Design Patterns

C.K.Leng

Background

- Gamma, Helm, Johnson, and Vlissides (the "Gang of Four") – Design Patterns, Elements of Reusable Object-Oriented Software
- This book solidified thinking about patterns and became the seminal Design Patterns text
- Software design patterns are based (somewhat) on work by the architect Christopher Alexander

Purpose

- A design pattern captures *design expertise* – patterns are not created from thin air, but abstracted from *existing* design examples
- Using design patterns is *reuse* of design expertise
- Studying design patterns is a way of studying how the "experts" do design
- Design patterns provide a *vocabulary* for talking about design

General Software Design Heuristics

- Do the right things, then do the things right
- Prevention is always better then cure
- Minimize the impact of change
- Maximize your freedom
- Single assignment of responsibility
- Loose Coupling
- High Cohesion
- Etc..

OO key selling points

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Why Design Patterns are Needed?

The one constant in software development

Okay, what's the one thing you can always count on in software development?

No matter where you work, what you're building, or what language you are programming in, what's the one true constant that will be with you always?

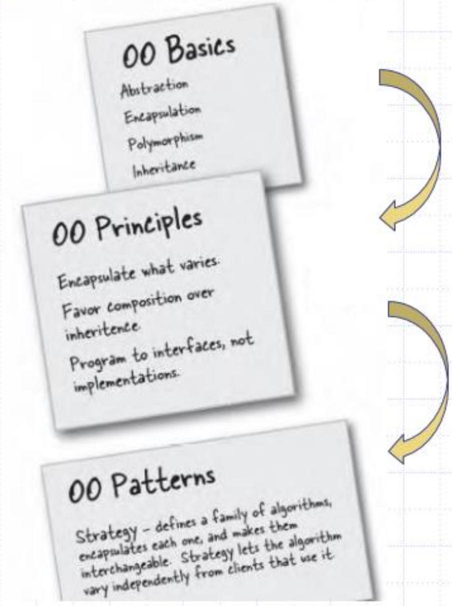
CHANGE

No matter how well you design an application, over time an application must grow and change or it will *die*.

Take what varies and "encapsulate" it so it won't affect the rest of your code.

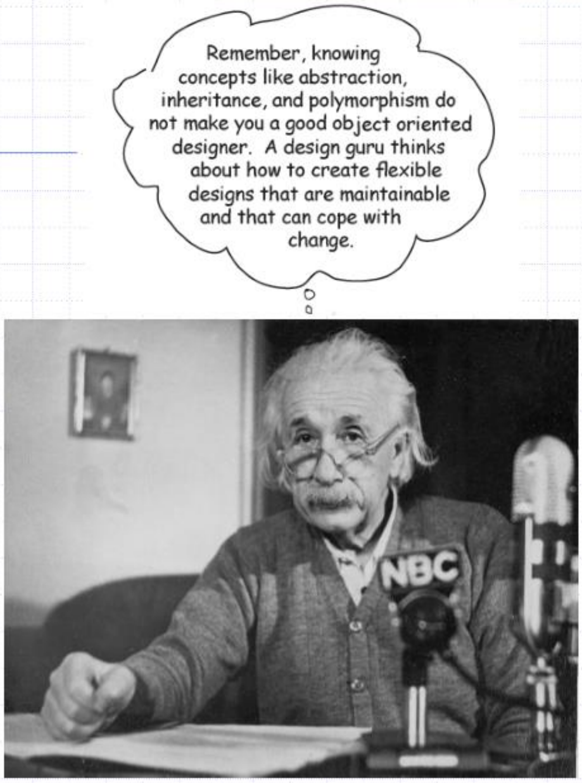
The result? Fewer unintended consequences from code changes and more flexibility in your systems!

Why Design Patterns are Needed?



OO Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favour composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Depend upon abstraction. Do not depend upon concrete classes
- Don’t call us, we’ll call you
- A class should have only one reason to change



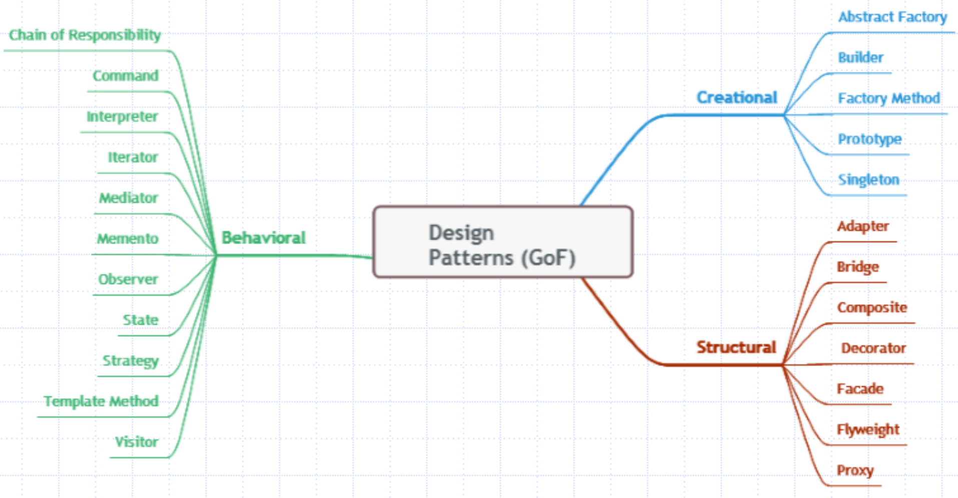
Structure of a pattern

- Name
- Intent
- Motivation
- Applicability
- Structure
- Consequences
- Implementation
- Known Uses
- Related Patterns

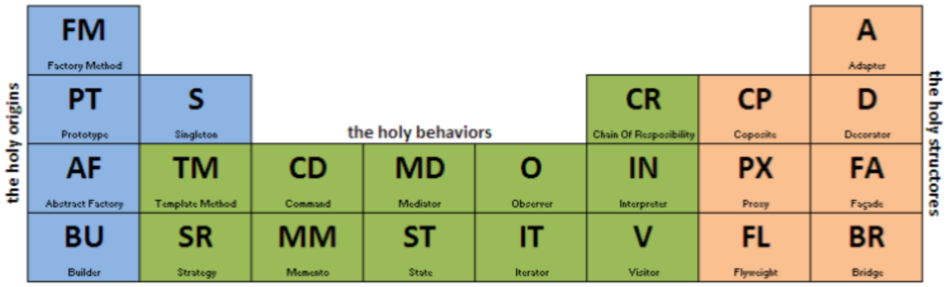
Key patterns

- The following patterns are what considered to be a good "basic" set of design patterns
- Competence in recognizing and applying these patterns *will* improve your low-level design skills
- 3 Categories
  - Structural
  - Behavioral
  - Creational

Key Patterns



Patterns



Understand Design Pattern Representation in UML

- Class
- Abstract Class
- Interface
- Aggregation & Composition
- Multiplicity
- Association
- Roles
- Operations and Methods

UML Class Diagram Notation

Program element	Diagram element	Meaning
Class	Class −attribute +operation()	Types and parameters specified when important; access indicated by + (public), (private), and # (protected).
Interface	<<interface>> IClass +operation()	Name starts with I. Also used for abstract classes.
Note	descriptive text	Any descriptive text.

UML Class Diagram Notation (Cont.)

Program element	Diagram element	Meaning
Package	Package	Grouping of classes and interfaces.
Inheritance	A B	B inherits from A.
Realization	A B	B implements A.

UML Class Diagram Notation (Cont.)

Program element	Diagram element	Meaning
Association	A ——— B	A and B call and access each other's elements.
Association (one way)	A ——> B	A can call and access B's elements, but not vice versa.
Aggregation	A ◇—— B	A has a B, and B can outlive A.
Composition	A ◆—— B	A has a B, and B depends on A.

Patterns vs "Design"

- Patterns *are* design
  - But: patterns transcend the "identify classes and associations" approach to design
  - Instead: learn to recognize patterns in the *problem* space and translate to the solution
- Patterns can capture OO design principles within a specific domain
- Patterns provide structure to "design"

Patterns vs Frameworks

- Patterns are lower-level than frameworks
- Frameworks typically employ many patterns:
  - Factory
  - Strategy
  - Composite
  - Observer
- Done well, patterns are the "plumbing" of a framework

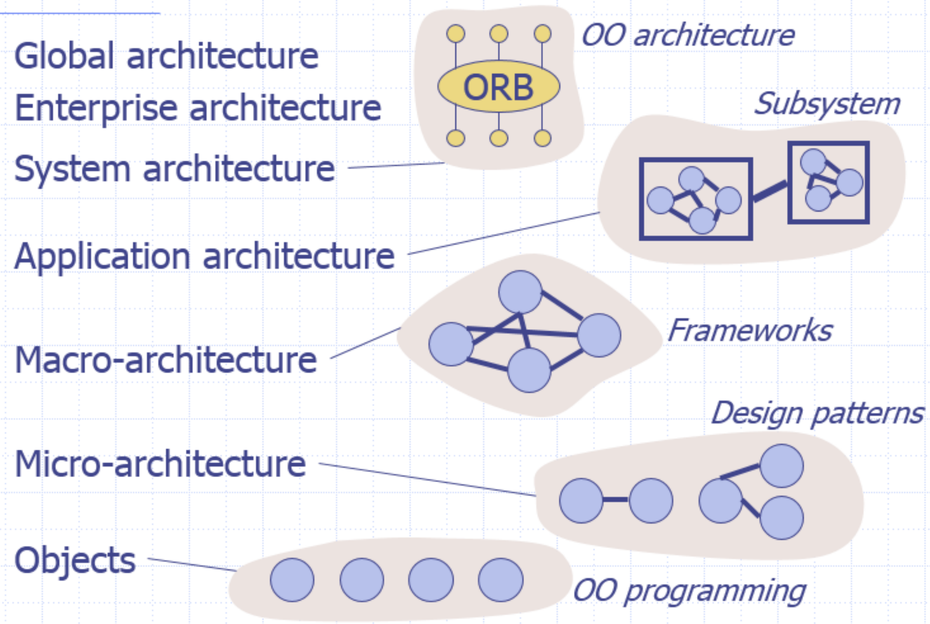
Patterns vs Architecture

- Design Patterns (GoF) represent a lower level of system structure than "architecture" (cf: seven levels of A)
- Patterns can be applied to architecture:
  - Mowbray and Malveau
  - Buschmann *et al*
  - Schmidt *et al*
- Architectural patterns tend to be focussed on middleware. They are good at capturing:
  - Concurrency
  - Distribution
  - Synchronization

Why design patterns in Software Architecture?

- If you're a software engineer, you should know about them anyway
- There are many architectural patterns published, and the GoF Design Patterns is a prerequisite to understanding these:
  - Mowbray and Malveau – CORBA Design Patterns
  - Schmidt et al – Pattern-Oriented Software Architecture
- Design Patterns help you *break out* of first-generation OO thought patterns

The seven layers of architecture*



* Mowbray and Malveau

Concluding remarks

- Design Patterns (GoF) provide a foundation for further understanding of:
  - Object-Oriented design
  - Software Architecture
- Understanding patterns can take some time
  - Re-reading them over time helps
  - As does applying them in your own designs!

Day 2: Creational Patterns

C.K.Leng

Creational Patterns

Purposes

- Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Purposes (Cont…)

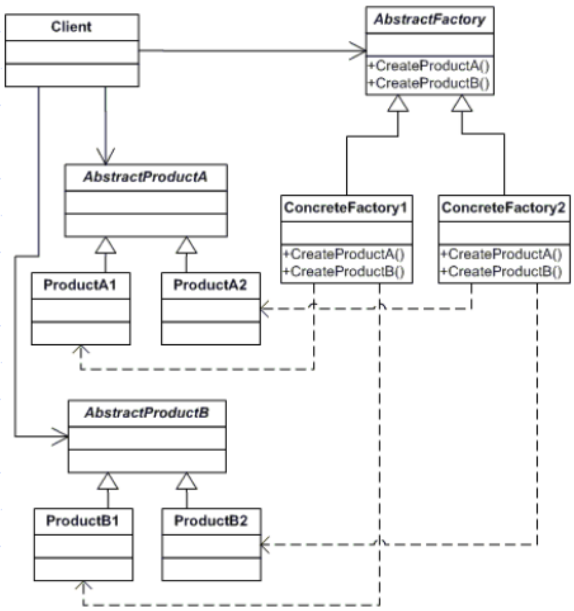
- Patterns whose sole purpose is to facilitate the work of creating, initializing, and configuring objects and classes.
- These types of patterns are useful when we need to render instances of objects, store these objects, perform complex initialization of objects, or create copies of objects.

Patterns

- **Abstract Factory**
  - Creates an instance of several families of classes
- **Builder**
  - Separates object construction from its representation
- **Factory Method**
  - Creates an instance of several derived classes
- **Prototype**
  - A fully initialized instance to be copied or cloned
- **Simple Factory Pattern**
  - Returns an instance of one of several possible classes, depending on the data provided to it
- **Singleton**
  - A class of which only a single instance can exist
- **Object Pool**
  - Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

Creational Patterns: Abstract Factory

UML: *Abstract Factory*



Frequency of use: 1 2 3 4 5 high

Creational Patterns: Abstract Factory

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible “platforms”, and the construction of a suite of “products”.
- The **new** operator considered harmful.

Creational Patterns: Abstract Factory

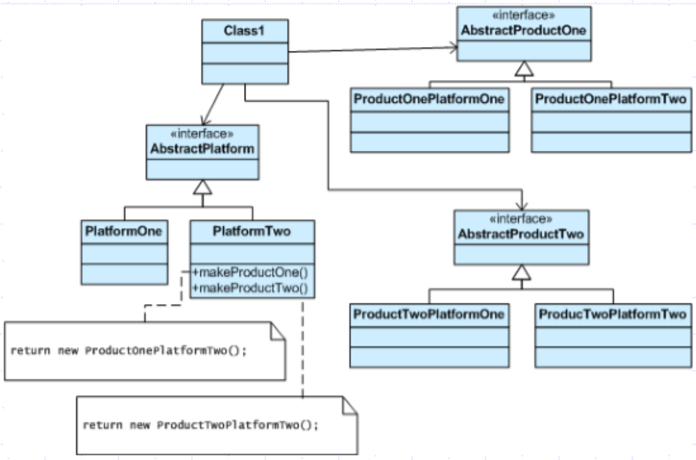
Role

- This pattern supports the creation of products that exist in families and are designed to be produced together.
- The abstract factory can be refined to concrete factories, each of which can create different products of different types and in different combinations.
- The pattern isolates the product definitions and their class names from the client so that the only way to get one of them is through a factory. For this reason, product families can easily be interchanged or updated without upsetting the structure of the client.

Creational Patterns: Abstract Factory

Structure

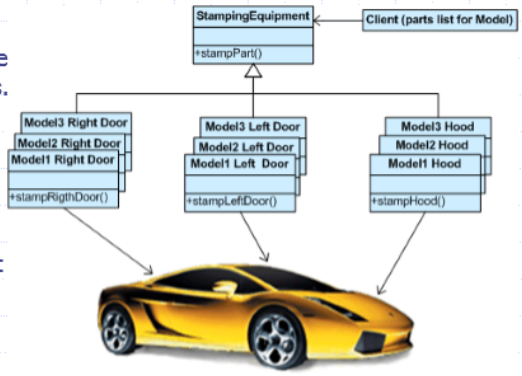
- The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the new operator and the concrete, platform-specific, product classes. Each "platform" is then modeled with a Factory derived class.



Creational Patterns: Abstract Factory

Example

- The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes.
- This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes



Creational Patterns: Abstract Factory

Problem

- If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc.
- Too often, this encapsulation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

- Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes.
- The "factory" object has the responsibility for providing creation services for the entire platform family.
- Clients never create platform objects directly, they ask the factory to do that for them.
- This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated
- The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.
- Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

Rules of Thumb

- Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes.

Rules of Thumb (Cont…)

- Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object.
- Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.
- Abstract Factory can be used as an alternative to Façade to hide platform-specific classes.

Creational Patterns: Abstract Factory

Rules of Thumb (Cont…)

- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

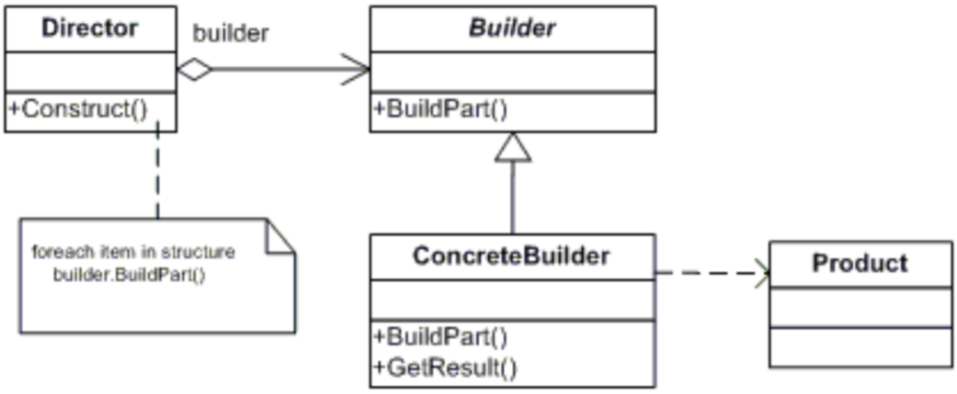
Creational Patterns: Abstract Factory

Known Uses

- Use the Abstract Factory pattern when…
  - A system should be independent of how its products are created, composed, and represented.
  - A system can be configured with one of multiple families of products.
  - The constraint requiring products from the same factory to be used

Creational Patterns: Builder

UML: *Builder*



Frequency of use: 1 2 3 4 5 medium low

Creational Patterns: Builder

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

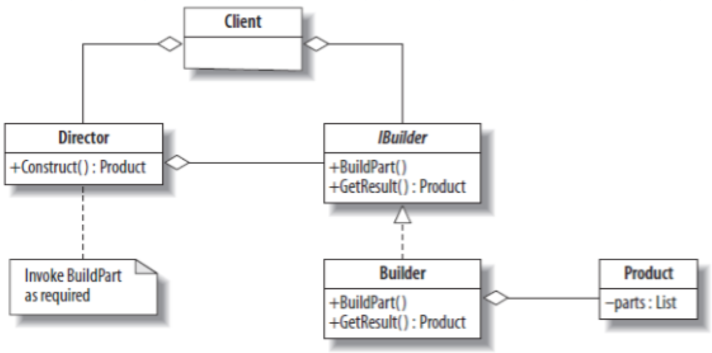
Role

- The Builder pattern separates the specification of a complex object from its actual construction. The same construction process can create different representations.

Creational Patterns: Builder

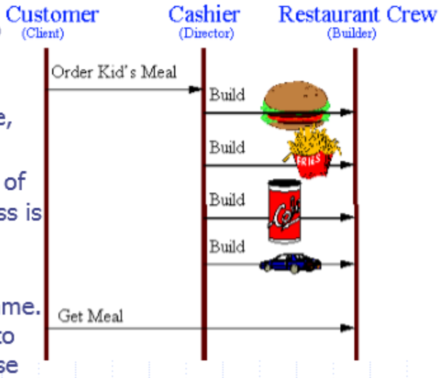
Structure

- The Builder pattern is based on Directors and Builders. Any number of Builder classes can conform to an IBuilder interface, and they can be called by a director to produce a product according to specification. The builders supply parts that the Product objects accumulate until the director is finished with the job. Suppose a Director wants two of one kind of part and one of another. It would request them from a Builder and pass them on to be added to the product's list. The point about the Builder pattern is that different builders can supply different (though conforming) parts.



Example

- The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.
- This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy car).
- Note that there can be variation in the content of the children's meal, but the construction process is the same.
- Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



Creational Patterns: Builder

Problem

- An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Creational Patterns: Builder

Discussion

- Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.
- The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".
- Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".

Creational Patterns: Builder

Rules of Thumb

- Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.
- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

Rules of Thumb (Cont…)

- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Known Uses

- Use the Builder pattern when…
  - The algorithm for creating parts is independent from the parts themselves.
  - The object to be assembled might have different representations.
  - You need fine control over the construction process.

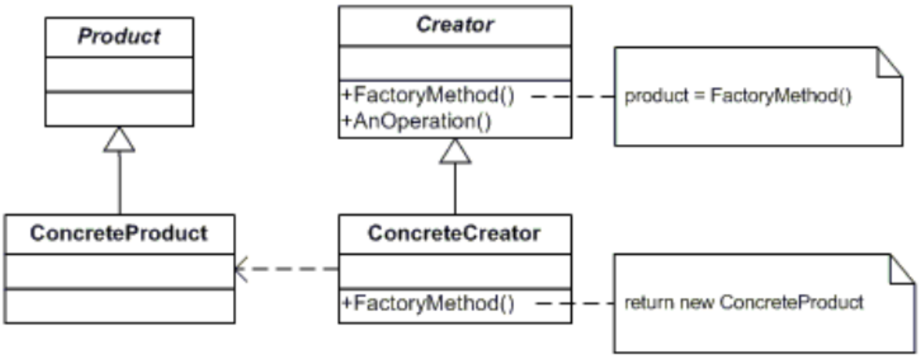
Creational Patterns: Builder

Pattern Comparison

- The Builder and Abstract Factory patterns are similar in that they both look at construction at an abstract level. However, the Builder pattern is concerned with how a single object is made up by the different factories, whereas the Abstract Factory pattern is concerned with what products are made. The Builder pattern abstracts the algorithm for construction by including the concept of a director. The director is responsible for itemizing the steps and calls on builders to fulfill them. Directors do not have to conform to an interface.
- A further elaboration on the theme of creating products is that instead of the client explicitly declaring fields of type ProductA and ProductB, say, the Product object the builder returns is actually a list of parts, which can have different lengths and contents depending on the director that was in charge at its creation.

Creational Patterns: Factory Method

UML: *Factory Method*

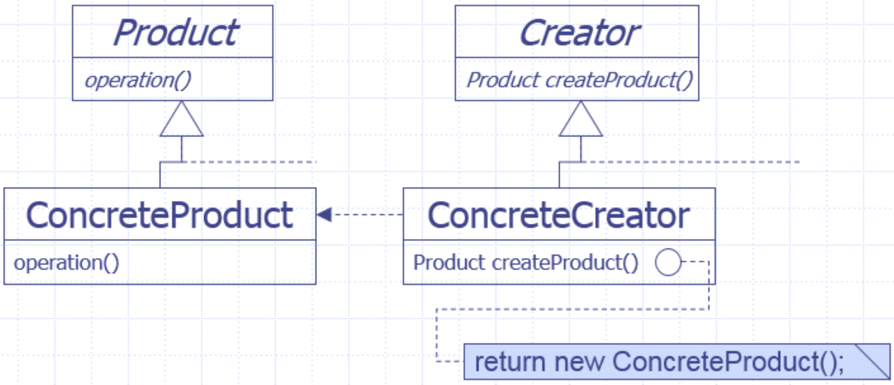


Frequency of use: 1 2 3 4 5 high

Creational Patterns: Factory Method

Intent

- Defer object instantiation to subclasses
- Eliminates binding of application-specific subclasses
- Connects parallel class hierarchies
- Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



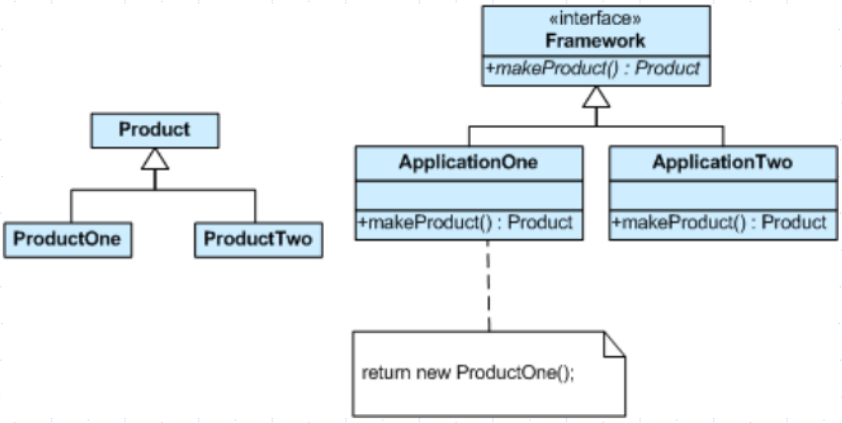
Creational Patterns: Factory Method

Role

- The Factory Method pattern is a way of creating objects, but letting subclasses decide exactly which class to instantiate.
- Various subclasses might implement the interface; the Factory Method instantiates the appropriate subclass based on information supplied by the client or extracted from the current state.

Structure

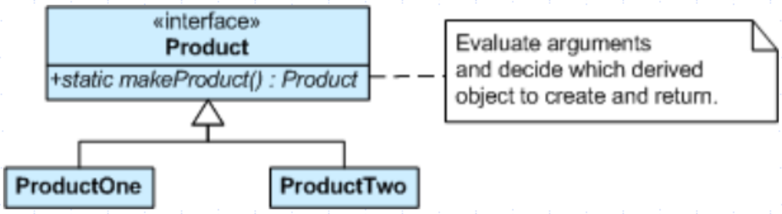
- The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation here focuses on the approach that has become popular since.



Creational Patterns: Factory Method

Structure (Cont…)

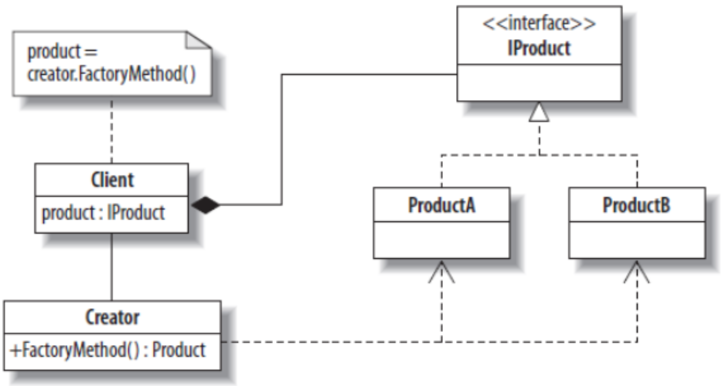
- An increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names (e.g. Color.make_RGB_color(float red, float green, float blue) and Color.make_HSB_color(float hue, float saturation, float brightness)



Creational Patterns: Factory Method

Structure (Cont…)

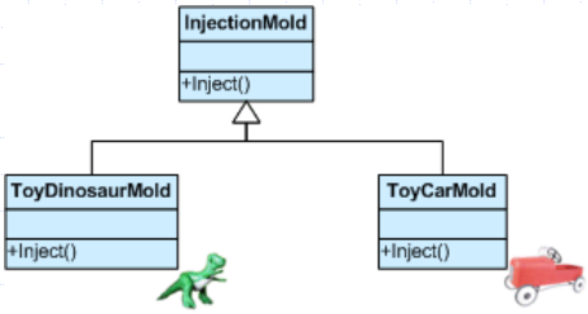
- The client is totally decoupled from the implementation details of derived classes. Polymorphic Creation is now possible.



Creational Patterns: Factory Method

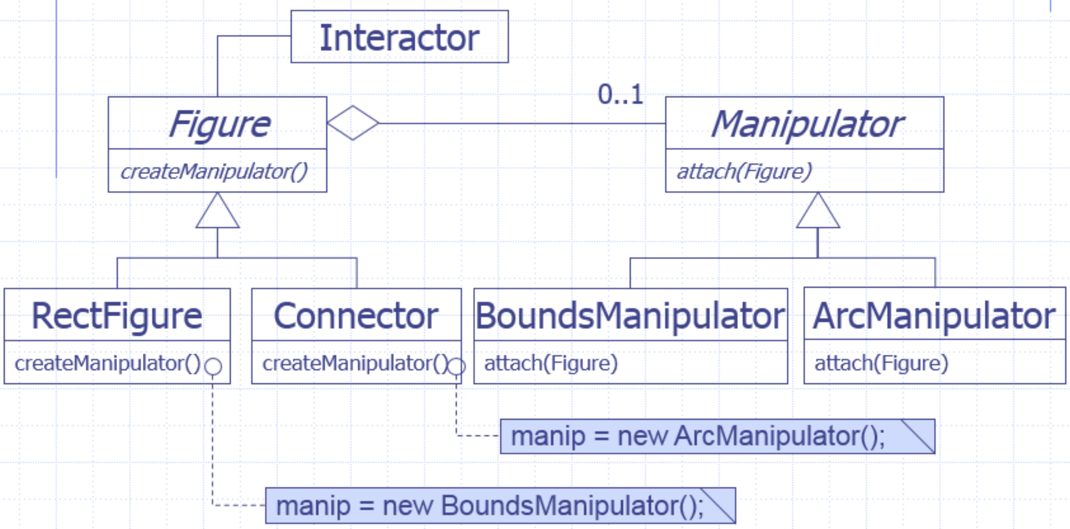
Example

- The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Example (Cont…)

- Creating manipulators on connectors:



Problem

- A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Creational Patterns: Factory Method

Discussion

- Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.
- Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.
- People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).
- Factory Method is similar to Abstract Factory but without the emphasis on families.

Creational Patterns: Factory Method

Rules of Thumb

- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Factory Methods are usually called within Template Methods.
- Factory Method: creation through inheritance. Prototype: creation through delegation.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Creational Patterns: Factory Method

Rules of Thumb (Cont…)

- Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize.
- The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.
- Some Factory Method advocates recommend that as a matter of language design (or failing that, as a matter of style) absolutely all constructors should be private or protected. It's no one else's business whether a class manufactures a new object or recycles an old one.

Creational Patterns: Factory Method

Rules of Thumb (Cont…)

- The new operator considered harmful. There is a difference between requesting an object and creating one.
- The new operator always creates an object, and fails to encapsulate object creation.
- A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

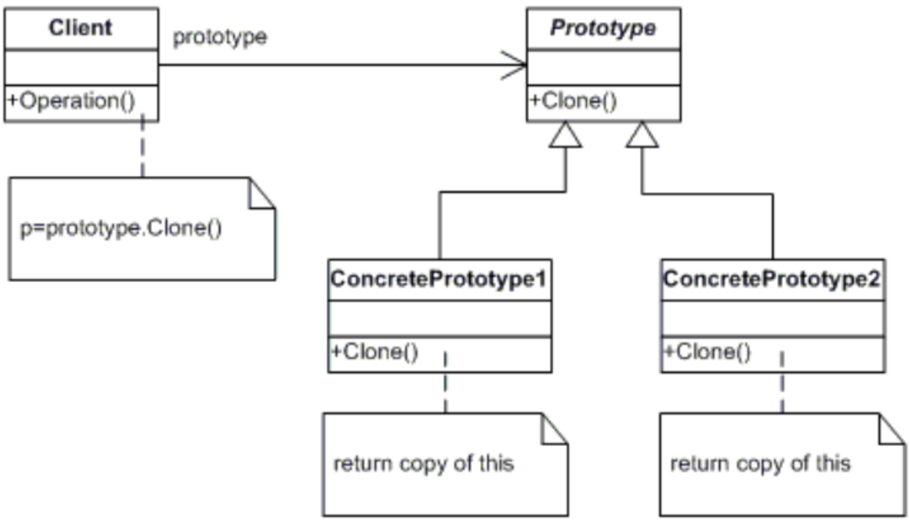
Creational Patterns: Factory Method

Known Uses

- Use the Factory Method pattern when…
  - Flexibility is important.
  - Objects can be extended in subclasses
  - There is a specific reason why one subclass would be chosen over another—this logic forms part of the Factory Method.
  - A client delegates responsibilities to subclasses in parallel hierarchies.

Creational Patterns: Prototype

UML: *Prototype*



Frequency of use: 1 2 3 4 5 medium

Creational Patterns: Prototype

Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The new operator considered harmful.

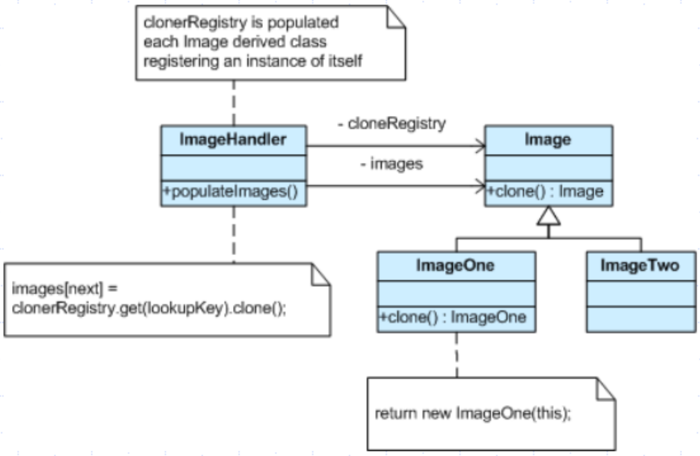
Creational Patterns: Prototype

Role

- The Prototype pattern creates new objects by cloning one of a few stored prototypes.
- The Prototype pattern has two advantages:
  - It speeds up the instantiation of very large, dynamically loaded classes (when copying objects is faster)
  - It keeps a record of identifiable parts of a large data structure that can be copied without knowing the subclass from which they were created.

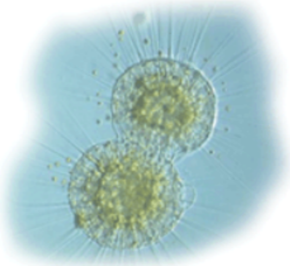
Structure

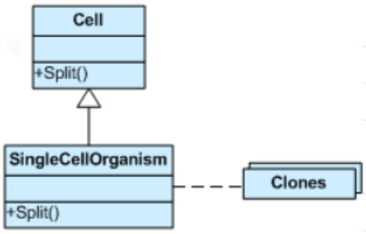
- The Factory knows how to find the correct Prototype, and each Product knows how to spawn new instances of itself.



Example

- The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself. The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotvpe result. In other words, the cell clones itself.





Problem

- Application "hard wires" the class of object to create in each "new" expression.

Creational Patterns: Prototype

Discussion

- Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.
- The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

Creational Patterns: Prototype

Rules of Thumb

- Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.
- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Factory Method: creation through inheritance. Protoype: creation through delegation.

Creational Patterns: Prototype

Rules of Thumb (Cont…)

- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Protoype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require Initialize.
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.
- Prototype co-opts one instance of a class for use as a breeder of all future instances.

Creational Patterns: Prototype

Rules of Thumbs (Cont…)

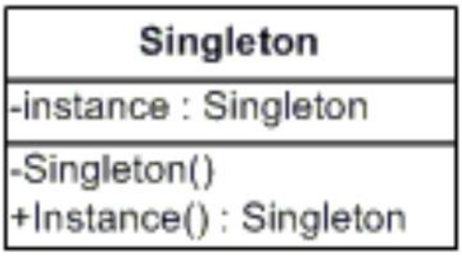
- Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype.
- Prototype is unique among the other creational patterns in that it doesn't require a class – only an object. Object-oriented languages like Self and Omega that do away with classes completely rely on prototypes for creating new objects.

Creational Patterns: Prototype

Known Uses

- Use Prototype pattern when you want to:
  - Hide concrete classes from the client.
  - Add and remove new classes (via prototypes) at runtime.
  - Keep the number of classes in the system to a minimum.
  - Adapt to changing structures of data at runtime.

UML: *Singleton*



Frequency of use: 1 2 3 4 5 medium high

Intent

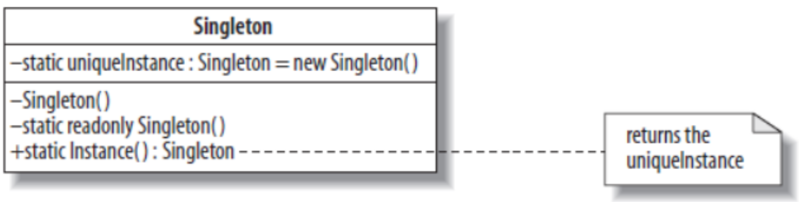
- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Role

- The purpose of the Singleton pattern is to ensure that there is only one instance of a class, and that there is a global access point to that object.
- The pattern ensures that the class is instantiated only once and that all requests are directed to that one and only object. Moreover, the object should not be created until it is actually needed.
- In the Singleton pattern, it is the class itself that is responsible for ensuring this constraint, not the clients of the class.

Structure

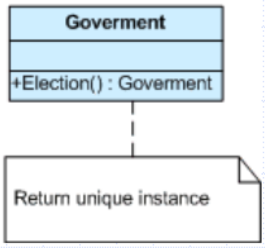
- The Singleton pattern adds functionality by modifying an existing class. The modifications required are:
  - Make the constructor private and add a private static constructor as well.
  - Add a private static read-only object that is internally instantiated using the private constructor.
  - Add a public static property that accesses the private object.



Creational Patterns: Singleton

Example

- The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



Creational Patterns: Singleton

Problem

- Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion

- Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.
- The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.
- Singleton should be considered only if all three of the following criteria are satisfied:
  - Ownership of the single instance cannot be reasonably assigned
  - Lazy initialization is desirable
  - Global access is not otherwise provided for

Discussion (Cont…)

- If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.
- The Singleton pattern can be extended to support access to an application-specific number of instances.
- The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.
- Deleting a Singleton class/instance is a non-trivial design problem. See "To Kill A Singleton" by John Vlissides for a discussion

Creational Patterns: Singleton

Rules of Thumb

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Façade objects are often Singletons because only one Façade object is required.
- State objects are often Singletons.
- The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances.

Creational Patterns: Singleton

Rules of Thumb (Cont…)

- The Singleton design pattern is one of the most inappropriately used patterns. Singletons are intended to be used when a class must have exactly one instance, no more, no less.
- Designers frequently use Singletons in a misguided attempt to replace global variables.
- A Singleton is, for intents and purposes, a global variable. The Singleton does not do away with the global; it merely renames it.

Creational Patterns: Singleton

Rules of Thumb (Cont…)

- When is Singleton unnecessary? Short answer: most of the time. Long answer: when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally. The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an object. Finding the right balance of exposure and protection for an object is critical for maintaining flexibility.
- Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" – even though this and that are the same thing.

Creational Patterns: Singleton

Known Uses

- Use the Singleton pattern when …
  - You need to ensure there is only one instance of a class.
  - Controlled access to that instance is essential.
  - You might need more than one instance at a later stage.
  - The control should be localized in the instantiated class, not in some other mechanism.

Day 3: Structural Patterns

C.K.Leng

Structural Patterns

Purposes

- In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

Structural Patterns

Patterns

- **Adapter:** Match interfaces of different classes
- **Bridge:** Separates an object's interface from its implementation
- **Composite:** A tree structure of simple and composite objects
- **Decorator:** Add responsibilities to objects dynamically
- **Façade:** A single class that represents an entire subsystem
- **Flyweight:** A fine-grained instance used for efficient sharing
- **Proxy:** An object representing another object
- **Private Class Data:** Restricts accessor/mutator access

Structural Patterns

Comparison

- Many patterns are structurally similar, if not identical. What you need to understand is **where**, **how**, and **when** to use it.

Structural Patterns

Comparison (Cont...)

- For example (Proxy Vs Façade):
  - Structurally they are similar where you have the proxy / Façade object in the front talking with the real [domain|implementataion|whatever] object at the back. This similarity also shared with Adaptor and Decorator pattern.
  - **Proxy:** typically to provide specific functionality that the user should not care or need to know the detail about. Some example: EJB proxy object, Spring transaction object, some of the AOP implementation use of proxy.
  - **Façade:** On the other hand, Façade is providing a different front or direct interface to the user. The purpose is to give a consistent or easier API for user to use without knowing the specific of how the overall implementation in the back end is handled.

Structural Patterns

Comparison (Cont...)

- For example: (Adapter Vs Bride)
  - **Adapter:** is used when two incompatible interfaces have to be unified together, ie., adapter is a result of existing incompatibilities.
  - **Bride:** is something we use when we actually need to separate interface from implementation. Varying types of implementation is one reason.

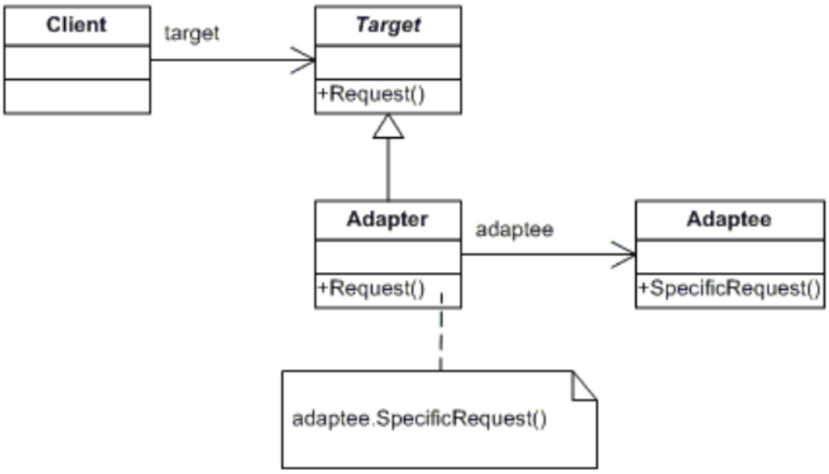
Structural Patterns

Comparison (Cont…)

- For example: (Adapter Vs Proxy)
  - The proxy pattern is very similar in concept to the adapter pattern - it provides a common API for multiple objects which could be varying in nature.
  - In general, the difference between the proxy and adapter pattern is you design your proxy first, the intention from the start being all client objects will use only the proxy API.

Structural Patterns: Adapter

UML: *Adapter*



Frequency of use: 1 2 3 4 5 medium high

Structural Patterns: 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.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

Structural Patterns: Adapter

Role

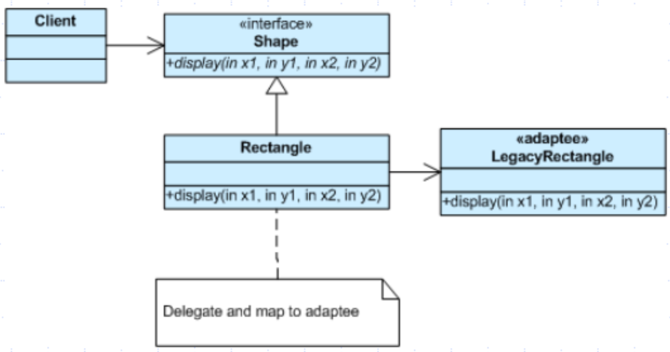
- The Adapter pattern enables a system to use classes whose interfaces don't quite match its requirements. It is especially useful for off-the-shelf code, for toolkits, and for libraries.
- Many examples of the Adapter pattern involve input/output because that is one domain that is constantly changing. For example, programs written in the 1980s will have very different user interfaces from those written in the 2000s. Being able to adapt those parts of the system to new hardware facilities would be much more cost effective than rewriting them.

Role (Cont…)

- Toolkits also need adapters. Although they are designed for reuse, not all applications will want to use the interfaces that toolkits provide; some might prefer to stick to a well-known, domain-specific interface. In such cases, the adapter can accept calls from the application and transform them into calls on toolkit methods.

Structure

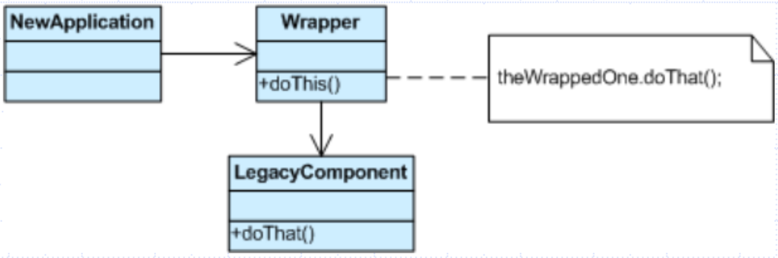
- Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.



Structural Patterns: Adapter

Structure (Cont…)

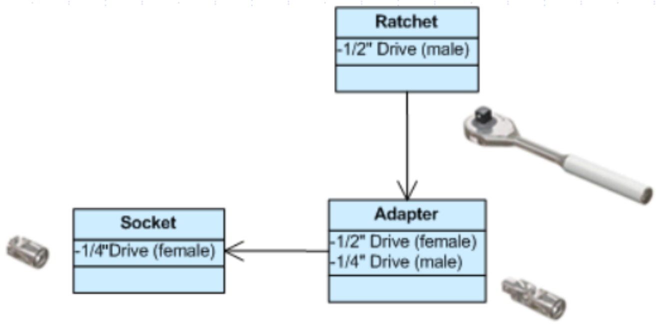
- The Adapter could also be thought of as a "wrapper".



Structural Patterns: Adapter

Example

- The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



Problem

- An "off the shelf" comp compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

- Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.
- It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.
- Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.
- Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Rules of Thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

Structural Patterns: Adapter

Rules of Thumb (Cont…)

- Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
- Façade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

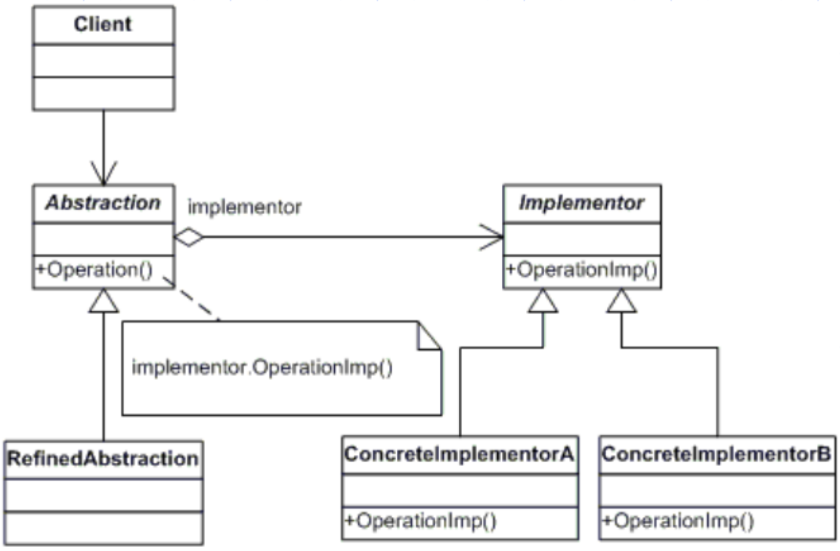
Structural Patterns: Adapter

Known Uses

- Use the Adapter pattern when…
  - You have:
    - A domain-specific interface.
    - A class to connect to with a mismatching interface.
  - You want to:
    - Create a reusable class to cooperate with yet-to-be-built classes.
    - Change the names of methods as called and as implemented.

Structural Patterns: Bridge

UML: *Bridge*



Frequency of use: 1 2 3 4 5 medium

Structural Patterns: Bridge

Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

Structural Patterns: Bridge

Role

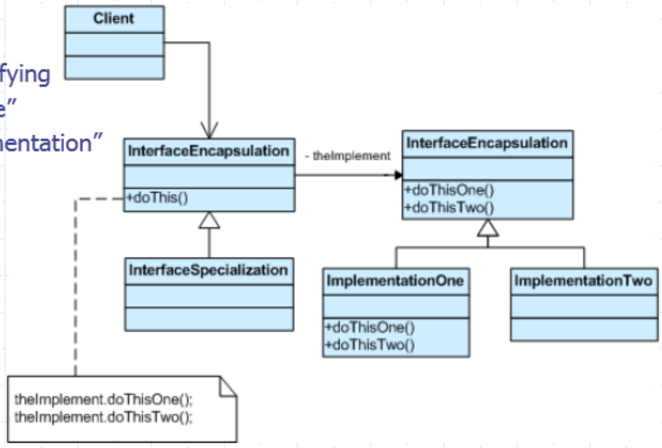
- The Bridge pattern decouples an abstraction from its implementation, enabling them to vary independently.
- The Bridge pattern is useful when a new version of software is brought out that will replace an existing version, but the older version must still run for its existing client base.
- The client code will not have to change, as it is conforming to a given abstraction, but the client will need to indicate which version it wants to use.

Structural Patterns: Bridge

Structure

- The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".

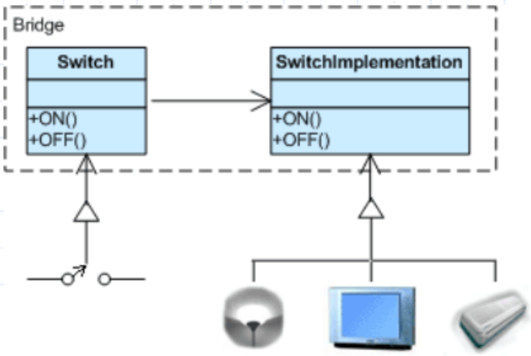
Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction.



Structural Patterns: Bridge

Example

- The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



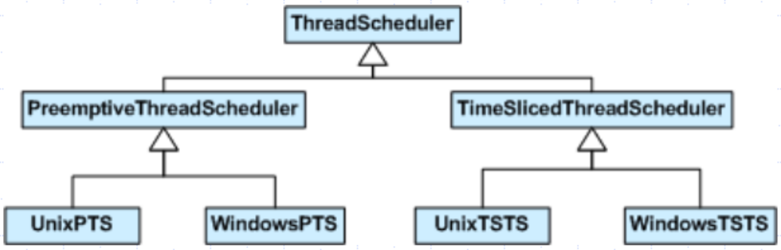
Problem

- "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations.
- This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Structural Patterns: Bridge

Motivation

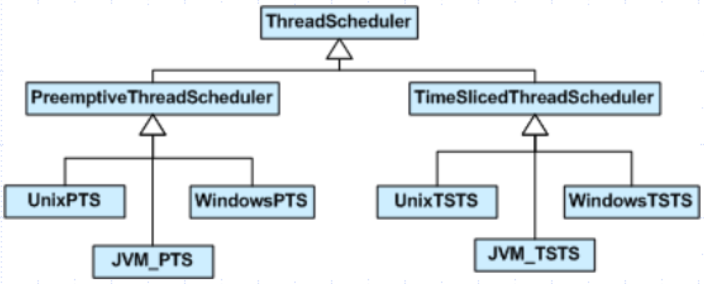
- Consider the domain of "thread scheduling".



There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?

Motivation (Cont…)

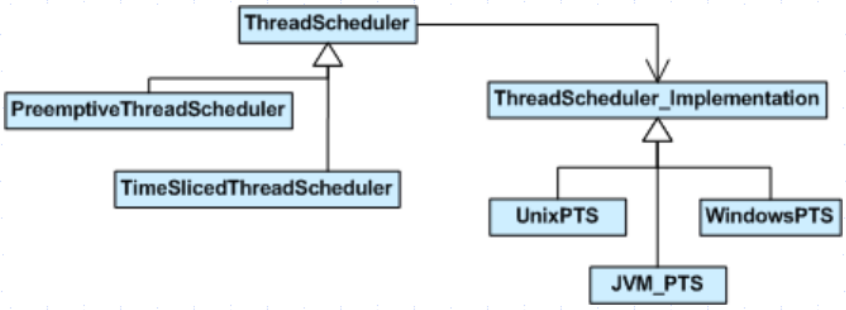
- What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.



Structural Patterns: Bridge

Motivation (Cont…)

- The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



Structural Patterns: Bridge

Discussion

- Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.
- The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.)

Structural Patterns: Bridge

Discussion (Cont…)

- Consequences include:
  - Decoupling the object's interface
  - Improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently)
  - Hiding details from clients

Structural Patterns: Bridge

Discussion (Cont…)

- Bridge is a synonym for the "handle/body" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class.
- The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes.
- The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)."

Structural Patterns: Bridge

Rules of Thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.

Structural Patterns: Bridge

Rules of Thumb (Cont…)

- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

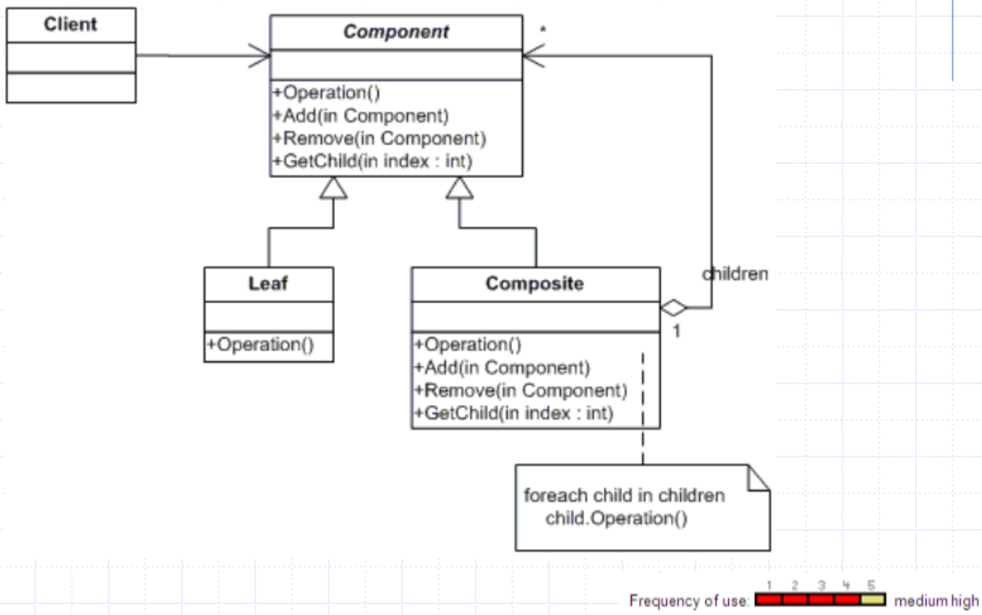
Structural Patterns: Bridge

Known Uses

- Use the Bridge pattern when…
  - You can:
    - Identify that there are operations that do not always need to be implemented in the same way.
  - You want to:
    - Completely hide implementations from clients.
    - Avoid binding an implementation to an abstraction directly.
    - Change an implementation without even recompiling an abstraction.
    - Combine different parts of a system at runtime.
    - you want run-time binding of the implementation.
    - you have a proliferation of classes resulting from a coupled interface and numerous implementations.
    - you want to share an implementation among multiple objects.
    - you need to map orthogonal class hierarchies

Structural Patterns: Composite

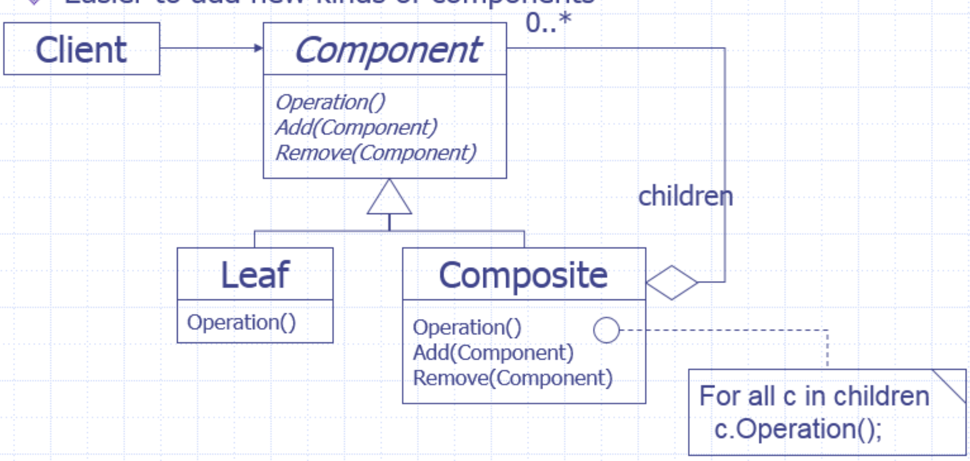
UML: *Composite*



Structural Patterns: Composite

Intent

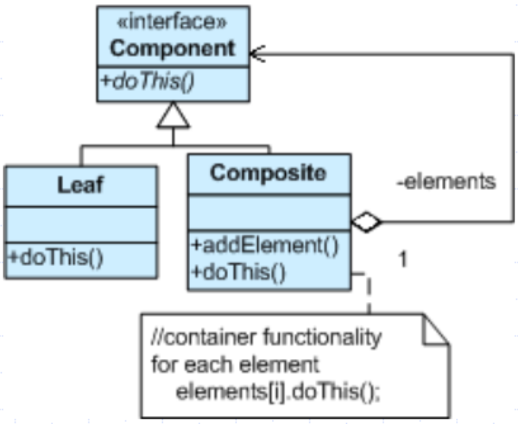
- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- Easier to add new kinds of components



Structural Patterns: Composite

Structure

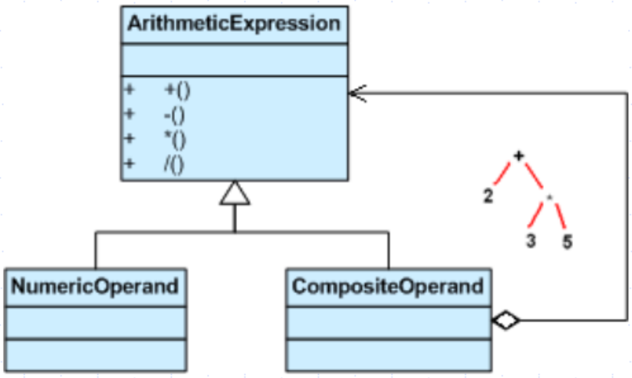
- Composites that contain Components, each of which could be a Composite.
- Menus that contain menu items, each of which could be a menu.
- Row-column GUI layout managers that contain widgets, each of which could be a row-column GUI layout manager.
- Directories that contain files, each of which could be a directory.
- Containers that contain Elements, each of which could be a Container.



Structural Patterns: Composite

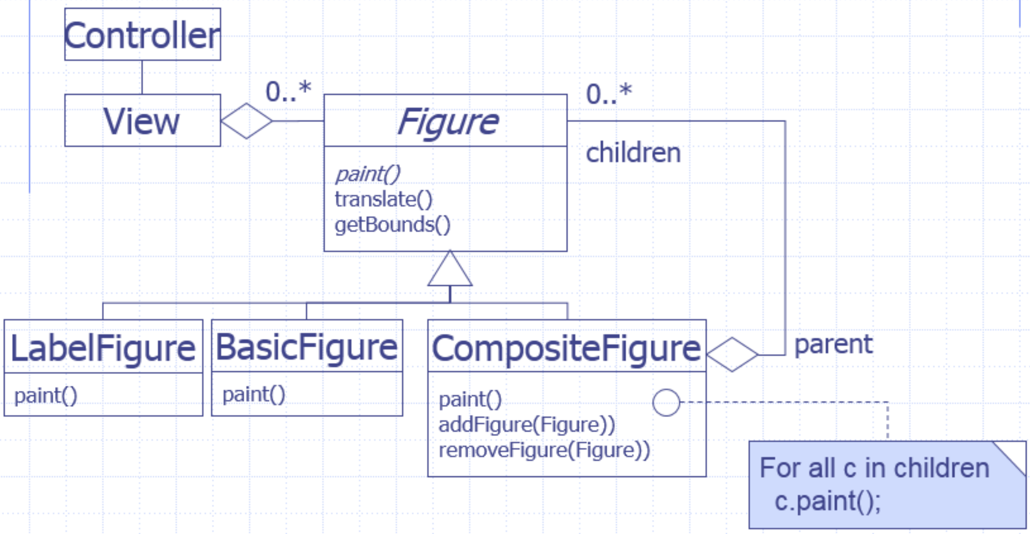
Example

- The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.



Example (Cont…)

- Figures in a structured graphics toolkit:



Structural Patterns: Composite

Role

- The Composite pattern arranges structured hierarchies so that single components and groups of components can be treated in the same way. Typical operations on the components include add, remove, display, find, and group.

Structural Patterns: Composite

Problem

- Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Structural Patterns: Composite

Discussion

- Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".
- Use this pattern whenever you have "composites that contain components, each of which could be a composite".
- Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the "Opinions" section below for a discussion of "safety" versus "transparency" issues.

Structural Patterns: Composite

Rules of Thumb

- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.

Structural Patterns: Composite

Rules of Thumb (Cont…)

- Composite can let you compose a Mediator out of smaller pieces through recursive composition.
- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Flyweight is often combined with Composite to implement shared leaf nodes.

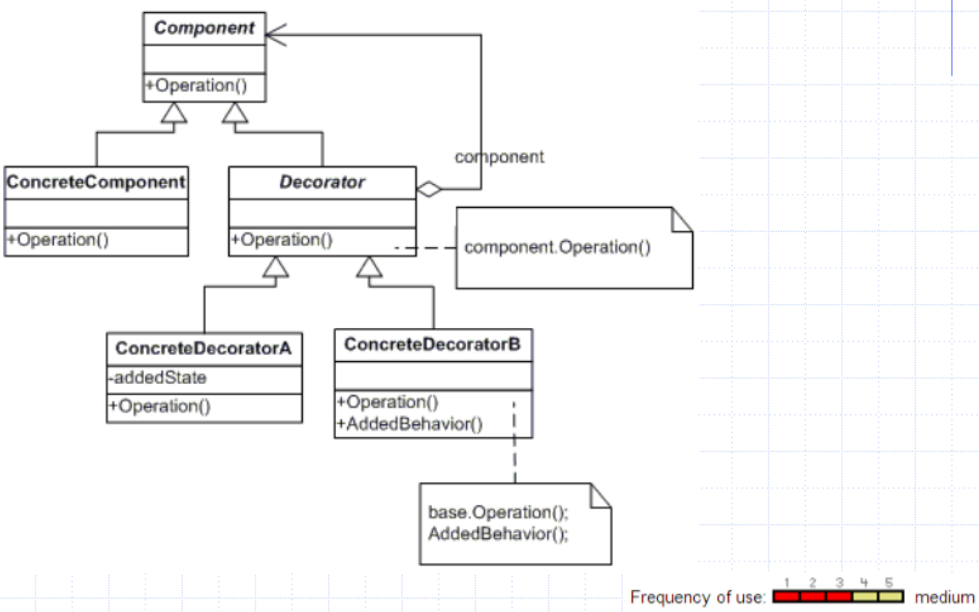
Structural Patterns: Composite

Known Uses

- Use the Composite pattern when…
  - You have:
    - An irregular structure of objects and composites of the objects
  - You want:
    - Clients to ignore all but the essential differences between individual objects and composites of objects
    - To treat all objects in a composite uniformly

Structural Patterns: Decorator

UML: *Decorator*



Structural Patterns: Decorator

Intent

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

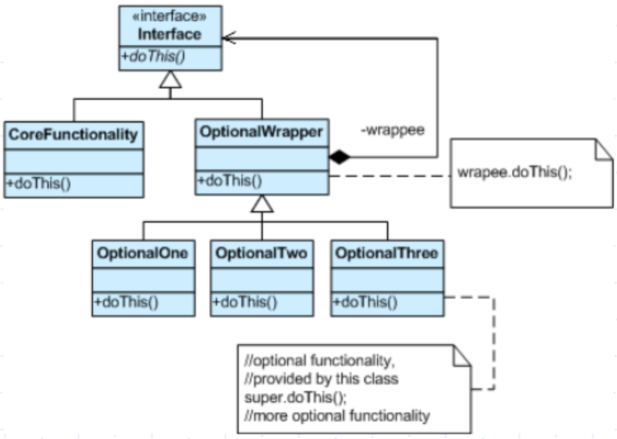
Structural Patterns: Decorator

Role

- The role of the Decorator pattern is to provide a way of attaching new state and behavior to an object dynamically. The object does not know it is being "decorated," which makes this a useful pattern for evolving systems. A key implementation point in the Decorator pattern is that decorators both inherit the original class and contain an instantiation of it.

Structure

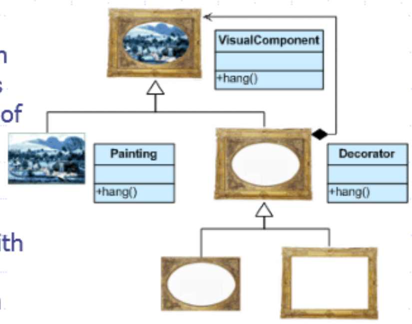
- The client is always interested in CoreFunctionality.doThis(). The client may, or may not, be interested in OptionalOne.doThis() and OptionalTwo.doThis(). Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.



Structural Patterns: Decorator

Example

- The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.



Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.

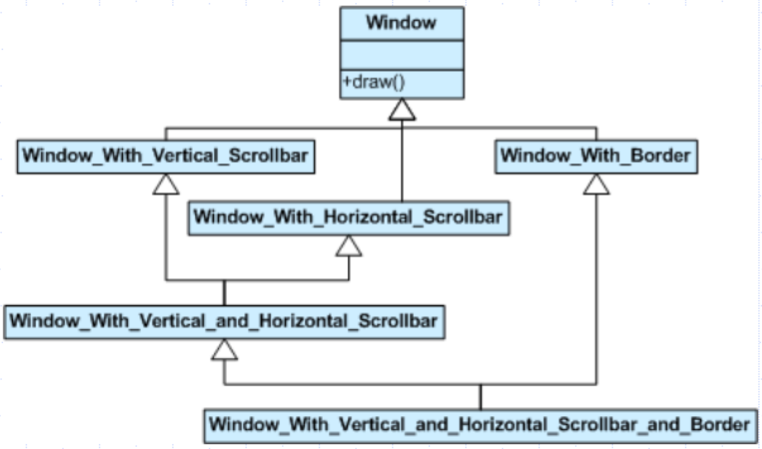
Problem

- You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Structural Patterns: Decorator

Discussion

- Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like …



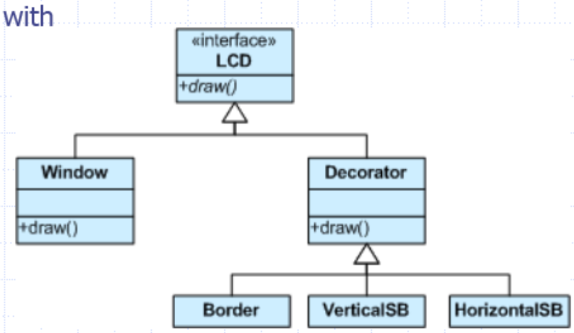
Structural Patterns: Decorator

Discussion (Cont...)

- But the Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired.

```
Widget*  aWidget = new BorderDecorator(
                  new HorizontalScrollBarDecorator(
                   new VerticalScrollBarDecorator(
                    new Window( 80, 24 ))));
aWidget->draw();
```

This flexibility can be achieved with the following design



Structural Patterns: Decorator

Discussion (Cont...)

- Another example of cascading (or chaining) features together to produce a custom object might look like ...

```
Stream*  aStream = new CompressingStream(
                  new ASCII7Stream(
                   new FileStream( "fileName.dat" )));
aStream->putString( "Hello world" );
```

- The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.
- Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.
- Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Rules of Thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters
- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.

Rules of Thumb (Cont...)

- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.
- Decorator lets you change the skin of an object. Strategy lets you change the guts.

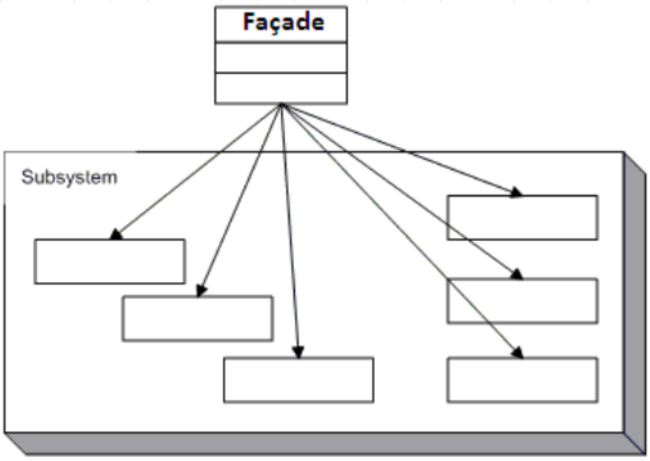
Structural Patterns: Decorator

Known Uses

- Use the Decorator pattern when…
  - You have:
    - An existing component class that may be unavailable for subclassing.
  - You want to:
    - Attach additional state or behavior to an object dynamically.
    - Make changes to some objects in a class without affecting others.
    - Avoid subclassing because too many classes could result.

Structural Patterns: Façade

UML: *Façade*

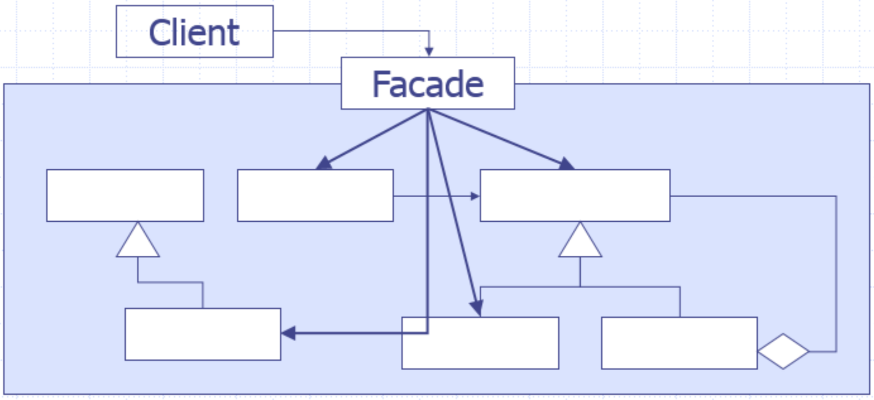


Frequency of use: 1 2 3 4 5 high

Structural Patterns: Façade

Intent

- Provide unified interface to interfaces within a subsystem
- Shield clients from subsystem components
- Promote weak coupling between client and subsystem components



Structural Patterns: Façade

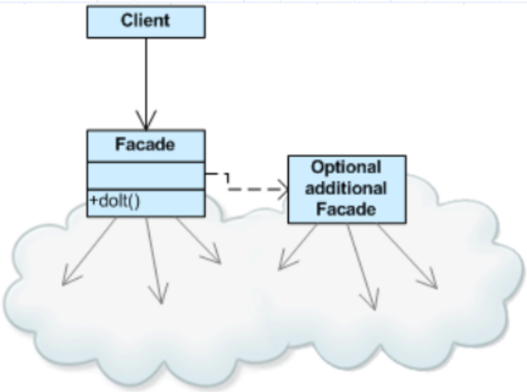
Role

- The role of the Façade pattern is to provide different high-level views of subsystems whose details are hidden from users. In general, the operations that might be desirable from a user's perspective could be made up of different selections of parts of the subsystems.

Structural Patterns: Façade

Structure

- Façade takes a "riddle wrapped in an enigma shrouded in mystery", and interjects a wrapper that tames the amorphous and inscrutable mass of software.

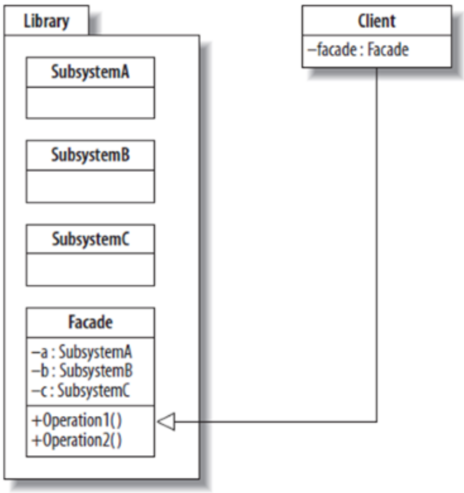


Structural Patterns: Façade

Structure (Cont…)

- Hiding detail is a key programming concept. What makes the Façade pattern different from, say, the Decorator or Adapter patterns is that the interface it builds up can be entirely new. It is not coupled to existing requirements, nor must it conform to existing interfaces. There can also be several façades built uparound an existing set of subsystems.

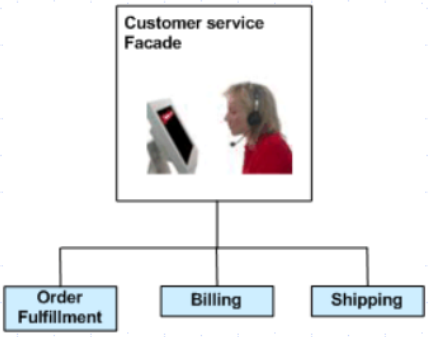
  The term "subsystem" is used here deliberately; we are talking at a higher level than classes. See the UML diagram in Figure at the right; it considers the subsystems to be grouped together, so they can interact in agreed ways to form the top-level operations.



Structural Patterns: Façade

Example

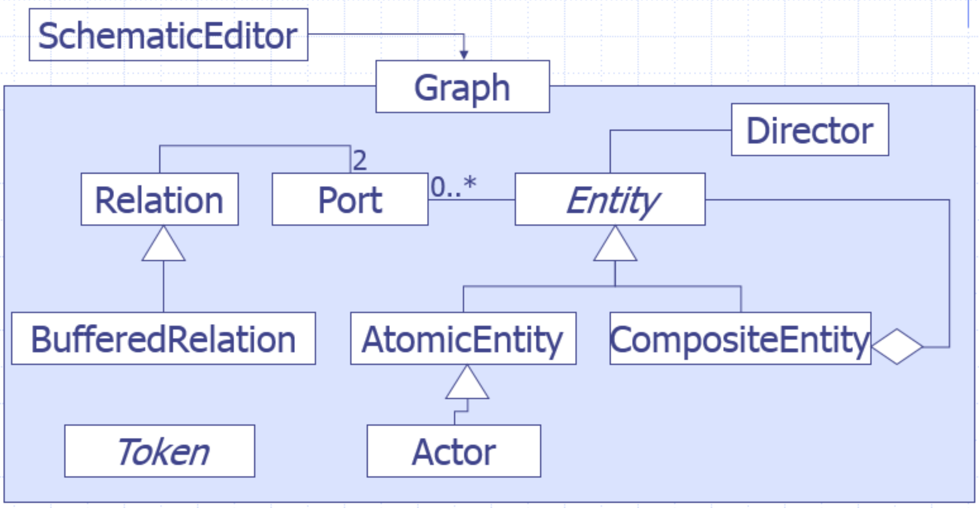
- The Façade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Façade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Façade, providing an interface to the order fulfillment department, the billing department, and the shipping department.



Structural Patterns: Façade

Example (Cont…)

- Graph interface to a simulation engine:



Structural Patterns: Façade

Problem

- A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Structural Patterns: Façade

Discussion

- Façade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Façade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.
- The Façade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Structural Patterns: Façade

Rules of Thumb

- Façade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
- Whereas Flyweight shows how to make lots of little objects, Façade shows how to make a single object represent an entire subsystem.
- Mediator is similar to Façade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Façade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.

Structural Patterns: Façade

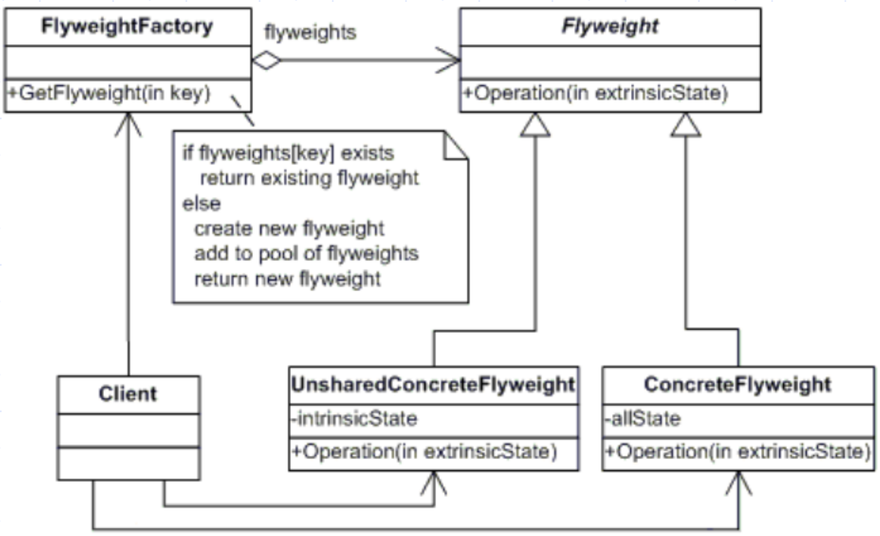
Rules of Thumb (Cont…)

- Abstract Factory can be used as an alternative to Façade to hide platform-specific classes.
- Façade objects are often Singletons because only one Façade object is required.
- Adapter and Façade are both wrappers; but they are different kinds of wrappers. The intent of Façade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface. While Façade routinely wraps multiple objects and Adapter wraps a single object; Façade could front-end a single complex object and Adapter could wrap several legacy objects.

Known Uses

- Use the Façade pattern when…
  - A system has several identifiable subsystems and:
    - The abstractions and implementations of a subsystem are tightly coupled.
    - The system evolves and gets more complex, but early adopters might want to retain their simple views.
    - You want to provide alternative novice, intermediate, and "power user" interfaces.
    - There is a need for an entry point to each level of layered software.
  - Choose the Façade you need…
    - Opaque - Subsystem operations can only be called through the Façade.
    - Transparent - Subsystem operations can be called directly as well as through the Façade.
    - Singleton - Only one instance of the Façade is meaningful.

UML: *Flyweight*



Frequency of use: 1 2 3 4 5 low

Structural Patterns: Flyweight

Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

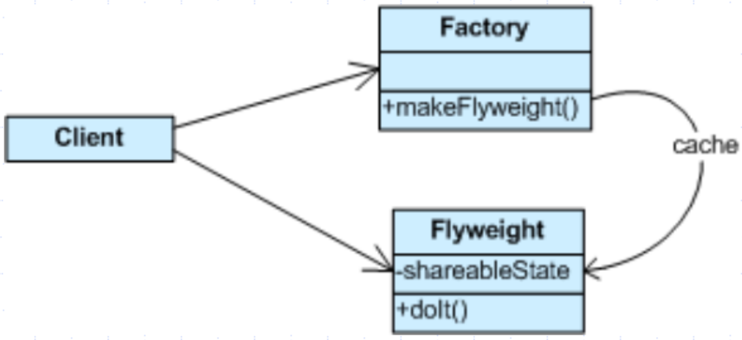
Structural Patterns: Flyweight

Role

- The Flyweight pattern promotes an efficient way to share common information present in small objects that occur in a system in large numbers. It thus helps reduce storage requirements when many values are duplicated.
- The Flyweight pattern distinguishes between the intrinsic and extrinsic state of an object.
- The greatest savings in the Flyweight pattern occur when objects use both kinds of state but:
  - The intrinsic state can be shared on a wide scale, minimizing storage requirements.
  - The extrinsic state can be computed on the fly, trading computation for storage.

Structure

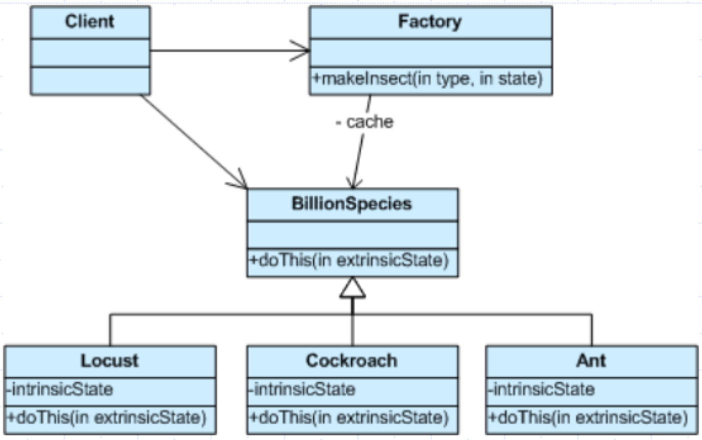
- Flyweights are stored in a Factory's repository. The client restrains herself from creating Flyweights directly, and requests them from the Factory. Each Flyweight cannot stand on its own. Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight. If the context lends itself to "economy of scale" (i.e. the client can easily compute or look-up the necessary attributes), then the Flyweight pattern offers appropriate leverage.



Structural Patterns: Flyweight

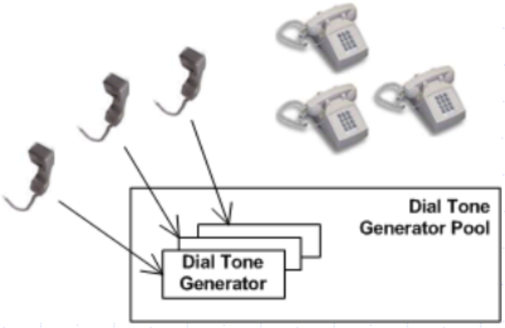
Structure (Cont…)

- The Ant, Locust, and Cockroach classes can be "light-weight" because their instance-specific state has been de-encapsulated, or externalized, and must be supplied by the client.



Example

- The Flyweight uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call. All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed.



Structural Patterns: Flyweight

Problem

- Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

Structural Patterns: Flyweight

Discussion

- The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.
- An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

Structural Patterns: Flyweight

Rules of Thumb

- Whereas Flyweight shows how to make lots of little objects, Façade shows how to make a single object represent an entire subsystem.
- Flyweight is often combined with Composite to implement shared leaf nodes.
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- Flyweight explains when and how State objects can be shared.

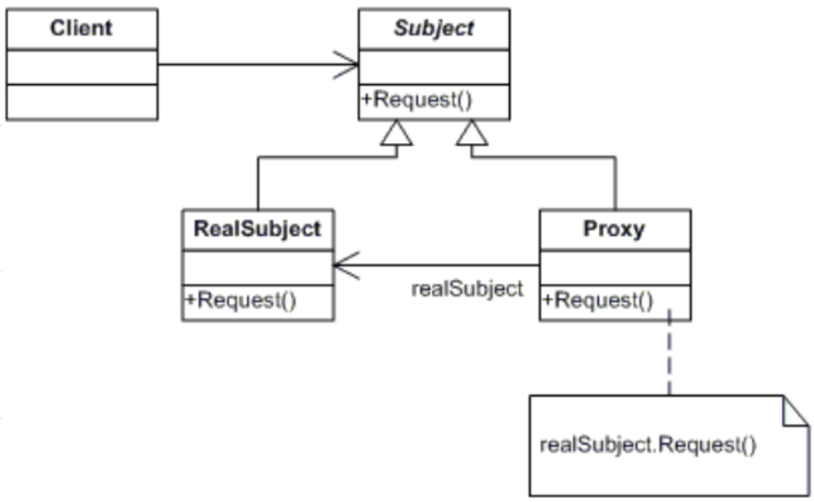
Structural Patterns: Flyweight

Known Uses

- Use the Flyweight pattern when…
  - There are:
    - Many objects to deal with in memory
    - Different kinds of state, which can be handled differently to achieve space savings
    - Groups of objects that share state
    - Ways of computing some of the state at runtime
  - You want to:
    - Implement a system despite severe memory constraints

Structural Patterns: Proxy

UML: *Proxy*



Frequency of use: 1 2 3 4 5 medium high

Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

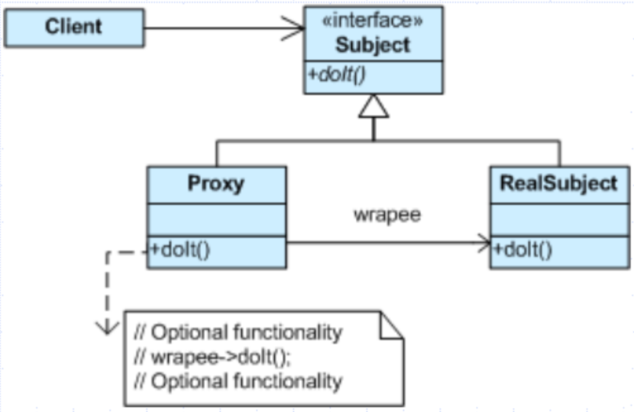
Role

- The Proxy pattern supports objects that control the creation of and access to other objects. The proxy is often a small (public) object that stands in for a more complex (private) object that is activated once certain circumstances are clear.

Structural Patterns: Proxy

Structure

- By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.

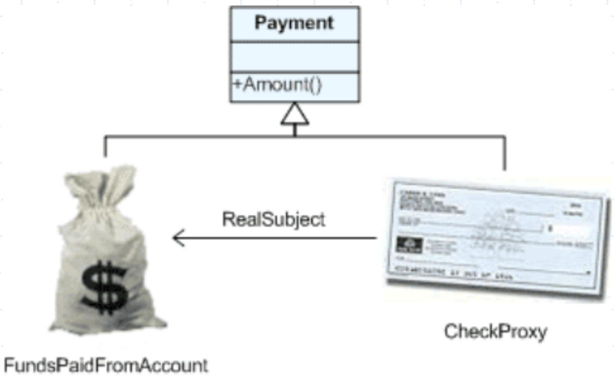


Structural Patterns: Proxy

Example

- ## Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Structural Patterns: Proxy

Problem

- You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Structural Patterns: Proxy

Discussion

- Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

Structural Patterns: Proxy

Discussion (Cont…)

- There are four common situations in which the Proxy pattern is applicable.
  1. A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
  2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
  3. A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
  4. A smart proxy interposes additional actions when an object is accessed. Typical uses include:
     - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
     - Loading a persistent object into memory when it's first referenced,
     - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Structural Patterns: Proxy

Rules of Thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

Known Uses

- Use the Proxy pattern when…
  - You have objects that:
    - Are expensive to create.
    - Need access control.
    - Access remote sites.
    - Need to perform some action whenever they are accessed.
  - You want to:
    - Create objects only when their operations are requested.
    - Perform checks or housekeeping on objects whenever accessed.
    - Have a local object that will refer to a remote object.
    - Implement access rights on objects as their operations are requested.

Day 4 & 5:Behavioral Patterns

C.K.Leng

Behavioral Patterns

Purposes

- In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Behavioral Patterns

Patterns

- **Chain of responsibility:** A way of passing a request between a chain of objects
- **Command:** Encapsulate a command request as an object
- **Interpreter:** A way to include language elements in a program
- **Iterator:** Sequentially access the elements of a collection
- **Mediator:** Defines simplified communication between classes
- **Memento:** Capture and restore an object's internal state
- **Null Object:** Designed to act as a default value of an object

Behavioral Patterns

Patterns (Cont…)

- **Observer:** A way of notifying change to a number of classes
- **State:** Alter an object's behavior when its state changes
- **Strategy:** Encapsulates an algorithm inside a class
- **Template method:** Defer the exact steps of an algorithm to a subclass
- **Visitor:** Defines a new operation to a class without change

Behavioral Patterns

Rules of thumb

- Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it.
- Chain of responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
- Chain of responsibility can use Command to represent requests as objects.
- Chain of responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.
- Command can use Memento to maintain the state required for an undo operation.

Behavioral Patterns

Rules of thumb (Cont…)

- Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.
- MacroCommands can be implemented with Composite.
- A Command that must be copied before being placed on a history list acts as a Prototype.
- Interpreter can use State to define parsing contexts.
- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.

Rules of thumb (Cont…)

- Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.
- Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.
- Mediator is similar to Façade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, Façade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa).

Behavioral Patterns

Rules of thumb (Cont…)

- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.
- State is like Strategy except in its intent.
- Flyweight explains when and how State objects can be shared.
- State objects are often Singletons.
- Strategy lets you change the guts of an object. Decorator lets you change the skin.
- Strategy is to algorithm. as Builder is to creation.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).

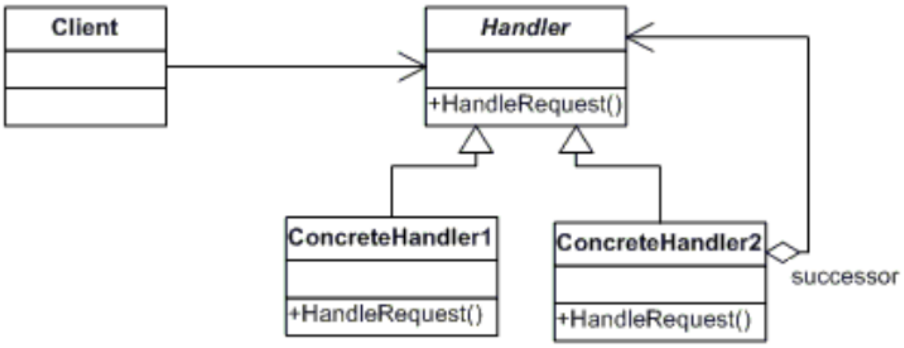
Behavioral Patterns

Rules of thumb (Cont…)

- Strategy objects often make good Flyweights.
- Strategy is like Template method except in its granularity.
- Template method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm.
- The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.

Behavioral Patterns: Chain of Responsibility

UML: *Chain of Responsibility*



Frequency of use: 1 2 3 4 5 medium low

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.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

Behavioral Patterns: Chain of Responsibility

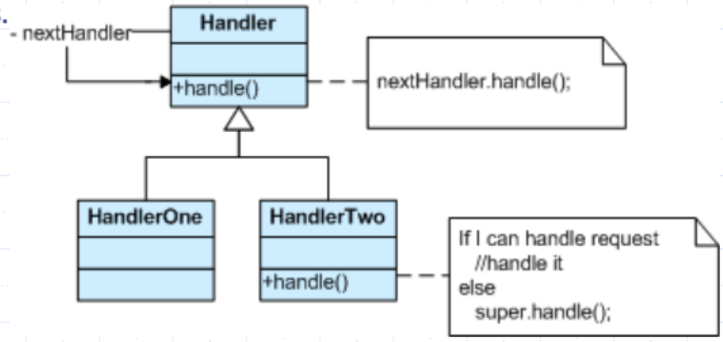
Role

- The Chain of Responsibility pattern works with a list of Handler objects that have limitations on the nature of the requests they can deal with. If an object cannot handle a request, it passes it on to the next object in the chain. At the end of the chain, there can be either default or exceptional behavior.

Behavioral Patterns: Chain of Responsibility

Structure

- The derived classes know how to satisfy Client requests. If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues.



Multiple handlers could contribute to the handling of each request. The request can be passed down the entire length of the chain, with the last link being careful not to delegate to a "null next".

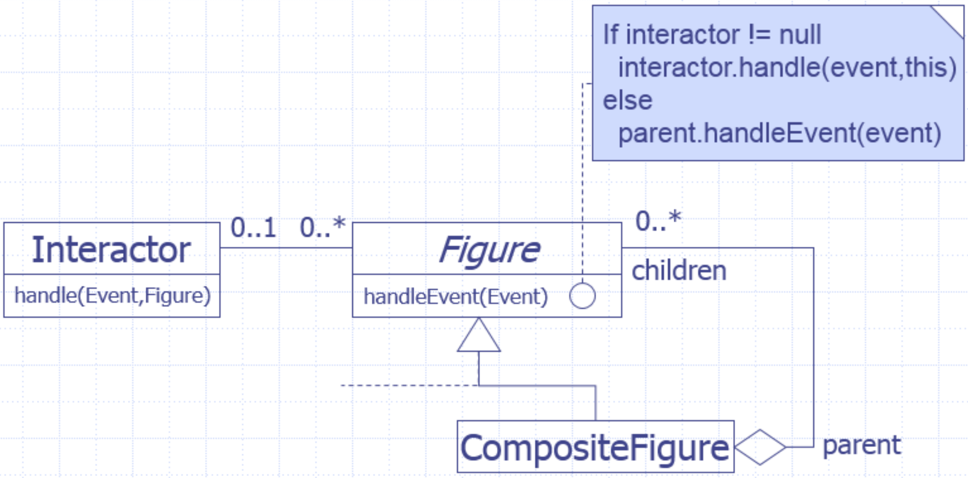
Behavioral Patterns: Chain of Responsibility

Example

- The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.

Example (Cont…)

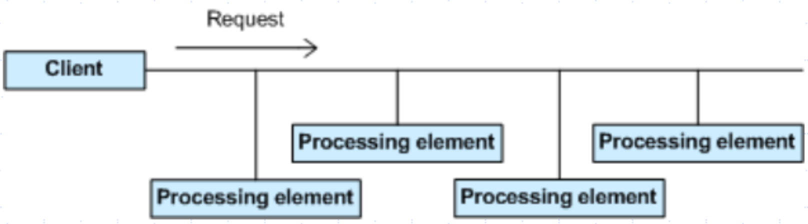
- Handling events in a graphical hierarchy:



Behavioral Patterns: Chain of Responsibility

Problem

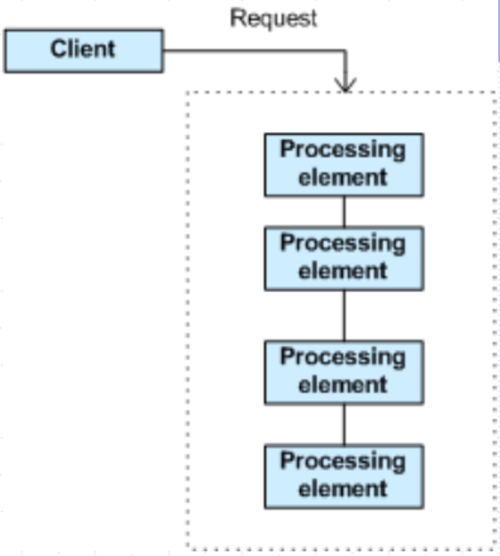
- There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.



Behavioral Patterns: Chain of Responsibility

Discussion

- Encapsulate the processing elements inside a "pipeline" abstraction; and have clients "launch and leave" their requests at the entrance to the pipeline.
- Chain of Responsibility functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.



Behavioral Patterns: Chain of Responsibility

Discussion (Cont…)

- The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.
- Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.
- Make sure there exists a "safety net" to "catch" any requests which go unhandled.
- Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Behavioral Patterns: Chain of Responsibility

Rules of Thumb

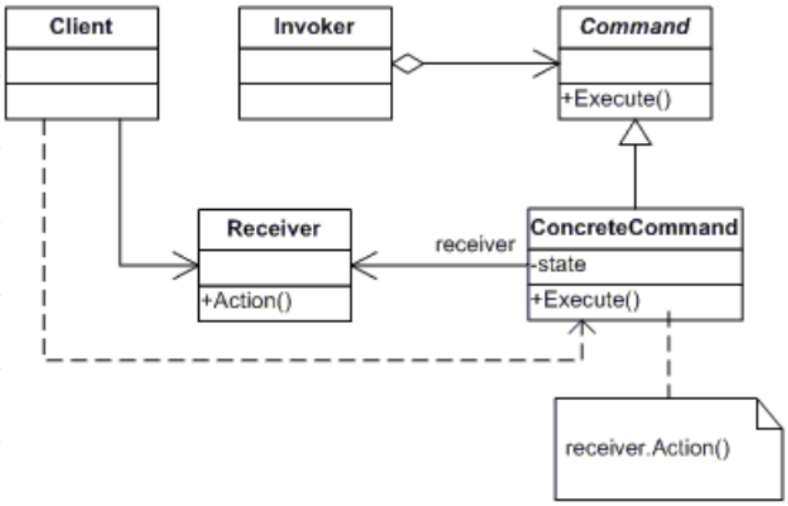
- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.
- Chain of Responsibility can use Command to represent requests as objects.
- Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

Behavioral Patterns: Chain of Responsibility

Known Uses

- Use the Chain of Responsibility pattern when…
  - You have:
    - More than one handler for a request
    - Reasons why a handler should pass a request on to another one in the chain
    - A set of handlers that varies dynamically
  - You want to:
    - Retain flexibility in assigning requests to handlers

UML: *Command*



Frequency of use: 1 2 3 4 5 medium high

Behavioral Patterns: Command

Intent

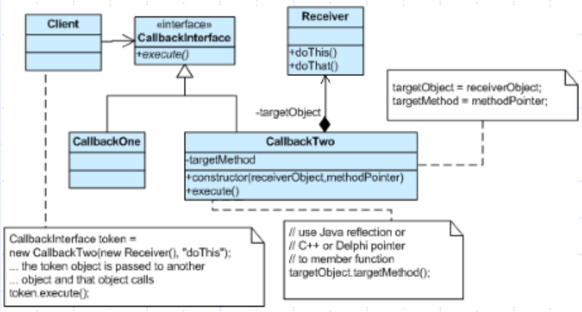
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback.

Role

- The Command pattern creates distance between the client that requests an operation and the object that can perform it. This pattern is particularly versatile. It can support:
  - Sending requests to different receivers
  - Queuing, logging, and rejecting requests
  - Composing higher-level transactions from primitive operations
  - Redo and Undo functionality

Structure

- The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.



Command objects can be thought of as “tokens” that are created by one client that knows what need to be done, and passed to another client that has the resources for doing it.

Behavioral Patterns: Command

Example

- The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.

Behavioral Patterns: Command

Problem

- Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Behavioral Patterns: Command

Discussion

- Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.
- All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".
- A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied. The Command's "execute" method then causes the pieces to come together.
- Sequences of Command objects can be assembled into composite (or macro) commands.

Behavioral Patterns: Command

Rules of Thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.
  - Chain of Responsibility can use Command to represent requests as objects.
  - Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.

Behavioral Patterns: Command

Rules of Thumb (Cont…)

- Command can use Memento to maintain the state required for an undo operation.
- MacroCommands can be implemented with Composite.
- A Command that must be copied before being placed on a history list acts as a Prototype.
- Two important aspects of the Command pattern: interface separation (the invoker is isolated from the receiver), time separation (stores a ready-to-go processing request that's to be stated later).

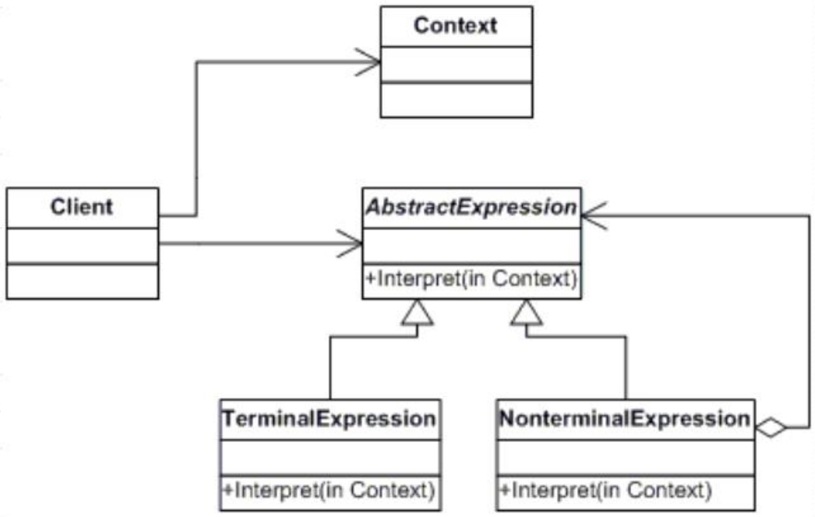
Behavioral Patterns: Command

Known Uses

- Use the Command pattern when…
  - You have:
    - Commands that different receivers can handle in different ways
    - A high-level set of commands that are implemented by primitive operations
  - You want to:
    - Specify, queue, and execute commands at different times
    - Support an Undo function for commands
    - Support auditing and logging of all changes via commands

Behavioral Patterns: Interpreter

UML: *Interpreter*



Frequency of use: 1 2 3 4 5 low

Behavioral Patterns: Interpreter

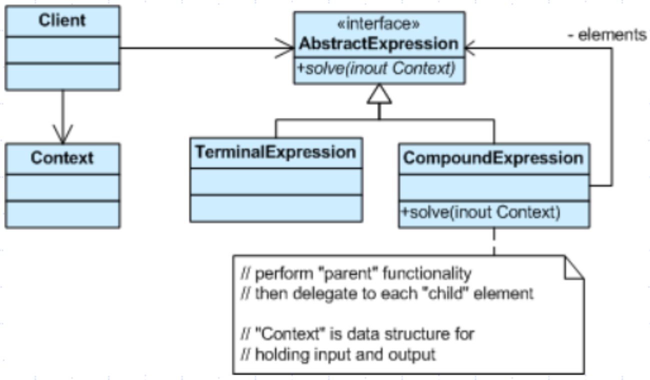
Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

Behavioral Patterns: Interpreter

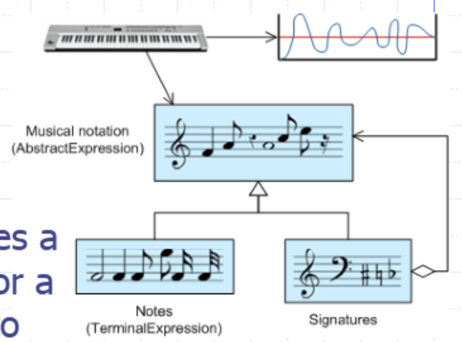
Structure

- Interpreter suggests modeling the domain with a recursive grammar. Each rule in the grammar is either a 'composite' (a rule that references other rules) or a terminal (a leaf node in a tree structure). Interpreter relies on the recursive traversal of the Composite pattern to interpret the 'sentences' it is asked to process.



Behavioral Patterns: Interpreter

Example



- The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.

Problem

- A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

Discussion

- The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.
- An abstract base class specifies the method interpret(). Each concrete subclass implements interpret() by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.

Behavioral Patterns: Interpreter

Rules of Thumb

- Considered in its most general form (i.e. an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of this class hierarchy as defining a language.
- Interpreter can use State to define parsing contexts.
- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- The pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate.

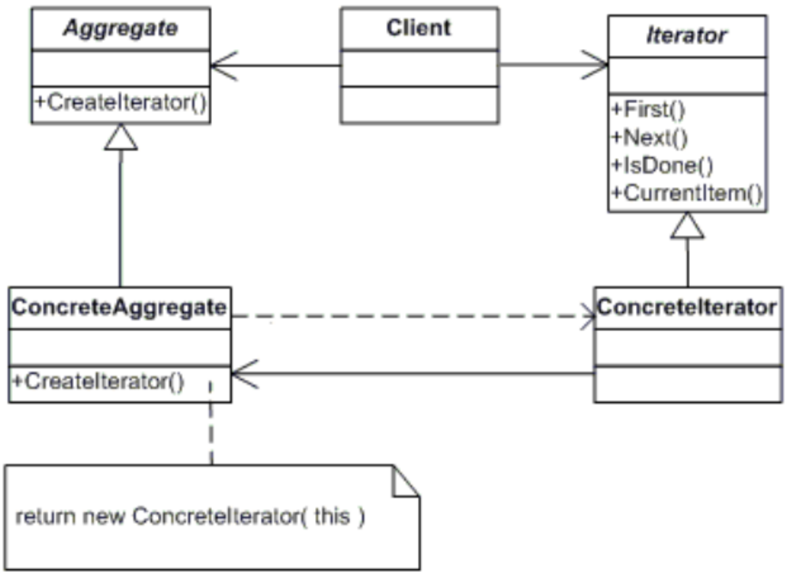
Behavioral Patterns: Interpreter

Known Uses

- Use the Interpreter pattern when…
  - You have a grammar to be interpreted and:
    - The grammar is not too large.
    - Efficiency is not critical.
    - Parsing tools are available.
    - XML is an option for the specification.

Behavioral Patterns: Iterator

UML: *Iterator*



Frequency of use: 1 2 3 4 5 high

Behavioral Patterns: Iterator

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ , Java, and .NET standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

Behavioral Patterns: Iterator

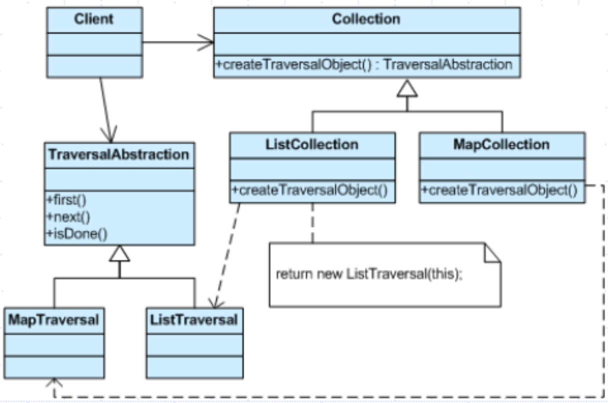
Role

- The Iterator pattern provides a way of accessing elements of a collection sequentially, without knowing how the collection is structured. As an extension, the pattern allows for filtering elements in a variety of ways as they are generated.

Behavioral Patterns: Iterator

Structure

- The Client uses the Collection class' public interface directly. But access to the Collection's elements is encapsulated behind the additional level of abstraction called Iterator. Each Collection derived class knows which Iterator derived class to create and return. After that, the Client relies on the interface defined in the Iterator base class.



Behavioral Patterns: Iterator

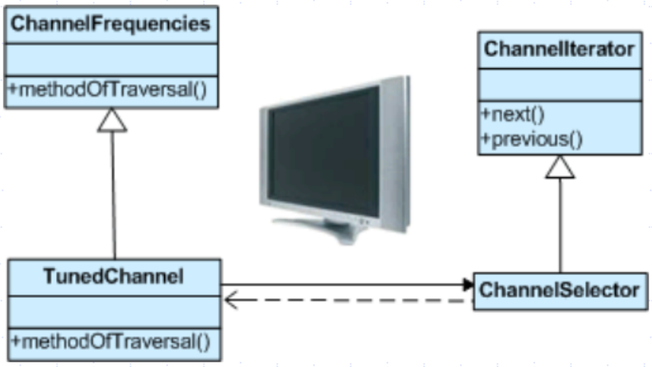
Example

- The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently.
- On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed.

Behavioral Patterns: Iterator

Example (Cont…)

- Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number..



Behavioral Patterns: Iterator

Problem

- Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Behavioral Patterns: Iterator

Discussion

- "An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list." And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.
- The Iterator pattern lets you do all this. The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

Discussion (Cont…)

- The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.
- As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items.

Rules of Thumb

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

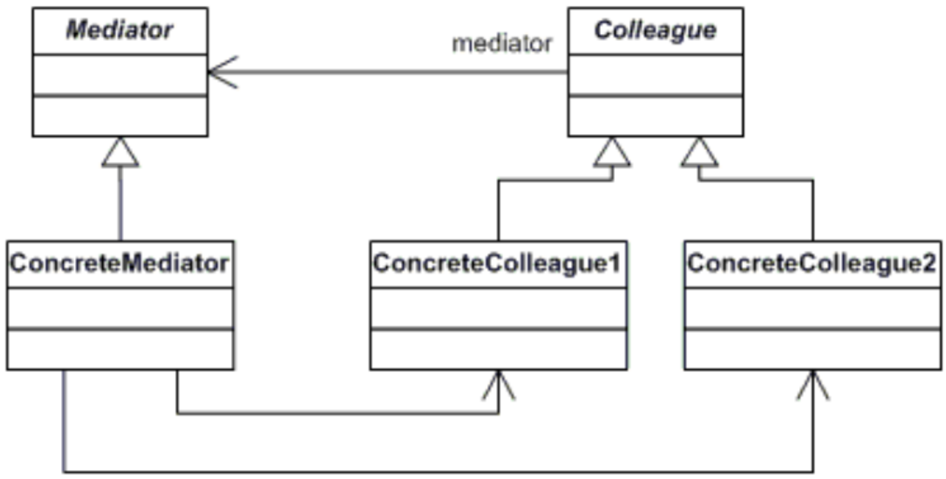
Behavioral Patterns: Iterator

Known Uses

- Use the Iterator pattern when…
  - You are iterating over a collection and one of these conditions holds:
    - There are various ways of traversing it (several enumerators).
    - There are different collections for the same kind of traversing.
    - Different filters and orderings might apply.

Behavioral Patterns: Mediator

UML: *Mediator*



Frequency of use: 1 2 3 4 5 medium low

Behavioral Patterns: 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.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to "full object status".

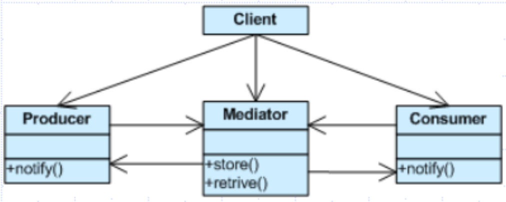
Behavioral Patterns: Mediator

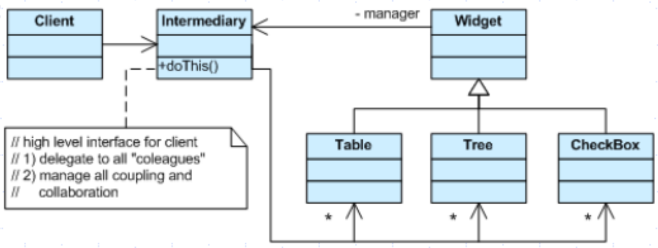
Role

- The Mediator pattern is there to enable objects to communicate without knowing each other's identities. It also encapsulates a protocol that objects can follow.

Structure

- Colleagues (or peers) are not coupled to one another. Each talks to the Mediator, which in turn knows and conducts the orchestration of the others. The "many to many" mapping between colleagues that would otherwise exist, has been "promoted to full object status". This new abstraction provides a locus of indirection where additional leverage can be hosted.





Behavioral Patterns: Mediator

Example

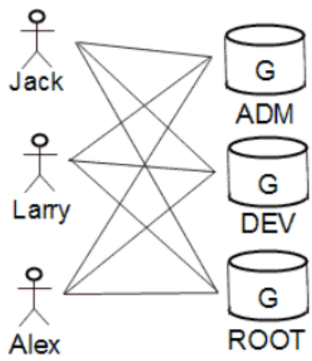
- The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

Discussion

- We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

Discussion (Cont...)

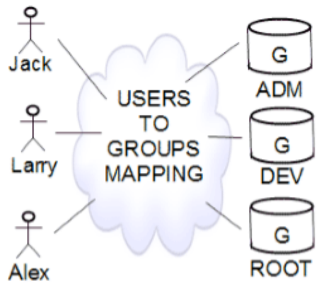
- In Unix, permission to access system resources is managed at three levels of granularity: world, group, and owner. A group is a collection of users intended to model some functional affiliation. Each user on the system can be a member of one or more groups, and each group can have zero or more users assigned to it. Next figure shows three users that are assigned to all three groups.
- If we were to model this in software, we could decide to have User objects coupled to Group objects, and Group objects coupled to User objects. Then when changes occur, both classes and all their instances would be affected.



Behavioral Patterns: Mediator

Discussion (Cont…)

- An alternate approach would be to introduce "an additional level of indirection" - take the mapping of users to groups and groups to users, and make it an abstraction unto itself. This offers several advantages: Users and Groups are decoupled from one another, many mappings can easily be maintained and manipulated simultaneously, and the mapping abstraction can be extended in the future by defining derived classes
- Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object: encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.



Behavioral Patterns: Mediator

Discussion (Cont…)

- The Mediator pattern promotes a "many-to-many relationship network" to "full object status". Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.
- 08An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The Mediator pattern provides a flexible and non-invasive way to associate and manage users and groups.

Rules of Thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.

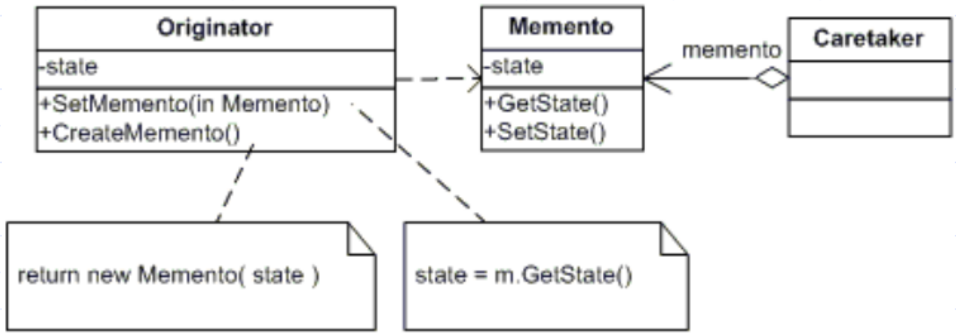
  Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.

Known Uses

- Use the Mediator pattern when…
  - Objects communicate in well-structured but potentially complex ways.
  - The objects' identities should be protected even though they communicate.
  - Some object behaviors can be grouped and customized

Behavioral Patterns: Memento

UML: *Memento*



Frequency of use: 1 2 3 4 5 low

Behavioral Patterns: Memento

Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- A magic cookie that encapsulates a "check point" capability.
- Promote undo or rollback to full object status.

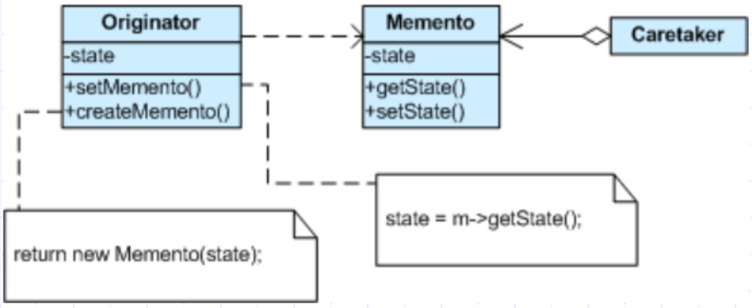
Behavioral Patterns: Memento

Role

- This pattern is used to capture an object's internal state and save it externally so that it can be restored later.

Behavioral Patterns: Memento

Structure

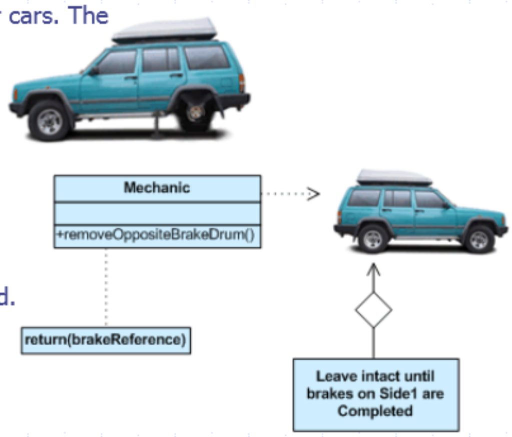


Behavioral Patterns: Memento

Example

◆ Example

The Memento captures and externalizes an object's internal state so that the object can later be restored to that state. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.



Behavioral Patterns: Memento

Problem

- ◆ Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

Discussion

- The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state. The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.
- An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento object.
- The Memento design pattern defines three distinct roles:
  - Originator - the object that knows how to save itself.
  - Caretaker - the object that knows why and when the Originator needs to save and restore itself.
  - Memento - the lock box that is written and read by the Originator, and shepherded by the Caretaker.

Rules of Thumb

- Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.
- Command can use Memento to maintain the state required for an undo operation.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Rules of Thumb (Cont…)

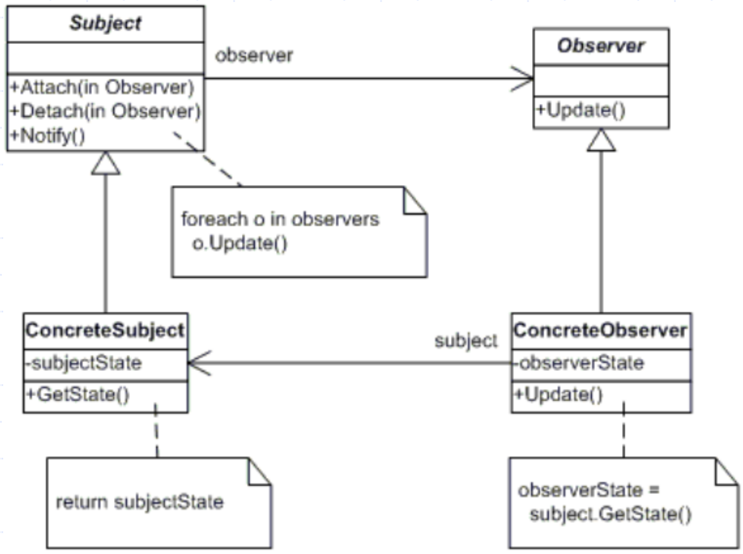
- Memento can let you compose a Mediator out of smaller pieces through recursive composition.
- Decorator is designed to let you add responsibilities to objects without subclassing. Memento's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Memento and Decorator are often used in concert.
- Flyweight is often combined with Memento to implement shared leaf nodes.

Known Uses

- Use the Memento pattern when…
  - An object's state must be saved to be restored later, and
  - It is undesirable to expose the state directly.

Behavioral Patterns: Observer

UML: *Observer*

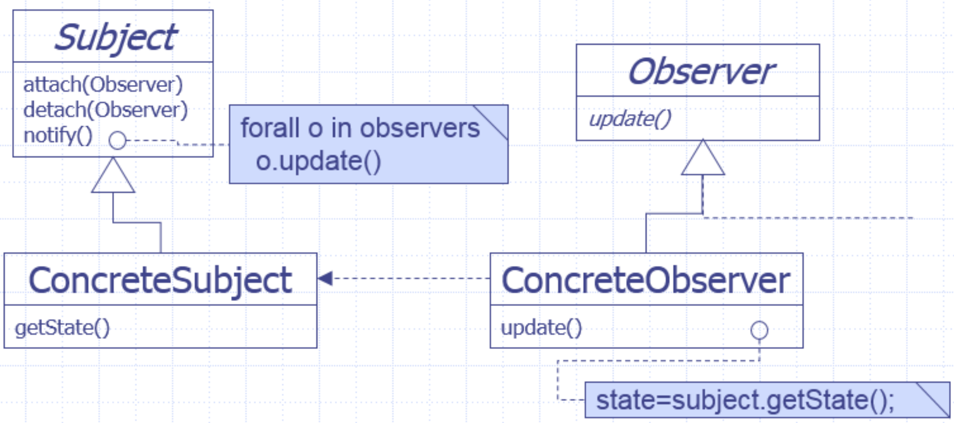


Frequency of use: 1 2 3 4 5 high

Behavioral Patterns: Observer

Intent

- Many-to-one dependency between objects
- Use when there are two or more views on the same "data"
- aka "Publish and subscribe" mechanism
- Choice of "push" or "pull" notification styles



Behavioral Patterns: Observer

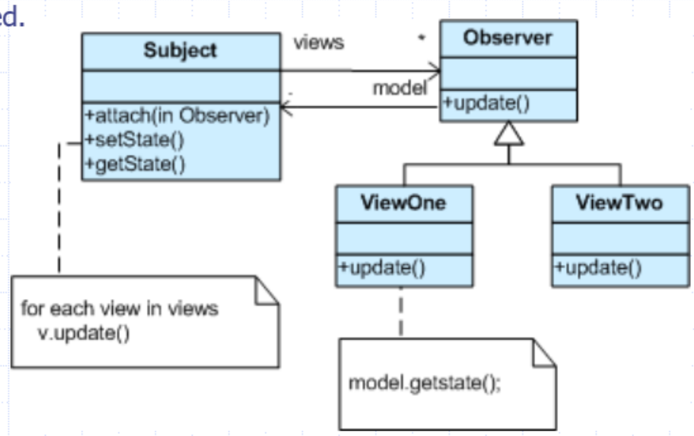
Role

- The Observer pattern defines a relationship between objects so that when one changes its state, all the others are notified accordingly. There is usually an identifiable single publisher of new state, and many subscribers who wish to receive it.

Behavioral Patterns: Observer

Structure

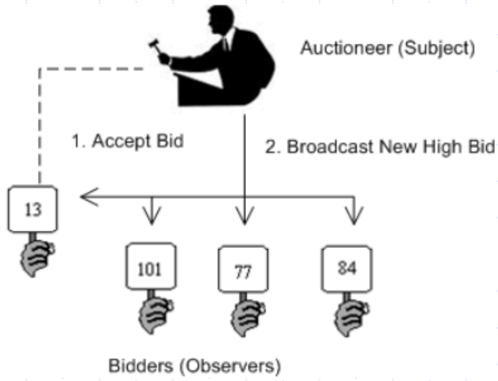
- Subject represents the core (or independent or common or engine) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. The Subject prompts the Observer objects to do their thing. Each Observer can call back to the Subject as needed.



Behavioral Patterns: Observer

Example

- The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.



Behavioral Patterns: Observer

Problem

- A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Discussion

- Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.
- This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.
- The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Behavioral Patterns: Observer

Discussion (Cont…)

- Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.
- The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

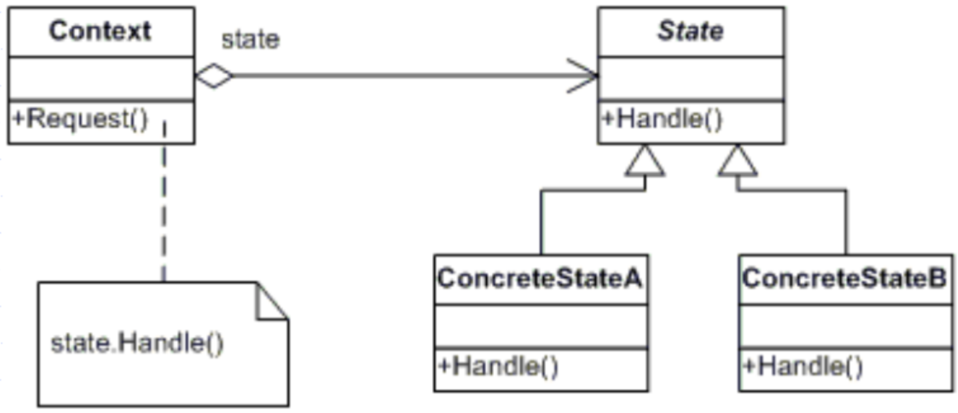
Rules of Thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
- Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.
- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.

Known Uses

- Use the Observer pattern when…
  - There are aspects to an abstraction that can vary independently.
  - Changes in one object need to be propagated to a selection of other objects, not all of them.
  - The object sending the changes does not need to know about the receivers.

UML: *State*



Frequency of use: 1 2 3 4 5 medium

Behavioral Patterns: State

Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

Behavioral Patterns: State

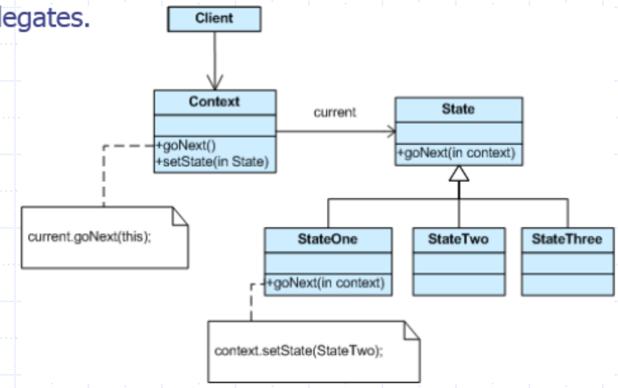
Role

- The next pattern in this group, the State pattern, can be seen as a dynamic version of the Strategy pattern. When the state inside an object changes, it can change its behavior by switching to a set of different operations. This is achieved by an object variable changing its subclass, within a hierarchy.

Behavioral Patterns: State

Structure

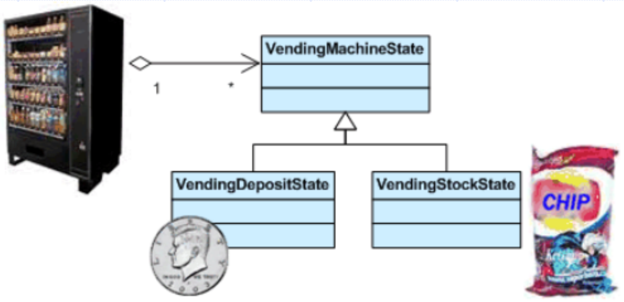
- The state machine's interface is encapsulated in the "wrapper" class. The wrappee hierarchy's interface mirrors the wrapper's interface with the exception of one additional parameter. The extra parameter allows wrappee derived classes to call back to the wrapper class as necessary. Complexity that would otherwise drag down the wrapper class is neatly compartmented and encapsulated in a polymorphic hierarchy to which the wrapper object delegates.



Behavioral Patterns: State

Example

- The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.



Behavioral Patterns: State

Problem

- A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterixed by large and numerous case statements that vector flow of control based on the state of the application.

Discussion

- The State pattern is a solution to the problem of how to make behavior depend on state.
- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer

Discussion (Cont…)

- The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.
- A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accommodating state transition actions.

Behavioral Patterns: State

Rules of Thumb

- State objects are often Singletons.
- Flyweight explains when and how State objects can be shared.
- Interpreter can use State to define parsing contexts.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects.

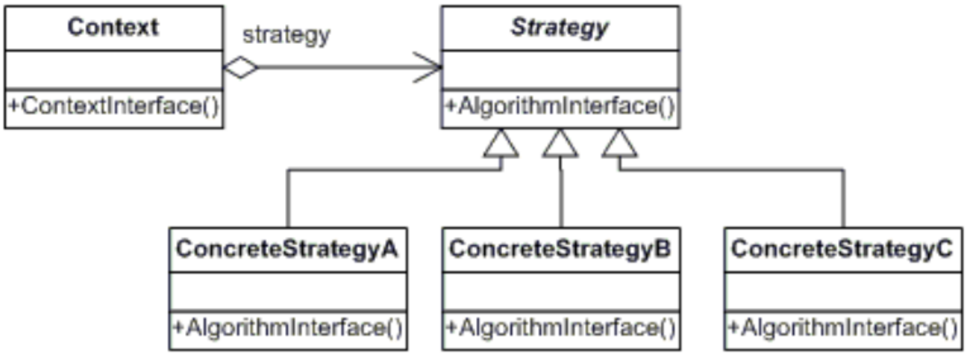
Behavioral Patterns: State

Known Uses

- Use the State pattern when…
  - You have objects that:
    - Will change their behavior at runtime, based on some context
    - Are becoming complex, with many conditional branches
  - You want to:
    - Vary the set of handlers for an object request dynamically
    - Retain flexibility in assigning requests to handlers

Behavioral Patterns: Strategy

UML: *Strategy*

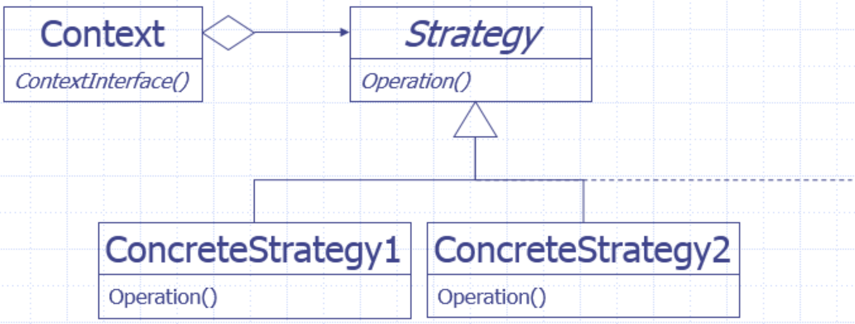


Frequency of use: 1 2 3 4 5 medium high

Behavioral Patterns: Strategy

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.
- Make algorithms interchangeable---"changing the guts"
- Alternative to subclassing
- Choice of implementation at run-time

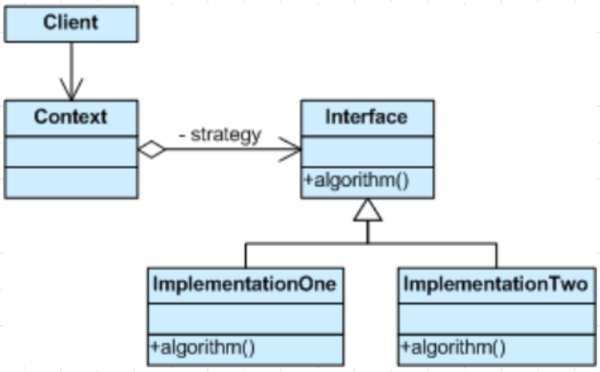


Role

- The Strategy pattern involves removing an algorithm from its host class and putting it in a separate class. There may be different algorithms (strategies) that are applicable for a given problem. If the algorithms are all kept in the host, messy code with lots of conditional statements will result. The Strategy pattern enables a client to choose which algorithm to use from a family of algorithms and gives it a simple way to access it. The algorithms can also be expressed independently of the data they are using.

Structure

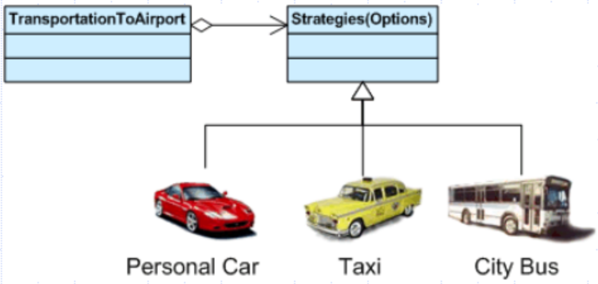
- The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.



Behavioral Patterns: Strategy

Example

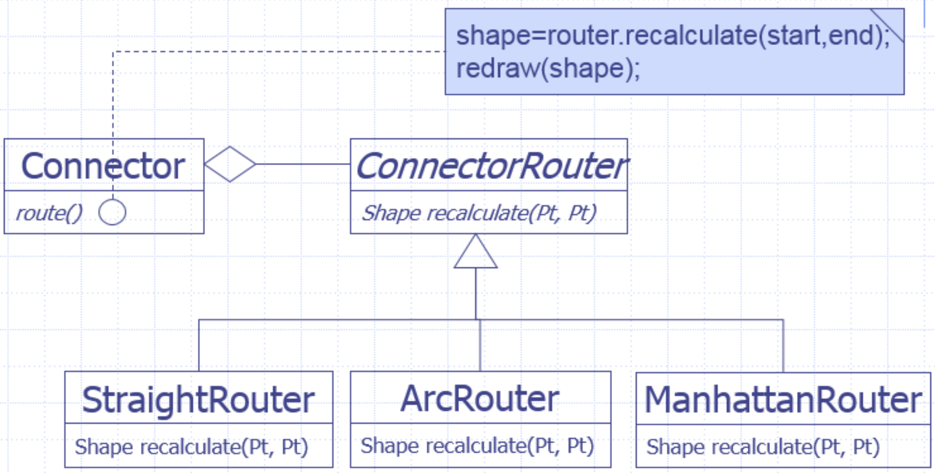
- A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.



Behavioral Patterns: Strategy

Example (Cont…)

- Drawing different connector styles:



Behavioral Patterns: Strategy

Problem

- One of the dominant strategies of object-oriented design is the "open-closed principle".

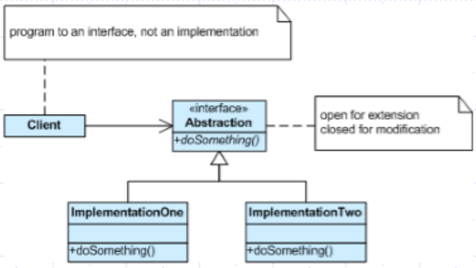


Figure demonstrates how this is routinely achieved - encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.

Behavioral Patterns: Strategy

Problem (Cont…)

- A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in figure is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".
- A more popular characterization of this "abstract coupling" principle is "Program to an interface, not an implementation".
- Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.

Rules of Thumb

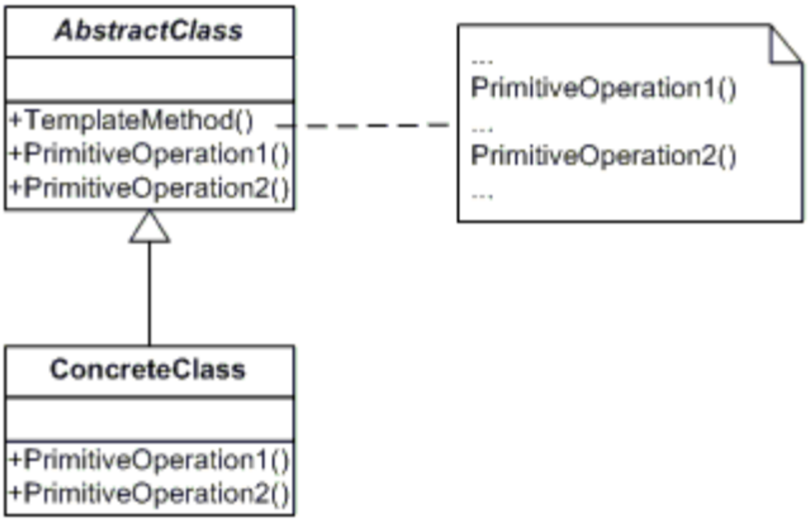
- Strategy is like Template Method except in its granularity.
- State is like Strategy except in its intent.
- Strategy lets you change the guts of an object. Decorator lets you change the skin.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the 'handle/body' idiom. They differ in intent - that is, they solve different problems.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- Strategy objects often make good Flyweights.

Known Uses

- Use the Strategy pattern when…
  - Many related classes differ only in their behavior.
  - There are different algorithms for a given purpose, and the selection criteria can be codified.
  - The algorithm uses data to which the client should not have access.

Behavioral Patterns: Template Method

UML: *Template Method*



Frequency of use: 1 2 3 4 5 medium

Behavioral Patterns: Template Method

Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

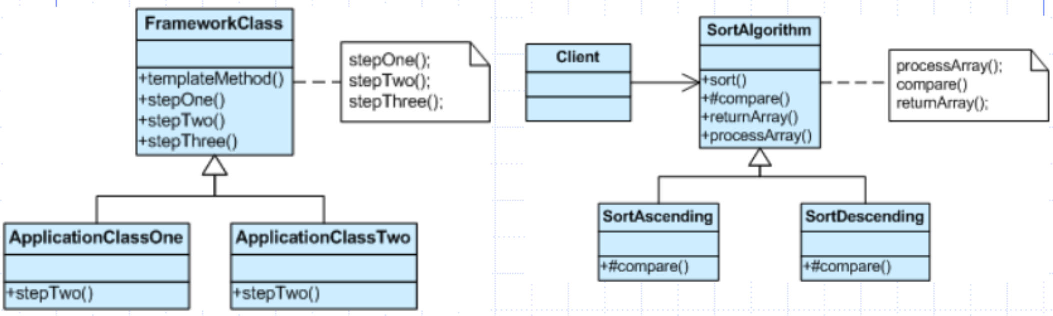
Behavioral Patterns: Template Method

Role

- The Template Method pattern enables algorithms to defer certain steps to subclasses. The structure of the algorithm does not change, but small well-defined parts of its operation are handled elsewhere.

Behavioral Patterns: Template Method

Structure

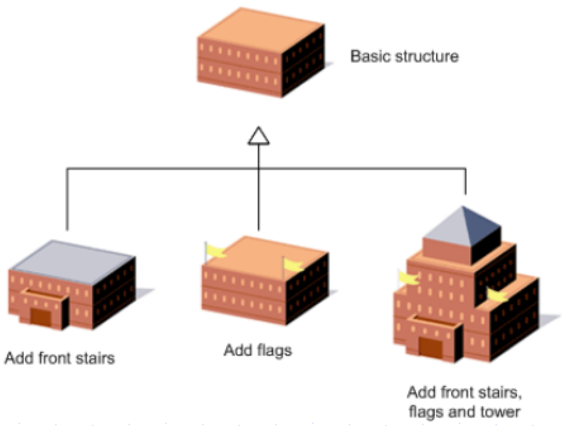


The implementation of template_method() is: call step_one(), call step_two(), and call step_three(). step_two() is a "hook" method – a placeholder. It is declared in the base class, and then defined in derived classes. Frameworks (large scale reuse infrastructures) use Template Method a lot. All reusable code is defined in the framework's base classes, and then clients of the framework are free to define customizations by creating derived classes as needed.

Behavioral Patterns: Template Method

Example

- The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans with different variations available for each. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the later stages of construction to produce a wider variety of models.



Behavioral Patterns: Template Method

Problem

- Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

Discussion

- The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.
- The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Discussion (Cont…)

- Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options. In so doing, the framework becomes the "center of the universe", and the client customizations are simply "the third rock from the sun". This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Rules of Thumb

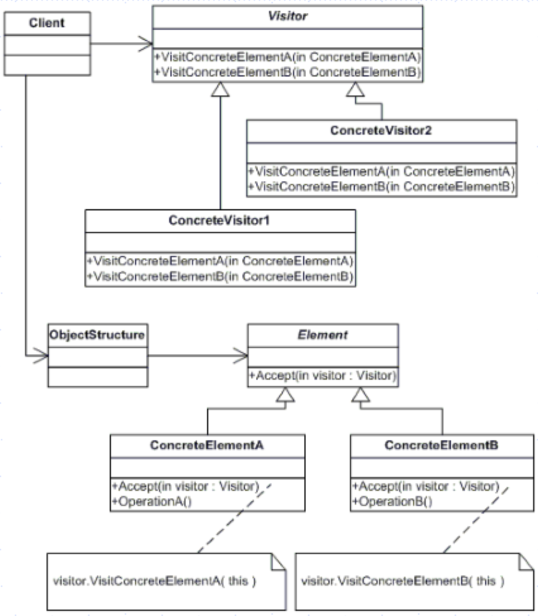
- Strategy is like Template Method except in its granularity.
- Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm.
- Strategy modifies the logic of individual objects. Template Method modifies the logic of an entire class.
- Factory Method is a specialization of Template Method.

Known Uses

- Use the Template Method pattern when…
  - Common behavior can be factored out of an algorithm.
  - The behavior varies according to the type of a subclass.

Behavioral Patterns: Visitor

UML: *Visitor*



Frequency of use: 1 2 3 4 5 low

Behavioral Patterns: 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.
- The classic technique for recovering lost type information.
- Do the right thing based on the type of two objects.
- Double dispatch

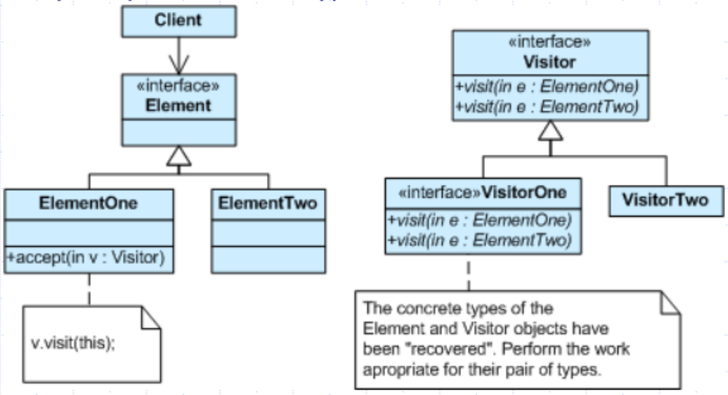
Role

- The Visitor pattern defines and performs new operations on all the elements of an existing structure, without altering its classes.

Behavioral Patterns: Visitor

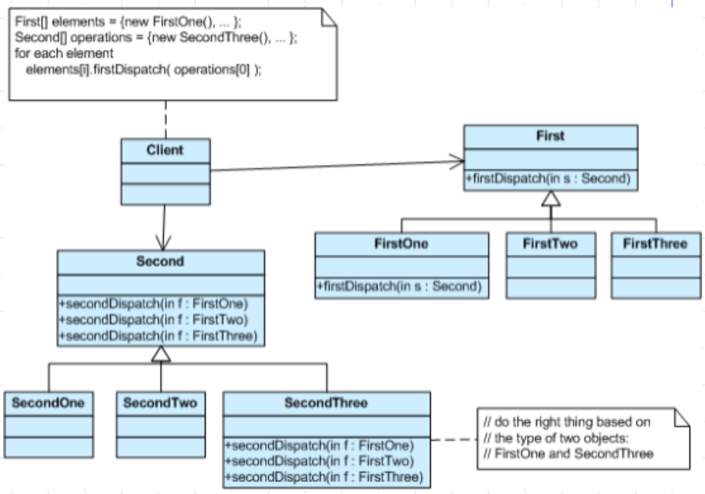
Structure

- The Element hierarchy is instrumented with a "universal method adapter". The implementation of accept() in each Element derived class is always the same. But – it cannot be moved to the Element base class and inherited by all derived classes because a reference to this in the Element class always maps to the base type Element.



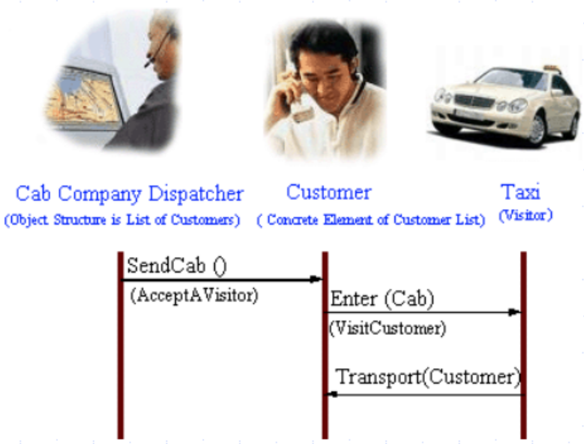
Structure (Cont…)

- When the polymorphic firstDispatch() method is called on an abstract First object, the concrete type of that object is "recovered". When the polymorphic secondDispatch() method is called on an abstract Second object, its concrete type is "recovered". The application functionality appropriate for this pair of types can now be exercised.



Example

- This pattern can be observed in the operation of a taxi company. When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is.



Problem

- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Behavioral Patterns: Visitor

Discussion

- Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects. The approach encourages designing lightweight Element classes - because processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.
- Visitor implements "double dispatch". OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver. In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).

Discussion (Cont…)

- The implementation proceeds as follows. Create a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.
- Each operation to be supported is modeled with a concrete derived class of the Visitor hierarchy. The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.

Discussion (Cont…)

- Add a single pure virtual accept() method to the base class of the Element hierarchy. accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.
- Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.
- Everything for "elements" and "visitors" is now set-up. When the client needs an operation to be performed, (s)he creates an instance of the Vistor object, calls the accept() method on each Element object, and passes the Visitor object.

Behavioral Patterns: Visitor

Discussion (Cont…)

- The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass. accept() dispatch plus visit() dispatch equals double dispatch.
- The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class. But, if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort.
- An acknowledged objection to the Visitor pattern is that is represents a regression to functional decomposition - separate the algorithms from the data structures. While this is a legitimate interpretation, perhaps a better perspective/rationale is the goal of promoting non-traditional behavior to full object status.

Behavioral Patterns: Visitor

Rules of Thumb

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.
- The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

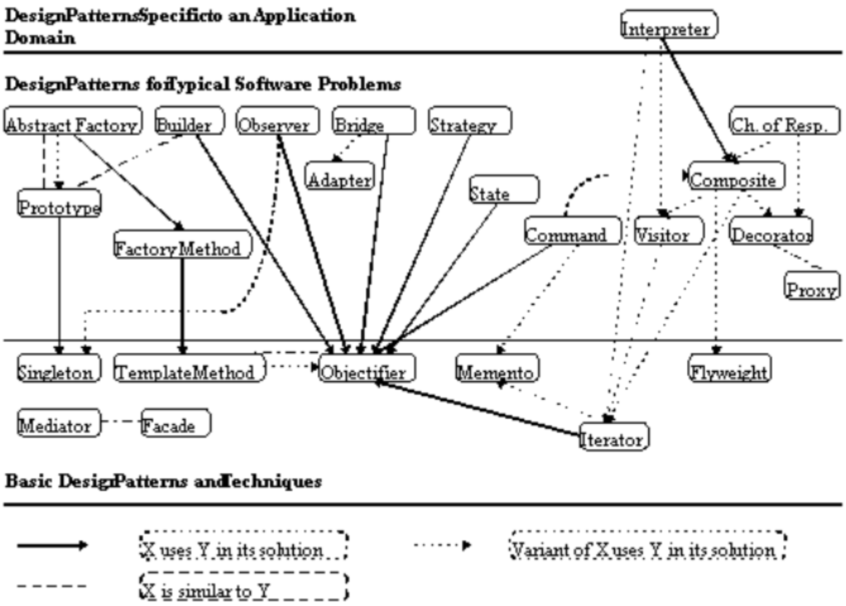
Behavioral Patterns: Visitor

Known Uses

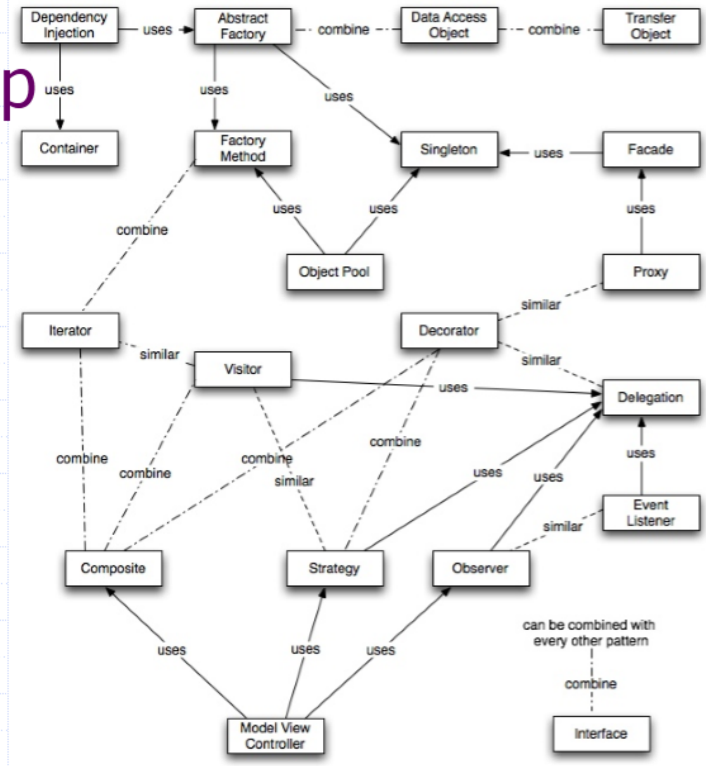
- Use the Visitor pattern when…
  - You need the flexibility to define new operations over time.
  - There is a need perform operations that depend on concrete classes of an object structure, and the structure may contain classes of objects with differing interfaces.
  - Distinct and unrelated operations must be performed on objects in an object structure, and you want to avoid distributing/replicating similar operations in their classes
  - The classes defining the object structure rarely change, but new operations may be added every once in a while.

Category	Patterns	Frequency of use
Creational	Abstract Factory	
	Builder	
	Factory Method	
	Prototype	
	Singleton	
Structural	Adapter	
	Bridge	
	Composite	
	Decorator	
	Façade	
	Proxy	
Behavioral	Chain of Responsibility	
	Command	
	Flyweight	
	Interpreter	
	Iterator	
	Mediator	
	Memento	
	Observer	
	State	
	Strategy	
	Template Method	
	Visitor	

Patterns Relationship



Patterns Relationship (Cont.)



Golden Rules of Design Patterns

- Client should always call the abstraction (interface) and not the exact implementation.
- Future changes should not impact the existing system.
- Change always what is changing.
- Have loose coupling
  - Inheritance (Very coupled)
  - Composition
  - Aggregation
  - Association
  - Dependency
  - Realization (Least couple)