that are already widely used. This unification is viewed by many as a belief and they implement patterns "to the point", without adapting them to the context of their project.
- **Unjustified use**
  - "If all you have is a hammer, everything looks like a nail."
  - This is the problem that haunts many novices who have just familiarized themselves with patterns. Having learned about patterns, they try to apply them everywhere, even in situations where simpler code would do just fine.