Design Patterns

Lecture 1

Manuel Mastrofini

Systems Engineering and Web Services

University of Rome Tor Vergata

June 2011

Definition

- A pattern is a reusable solution to a commonly occurring problem within a given context.
  - A pattern is a proposed way to resolve a problem.
  - In the field of software design, examples of a problem may be:
    - Control the creation of objects.
    - Assign responsibilities to classes or objects.
    - Manage collections of classes and objects.
    - Adapt the runtime behavior of objects.
  - Context may differentiate two structurally similar patterns.
- A pattern may be used for both design and refactoring, in order to improve quality.
- *Main reference for design patterns: "Design Patterns: Elements of Reusable Object Oriented Software" by E. Gamma,* R. Helm, R. Johnson, and J. Vlissides (Gang of four).
  - It contains 23 patterns, classified into 3 categories: creational, behavioral and structural.

Classification of patterns (GoF)

- A pattern has:
  - **Name**
  - **Purpose**: what a pattern does.
    - *Creational*: facilitate object creation.
    - *Structural*: support composition of classes and objects.
    - *Behavioral*: help distribute responsibilities and control interactions.
  - **Scope**: if a pattern applies to classes or to objects.
    - Class patterns include compile-time defined relationships (generalization).
    - Object patterns allow relationships to be created and changed at run-time.

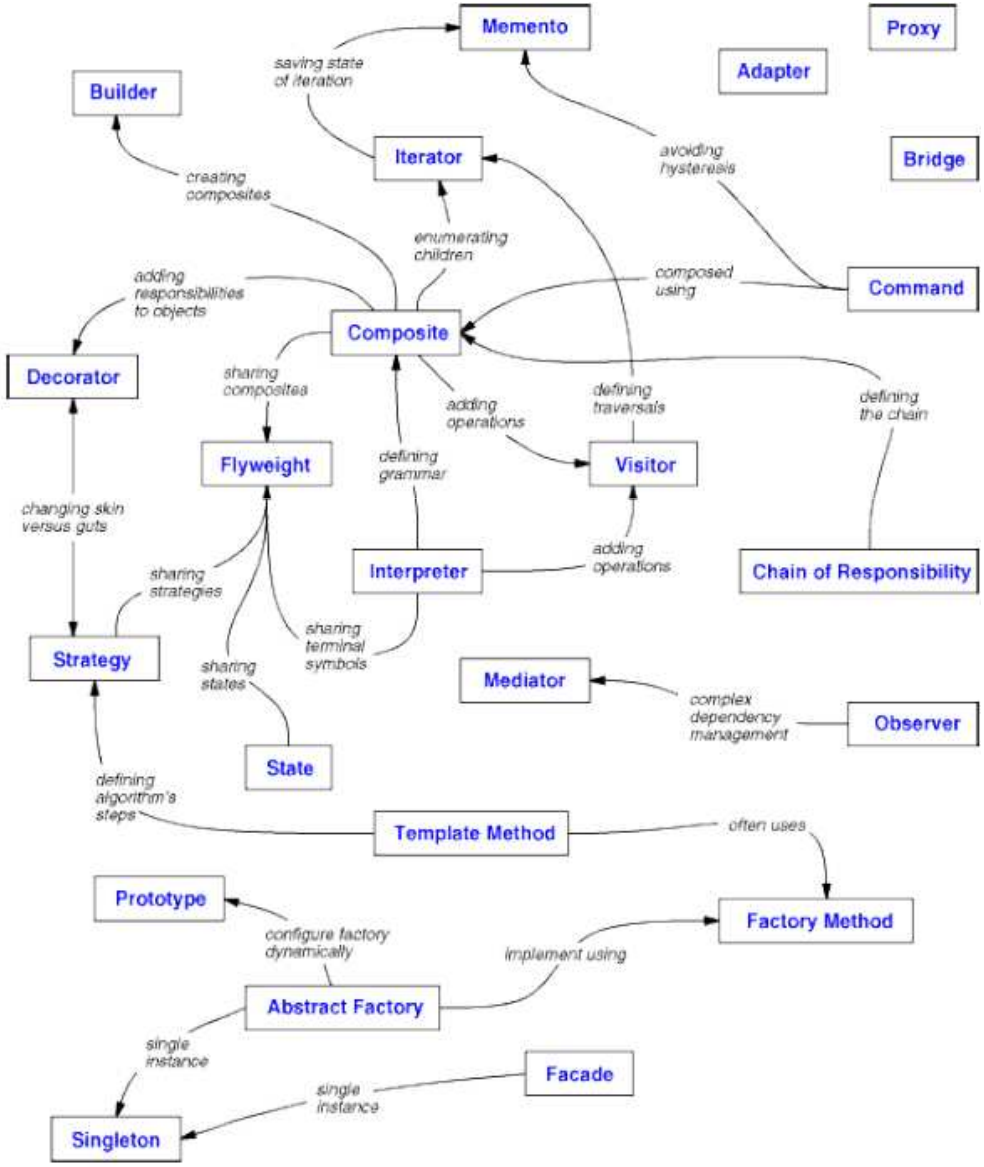
Creational	Structural	Behavioral	
Abstract Factory (o)	*Adapter (c/o)*	Chain of responsibility (o)	Observer (o)
Builder (o)	Bridge (o)	Command (o)	State (o)
Factory Method (c)	Composite (o)	Interpreter (o)	Strategy (o)
Prototype (o)	Decorator (o)	Iterator (o)	*Template method (c)*
Singleton (o)	Facade (o)	Mediator (o)	Visitor (o)
	Flyweight (o)	Memento (o)	
	Proxy (o)		

Why (and what) patterns

- Patterns help create **high quality software**, i.e. encapsulation, modularity, flexibility, ease of use, evolution, extensibility.
  - ***Algorithmic independency***: c: Builder; s:; b: Iterator, Strategy, *Template Method*, Visitor.
  - ***Creating an object without specifying a class explicitly***: c: Abstract Factory, Factory Method, Prototype.
  - ***Independence from specific operations***: b: Chain of responsibility, Command.
  - ***Independence from object representation or implementation***: c: Abstract Factory; s:Bridge, Proxy.
  - ***Loose coupling***: c: Abstract Factory; s: Bridge, Façade; b: Command, Mediator, Observer.
  - ***Extending functionality by sub-classing***: s: Bridge, Composite, Decorator; b:Command, Observer, Strategy.
  - ***Ability to alter classes conveniently***: c:*Adapter*; b:Decorator, Visitor.

Pattern inter-relationships

- There are many relationships among patterns.
- Some patterns are alternative.
  - One has to be selected.
- Some patterns are complementary.
  - They are expected to be combined.
- Parts of a complex pattern may be implemented by using other patterns.

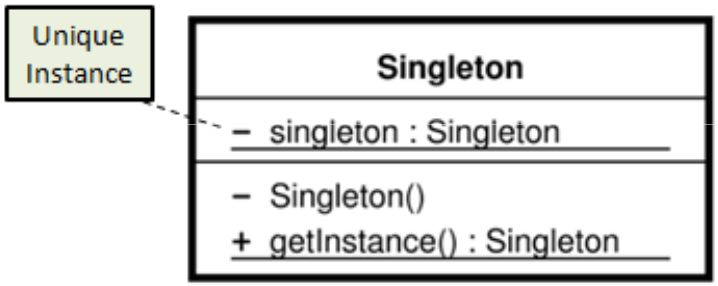


Creational patterns

Singleton

- **Intent**: ensure a class only has one instance at most, and provide a global point of access to it.

- ***Singleton***: defines an Instance operation that lets clients access its unique instance. Instance is a class operation.

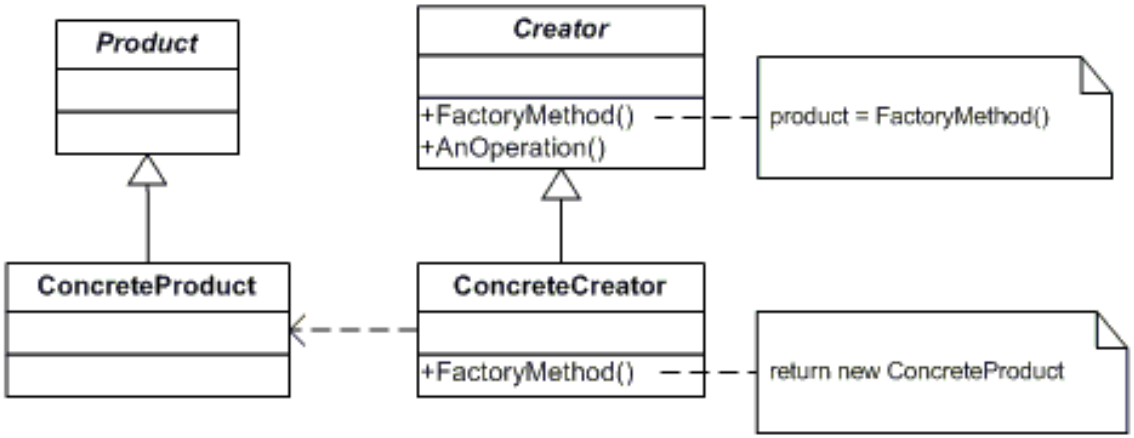


Variant: Instead of one instance, the number of class instances may be controlled to be any number.

Factory Method (Virtual Constructor)

- **Intent**: define an interface for creating an object, but let subclasses decide which class to instantiate.

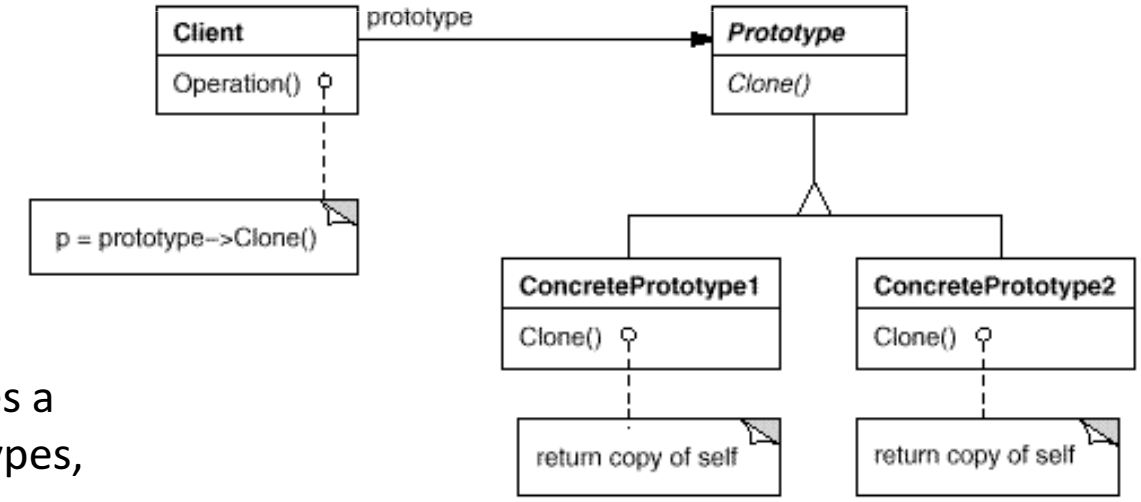
- ***Product***: defines the interface of objects the factory method creates.
- ***ConcreteProduct***: implements the Product interface.
- ***Creator***: declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns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



Prototype

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

- ***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.



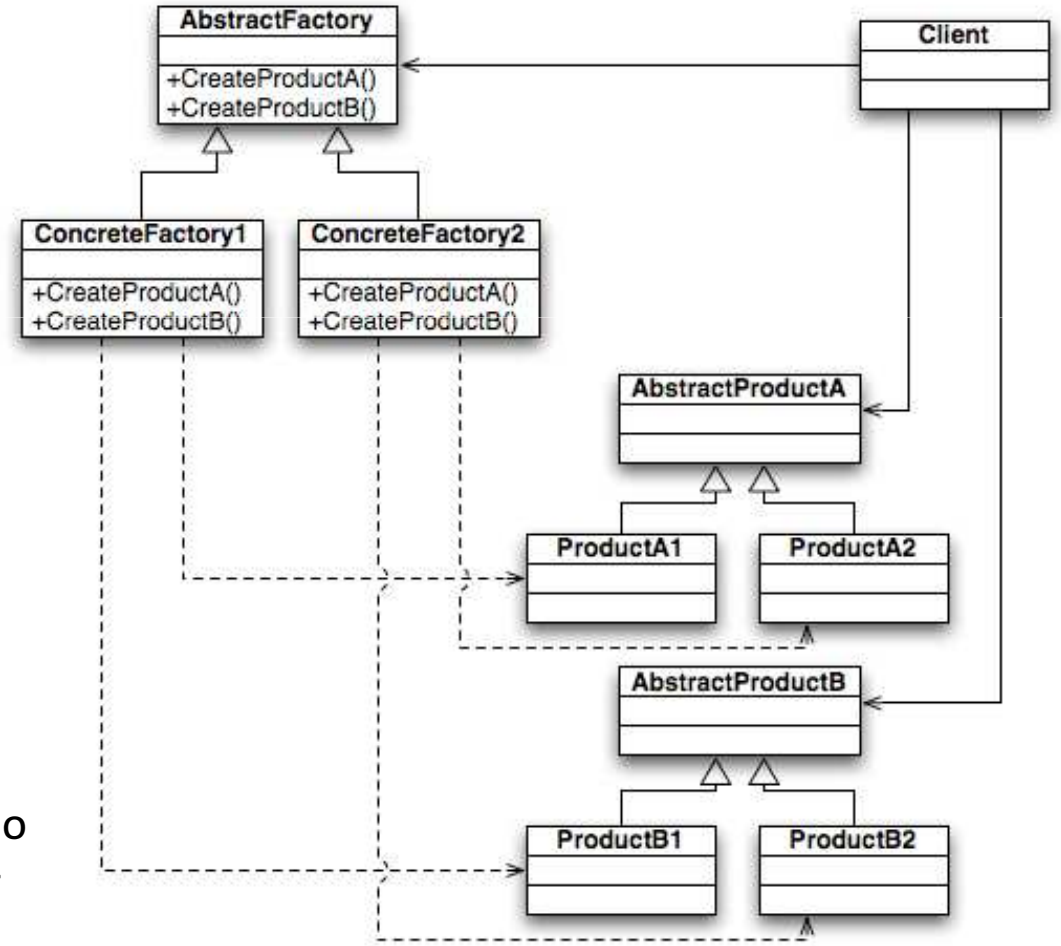
Variant: Create a Prototype Manager class which manages a registry containing all prototypes, so that the Client ask for a prototype by providing a key.

Abstract Factory (Kit)

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

- ***AbstractFactory***: interface for operations that create abstract product objects. It could be a singleton and generally leverages the factory method pattern.
- ***ConcreteFactory***: implements the operations to create concrete product objects.
- ***AbstractProduct***: interface for a type of product object.
- ***ConcreteProduct***: to be created by the corresponding concrete factory.

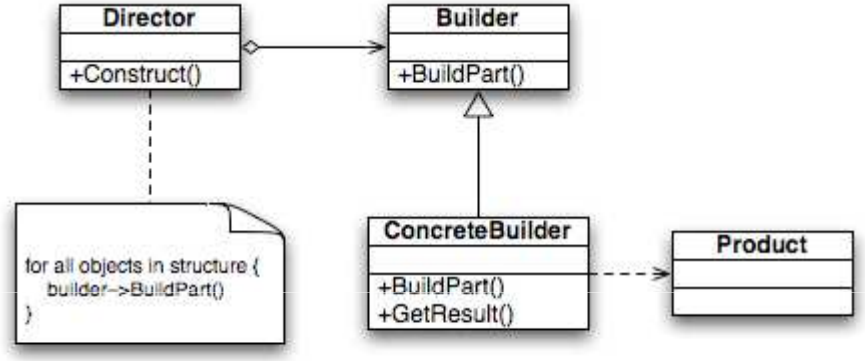
Variant: If there are many product families, then the Prototype pattern could be used; in this case, there is no need to create a ConcreteFactory for each family.

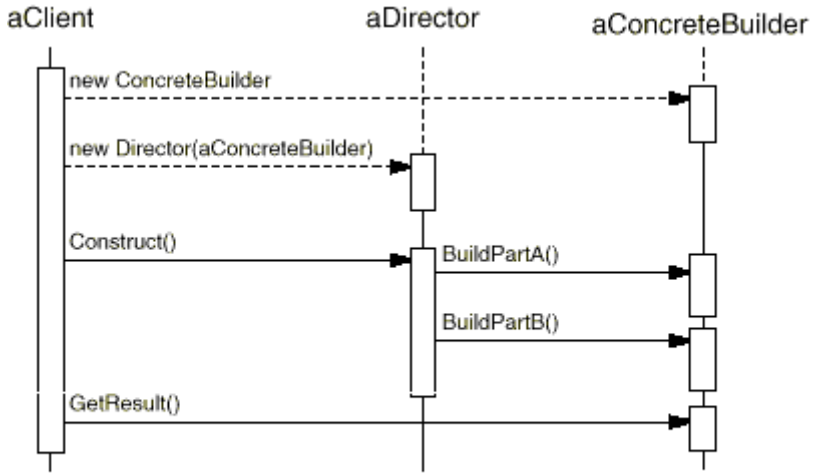


Builder

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

- ***Builder***: specifies an abstract interface for creating parts of a Product object. ***Director***: constructs an object using the Builder interface.
- ***Product***: represents the complex object under construction.
- ***ConcreteBuilder***: constructs and assembles parts of the product by implementing the Builder interface and defines the process by which it is assembled; comprises classes that define the constituent parts; defines and keeps track of the representation it creates; provides an interface for retrieving the product.





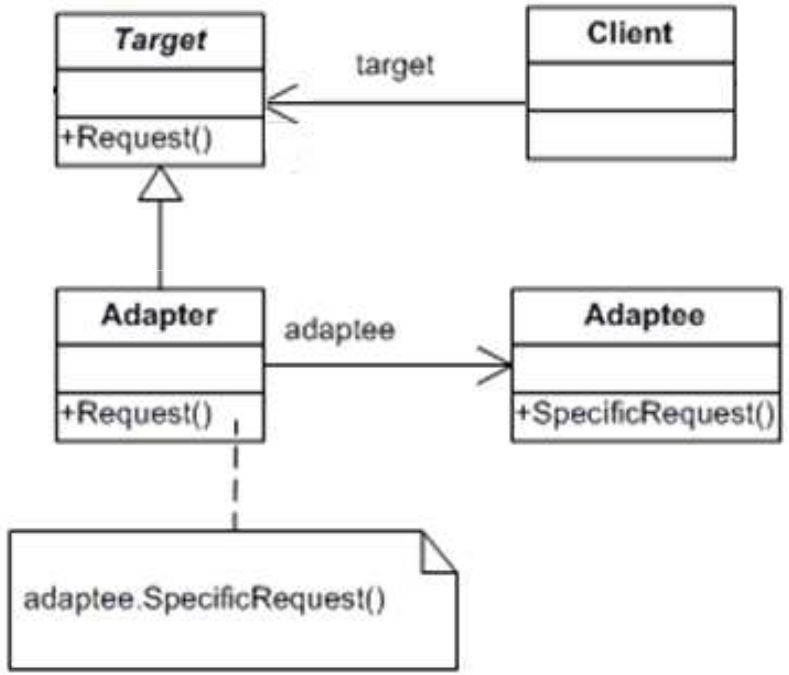
Structural patterns

Part 1

Adapter (Wrapper)

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

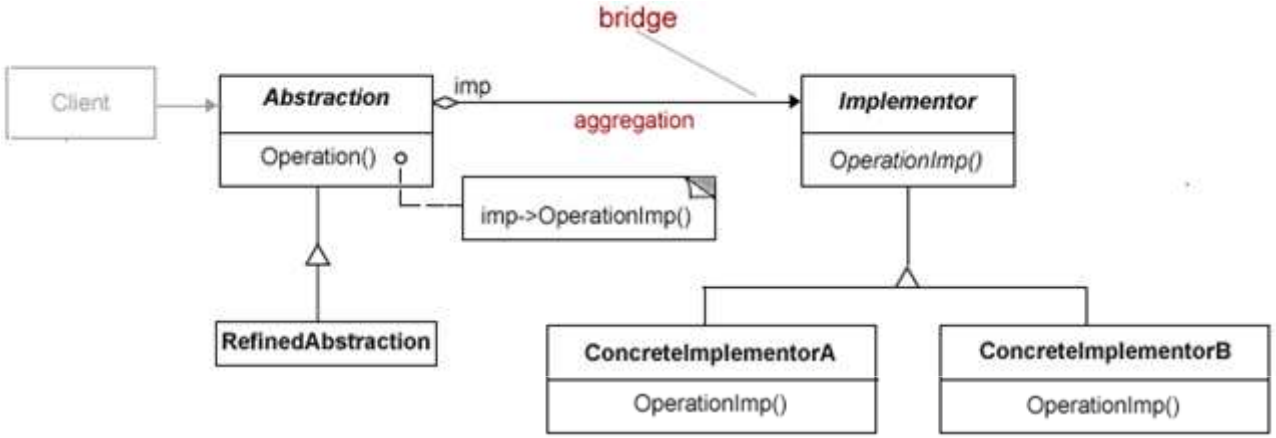
- ***Target***: defines the domain-specific interface that Client uses.
- ***Adapter*****:** adapts the interface of Adaptee to the Target interface. In inherits from Adaptee (class adapter) or have a composition with it (object adapter). In the first case, subclasses of Adaptee cannot leverage the pattern adaption
- ***Adaptee***: defines an existing interface that needs adapting.



Variant: When two views of the same entity are used by two clients, a two-way adapter can be built by using multiple inheritance: the adapter inherits from the two classes used by the two clients.

Bridge (Body, Handle)

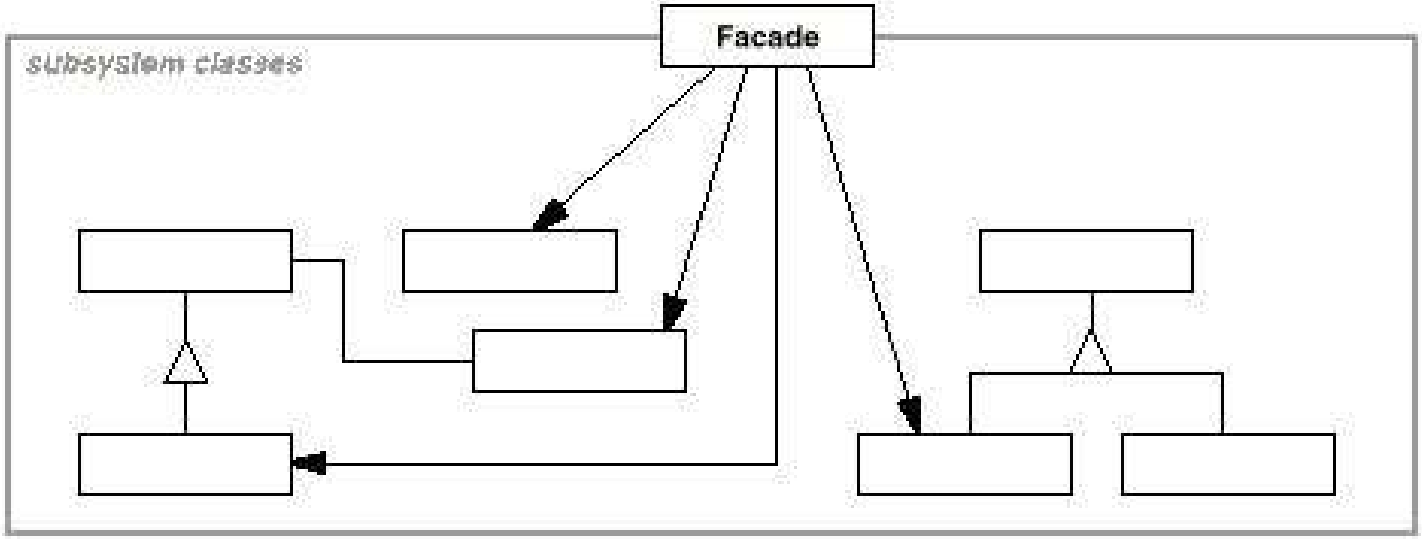
- **Intent**: It decouples an abstraction from its implementation so that the two can vary independently and be extended independently.



- ***Abstraction***: defines the abstraction's interface and maintains a reference to an object of type Implementor. It could be aware of the Implementor's hierarchy and use the right subclass, or it could leverage the Abstract Factory pattern to instantiate the right subclass without statically coupling any concrete implementor to the Abstraction.
- ***RefinedAbstraction***: extends the interface defined by Abstraction.
- ***Implementor***: defines the interface for implementation classes. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ***ConcreteImplementor***: defines the concrete implementation of Implementor.

Façade

- **Intent**: Provide a unified interface to a set of interfaces in a subsystem.



- ***Façade***: knows which subsystem classes are responsible for a request and delegates client requests to appropriate subsystem objects. It is generally a Singleton.

Variant: If façade is abstract, then different implementation of the same subsystems can be configured by the client. This could be an implementation of Abstract Factory.