Design Patterns
Publish Link:
https://docs.google.com/document/d/e/2PACX-1vRdHi_NfXo1h2p_s1hZu0xAwR_9ONhNBOsvfH7vCk49zAtJcNKIBQfXxs3S7vsnNg5jwBGXJEVt9Q7X/pub

Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

1. Introduction

1.1 What Is a Design Pattern?

- A general pattern has four essential elements:
  - Pattern name
  - Problem
  - Solution
  - Consequences

1.2 Design Patterns in Smalltalk MVC

1.3 Describing Design Patterns

- Pattern Name and Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

1.4 The Catalog of Design Patterns

- **Abstract Factory**
- **Adaptor**
- **Bridge**
- **Builder**
- **Chain of Responsibility**
- **Command**
- **Composite**
- **Decorator**
- **Facade**
- **Factory Method**
- **Flyweight**
- **Interpreter**
- **Iterator**

- **Mediator**
- **Memento**
- **Observer**
- **Prototype**
- **Proxy**
- **Singleton**
- **State**
- **Strategy**
- **Template Method**
- **Visitor**

1.5 Organizing the Catalog

1.6 How Design Pattern Solve Design Problems

- Finding Appropriate Objects
- Determining Object Granularity
- Specifying Object Interfaces
- Specifying Object Implementations
- Class versus Interface Inheritance
- Programming to an Interface, not an Implementation
- Putting Reuse Mechanisms to Work
    - Inheritance versus Composition
    - Delegation
    - Inheritance versus Parameterized Types
    - Relating Run-Time and Compile-Time Structures
- Designing for Change
    - *Creating an object by specifying a class explicitly*
    - *Dependence on specific operations*
    - *Dependence on hardware and software platform*
    - *Dependence on object representations or implementations*
    - *Algorithmic dependencies*
    - *Tight coupling*
    - *Extending functionality by subclassing*
    - *Inability to alter classes conveniently*
    - **Application Programs**
    - **Toolkits**
    - **Frameworks**
        - *Design patterns are more abstract than frameworks*
        - *Design patterns are smaller architectural elements than frameworks*
        - *Design patterns are less specialized than frameworks*

1.7 How to Select a Design Pattern

1.8 How to Use a Design Pattern

- *Read the patterns once through for an overview*
- *Go back and study the Structure, Participants, and Collaborations sections*
- *Look at the Sample Code section to see a concrete example of the pattern in code*
- *Choose names for pattern participants that are meaningful in the application context*
- *Define the classes*
- *Define application-specific names for operations in the pattern*
- *Implement the operations to carry out the responsibilities and collaboration in the pattern*

2. A Case Study: Designing a Document Editor

2.1 Design Problems

2.2 Document Structure

2.3 Formatting

2.4 Embellishing the User Interface

2.5 Supporting Multiple Look-and-Feel Standards

2.6 Supporting Multiple Window Systems

2.7 User Operations

2.8 Spelling Checking and Hyphenation

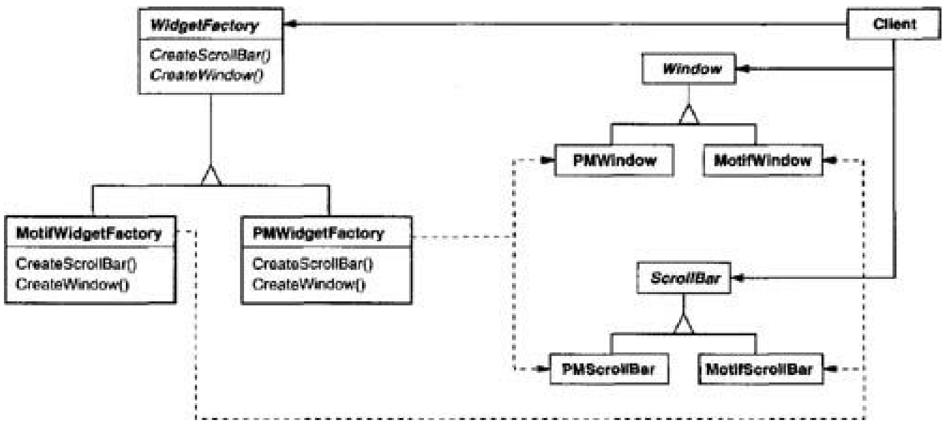
2.9 Summary

Design Pattern Catalog

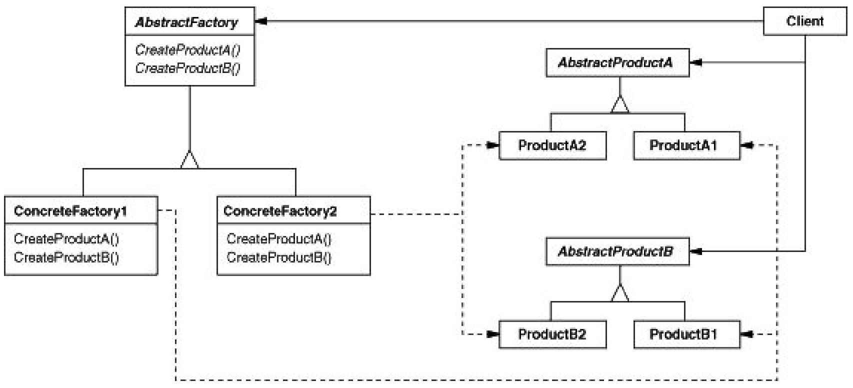
3. Creational Patterns

- Class Creational Pattern - uses inheritance to vary the class that’s instantiated
- Object Creation Pattern - delegates instantiation to another object
- **Creational patterns become important as systems evolve to depend more on object composition than class inheritance**
- Recurring themes in these patterns:
  - they encapsulate knowledge about which concrete classes the system uses
  - they hid how instance of theses classes are created and put together

Abstract Factory (Object Creational)

- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Also Known As**
  - Kit
- **Motivation**
  - To create a family of products that are related and are to be used together
  - Concrete factories enforces the relationship and dependencies
  - 

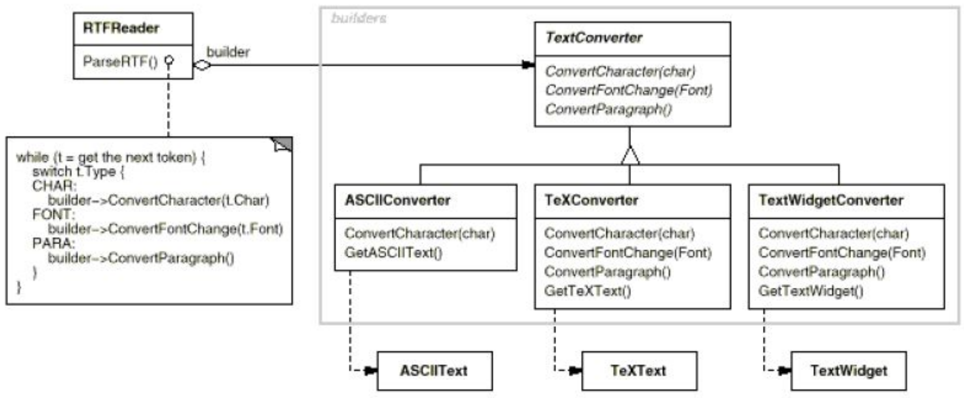
- **Applicability**
  - Use the abstract factory when:
    - a system should be independent of how its products are created, composed, and represented
    - a system should be configured with one of multiple families of products
    - a family of related product objects is designed to be used together, and you need to enforce this constraint
    - you want to provide a class library of products, and you want to reveal just the interfaces, not their implementations
- **Structure**

- 

- **Participants**
    - **AbstractFactory**
        - declares an interface for operations that create abstract product objects
    - **ConcreteFactory**
        - implements the operations to create concrete product objects
    - **AbstractProduct**
        - declares an interface for a type of product object
    - **ConcreteProduct**
        - defines a product object to be created by the corresponding concrete factory
        - implements the AbstractProduct interface
    - **Client**
        - uses only interfaces by AbstractFactory and AbstractProduct classes
- **Collaborations**
    - Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
    - AbstractFactory defers creation of product objects to its ConcreteFactory subclass
- **Consequences**
    - The Abstract Factory pattern has the following benefits and liabilities
        - *It isolates concrete classes* - client manipulate instance through abstract interfaces
        - *It makes exchanging product families easy* - this can be achieved simply by changing the concrete factory
        - *It promotes consistency among products* - concrete factory ensures this and it’s important that an application use objects from only one family at a time
        - *Supporting new kinds of products is difficult* - extending abstract factory to support new kinds of product isn’t easy as that will requires extending the factory interface thus changing AbstractFactor class and all its subclasses, this can be circumvented by passing the required product as parameter
- **Implementation**

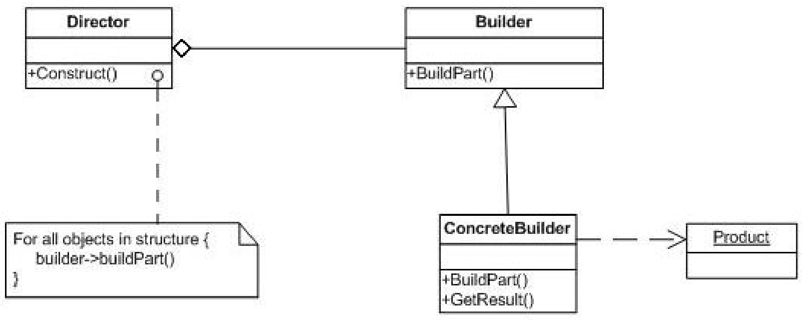
    - *Factories as singletons* - concrete factories can be singletons
    - *Creating the products*
        - a concrete factories specifies its products by overriding the factory method for each so a new concrete factory subclass is required for each product family (even if the product families differ slightly
        - if many product families are possible, the concrete factory can be implemented using the Prototype pattern
    - *Defining extensible factories*
        - the kind of product is encoded in operation signatures
        - a more flexible but less safe design is to add a parameter to operations that create objects
- **Sample Code**
    - Instead of having a AbstractFactory class a concrete class can act as an interface
- **Related Patterns**
    - AbstractFactory classes are often implemented with factory methods (Factory Method), but they can also be implemented using Prototype
    - A concrete factory is often a singleton (Singleton)

Builder (Object Creational)

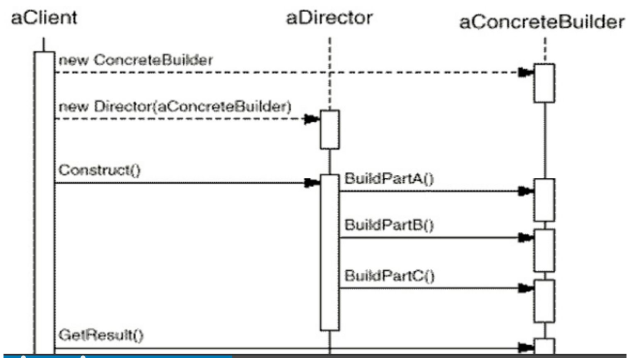
- **Intent**
    - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- **Motivation**
    - 

```
while (t = get the next token) {
   switch t.Type {
   CHAR:
      builder->ConvertCharacter(t.Char)
   FONT:
      builder->ConvertFontChange(t.Font)
   PARA:
      builder->ConvertParagraph()
   }
}
```

    - Each kind of converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface
    - Each converter class is called a **builder** and the reader is called the **director**
- **Applicability**
    - Use the Builder pattern when
        - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
        - the construction process must allow different representations for the object that's constructed
- **Structure**



- **Participants**
  - **Builder** (TextConverter)
    - specifies an abstract interface for creating parts of a Product object
  - **ConcreteBuilder** (ASCIIConverter, TeXConverter…)
    - constructs and assembles parts of the product by implementing the Builder interface
    - defines and keeps track of the representation it creates
    - provides an interface for retrieving the product (eg GetASCIIText..)
  - **Director** (RTFReader)
    - constructs an object using the Builder interface
  - **Product** (ASCIIText…)
    - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
    - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result
- **Collaborations**
  - The client creates the Director object and configures it with the desired Builder object
  - Director notifies the builder whenever a part of the product should be built
  - Builder handles requests from the builder and adds parts to the product
  - The client retrieves the product from the builder



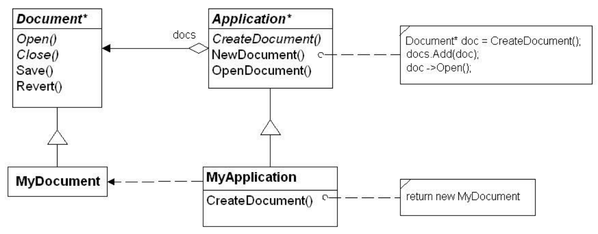
- **Consequences**
  - *It lets you vary a product's internal representation* - the builder object provides director with an abstract interface for constructing the product

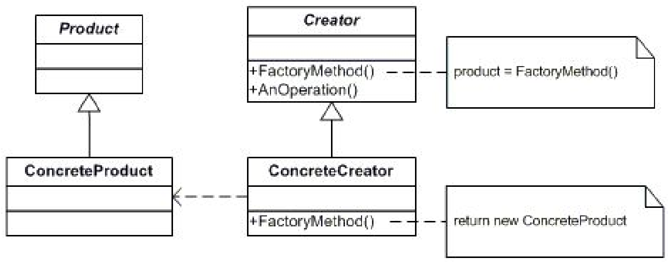
    - *It isolates code for construction and representation* - improves modularity by encapsulating the way a complex object is constructed & represented, each builder contains all the code to create and assemble a product
    - *It gives you finer control over the construction process* - constructs the product step by step under director's control, only finished product is retrieved by the director
- **Implementation**
    - *Assembly and construction interface* - builder creates the product step-by-step so the Builder class interface must be general enough, sometimes you might need access to parts of the product constructed earlier
    - *Why no abstract class for products?* - the products differ greatly in their representation that there is little gain from giving different products a common parent class
    - *Empty methods as default in Builder* - clients override only the operations they're interested in
- **Sample Code**
    - each step could return the builder so that the steps can be chained
    - internal representation and fields can be hidden in some of the constructors
- **Related Patterns**
    - Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.
    - A composite is what the builder often builds.

Factory Method (Class Creational)

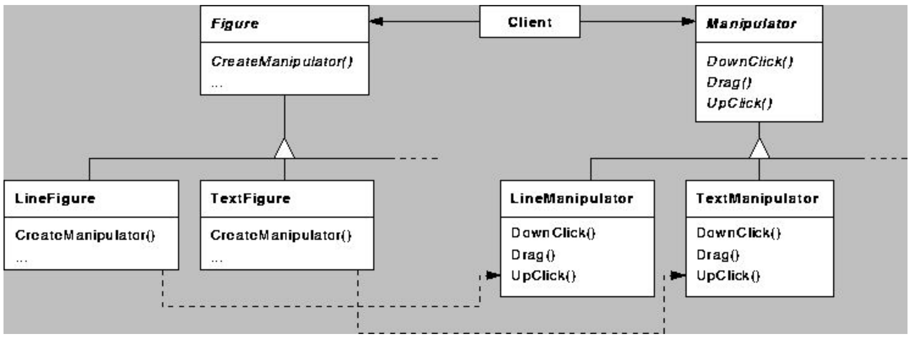
- **Intent**
    - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Also Known As**
    - Virtual Constructor
- **Motivation**

Factory Method Motivation



-
    -
    - Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.
    - Applications can't predict the type of subclass to instantiate. Frameworks must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.
    - Factory method encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.
- **Applicability**
    - a class can't anticipate the class of objects it must create
    - a class wants its subclasses to specify the objects it creates
    - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
- **Structure**
    - 

- **Participants**
    - **Product** (Document)
        - defines the interface of objects the factory method creates
    - **ConcreteProduct** (MyDocument)
        - implements the Product interface
    - **Creator** (Application)
        - declares the factory method, which returns an object of type Product. Create may also define a default implementation of the factory method that returns a default ConcreteProduct object.
        - may call the factory method to create a Product object.
    - **ConcreteCreator** (MyApplication)

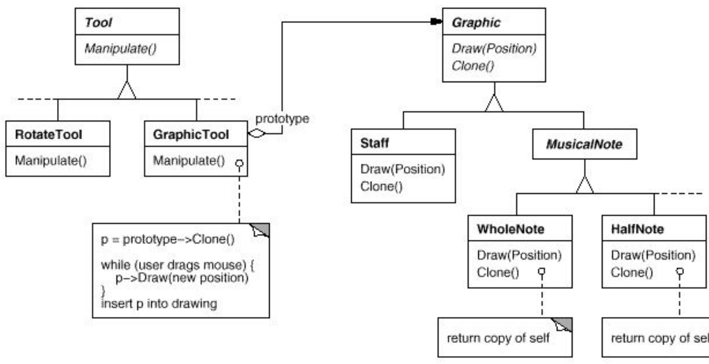
        - overrides the factory method to return an instance of a ConcreteProduct
- **Collaborations**
    - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct
- **Consequences**
    - A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass to Creator class anyway, but otherwise the client now must deal with another point of evolution.
    - Additional consequences:
        - *Provides hooks for subclasses* - Factory method gives subclasses a hook for providing an extended version of an object.
        - *Connects parallel class hierarchies* - Clients can find factory methods useful, it doesn't need to be only called by Creators, especially in the case of parallel class hierarchies. Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class.
        - 

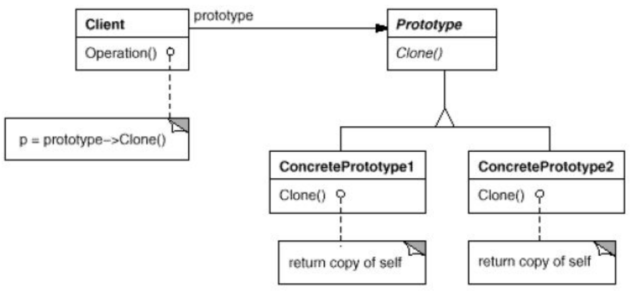
- **Implementation**
    - *2 major varieties*
        - the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares
        - the case when the Creator is a concrete class and provides a default implementation for the factory method.
    - *Parameterized factory methods* - the factory method takes a parameter that identifies the kind of object to create
    - *Language-specific variants and issues* - Factory method can just return the class token and the actual object may be initialized even later
    - *Using templates to avoid subclassing* - Provide a template subclass of Creator that's parameterized by the Product class instead of instead of subclassing every time
    - *Naming conventions*

- **Related Patterns**
    - Abstract Factory is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.

- Factory methods are usually called within Template Methods. In the document example above, New Document is a template method.
- Prototypes don't require subclassing the Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.

Prototype (Object Creational)

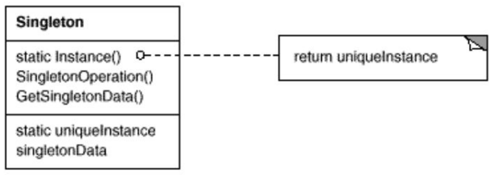
- **Intent**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- **Motivation**
  - Creating objects of a specific type using a framework (the framework is generic) which doesn't know bout the specific type
  - Subclassing will create many classes, composition can also be used
  - Prototype pattern reduces the number of classes required
  - 

- **Applicability**
  - Use it when a system should be independent of how its products are created, composed, and represented; *and*
    - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
    - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
    - when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.
- **Structure**
  - 

- **Participants**
    - **Prototype** (Graphic) - declares an interface for cloning itself
    - **ConcretePrototype** (Staff, WholeNote, HalfNote) - implements an operations for cloning itself
    - **Client** (GraphicTool) - creates a new object by asking a prototype to clone itself
- **Collaborations**
    - a client asks a prototype to clone itself
- **Consequences**
    - similar to Abstract Factory & Builder it hides the concrete product classes from client, thereby reducing the no. of names clients know about, additionally;
    - *Adding and removing products at run time* - a new concrete product can be incorporated by simply registering a prototypical instance, a client can install & remove prototypes at run-time
    - *Specifying new objects by varying values* - define new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects (cloning a prototype is similar to instantiating a class), values for object variables can be defined
    - *Specifying new objects by varying structure* - Composite objects can be created using deep copy
    - *Reduced subclassing* - unlike Factory Method Prototype pattern lets you clone a prototype by no using parallel hierarchy of Creator classes, Class objects act like prototypes in certain languages (Java, Objective C, not C++)
    - *Configuring an application with classes dynamically* - at run time prototype manager can create objects of classes that are dynamically loaded
- **Implementation**
    - *Using a prototype manager* - a client will ask the registry for a prototype before cloning it, **prototype manager** is this registry
    - *Implementing the Clone operations* - shallow vs deep copy, should variables be copied, using Save & Load operations (deserialization creates a new object)
    - *Initiating clones* - clone operation needs to be uniform so instance variables cannot be specified, provide a new *Initialize* method to do the same
- **Related Patterns**
    - Prototype and Abstract Factory are competing patterns and they can be used together
    - An Abstract Factory might strore a set of prototypes from which to clone and return product objects
    - Designs that make heavy use of the Composite and Decorator patterns often can benefit Prototype as well

Singleton (Object Creational)

- **Intent**

    - Ensure a class only has one instance, and provide a global point of access to it
- **Motivation**
    - Ensuring only one instance of a class
    - Making globally accessible
    - Making class itself responsible to keep track of its sole instance
- **Applicability**
    - When there must be exactly 1 instance of a class, and it must be accessible to clients from a well-known access point
    - When the sole instance should be extensible by subclassing, and clients should be able to sue an extended instance without modifying their code
- **Structure**
    - 

- **Participants**
    - **Singleton**
        - defines an *Instance* operation that lets clients access its unique instance. Instance is a class operation.
        - may be responsible for creating its own unique instance
- **Collaborations**
    - Clients access a Singleton instance solely through Singleton's Instance operation
- **Consequences**
    - *Controlled access to sole instance* - encapsulates instance
    - *Reduced namespace* - avoids global variables
    - *Permits refinement of operations and representation* - Singleton class can be subclassed and application can be configured to get the required extended class at runtime
    - *Permits a variable number of instances* - more than 1 instance can be supported
    - *More flexible than class operations* - another way to package a singleton's functionality is to use class operations, subclass can't override them polymorphically
- **Implementation**
    - *Ensuring a unique instance*
        - It isn't enough to define the singleton as a global or static object and they rely on automatic initialization:
            - We can't guarantee that only one instance of a static object will ever be declared
            - We might not have enough information to instantiate every singleton at static initialization time

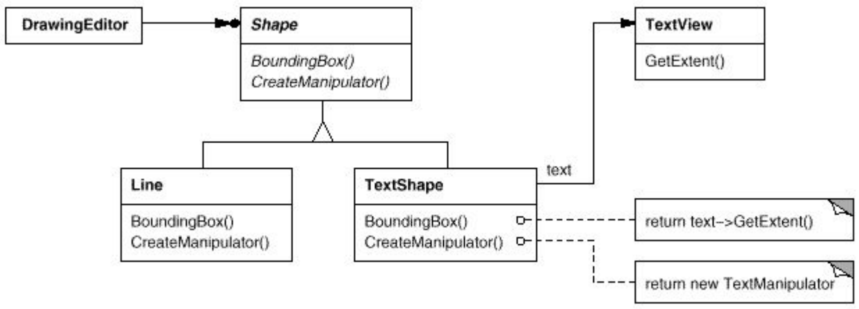
            - No dependencies between singleton’s should exits because order of constructing global objects might not be defined (C++)
    - *Subclassing the Singleton class*
        - Implementation of *Instance* can be moved out of the parent class to a Factory
        - This still doesn’t provide runtime flexibility because factory logic needs to be coded, so maintain a **registry of singletons**
        - Singleton class itself can register itself, if objects are registered in constructor then instances of all possible Singleton subclasses must be created, or else they won’t get registered
    - Instance must be modified whenever you define a new subclass of Factory, this might a problem of abstract factories
    - A possible solution would be to use the registry approach described above
    - Dynamic linking could be useful here as well - it would keep the application from having to load all the subclasses that are not used
- **Related Patterns**
    - Many patterns can be implemented using the Singleton pattern. See Abstract Factory, Builder, and Prototype

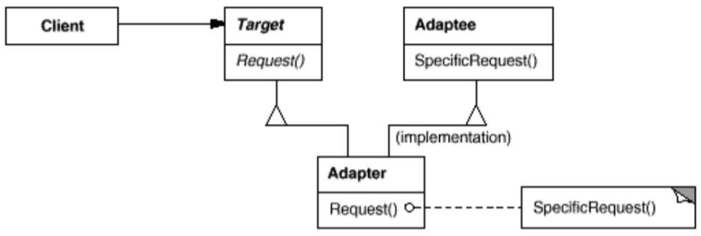
Discussion of Creational Patterns

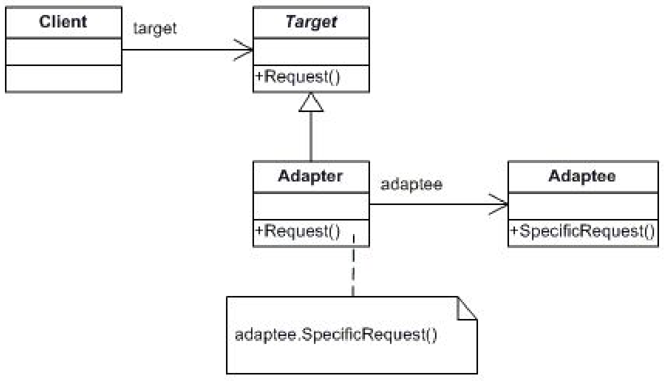
4. Structural Patterns

Adapter (Class, Object Structural)

- **Intent**
    - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
- **Also Known As**
    - Wrapper
- **Motivation**
    - Use a TextView class with Shape (LineShape, PolygonShape)
    - TextView itself supports lots of complicated things like rendering various fonts on screen, buffer management
    - Such classes might be independent and it doesn’t make sense them to confirm to a specific interface (eg Shape) to be used with them
    - Instead TextShape could be defined to adapt TextView interface to Shape’s:
        - by inheriting Shape’s interface and TextView’s implementation
        - by composing a TextView instance within a TextShape and implementing TextShape in terms of *TextView’s* interface
    - These 2 approaches correspond to the class and object versions of the pattern

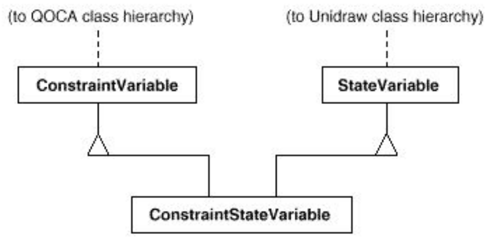


- **Applicability**
  - Use the pattern when
    - you want to use an existing class, and its interface does not match the one you need
    - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
    - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.
- **Structure**
  - Class adapter uses multiple inheritance to adapt one interface to another:
    - 

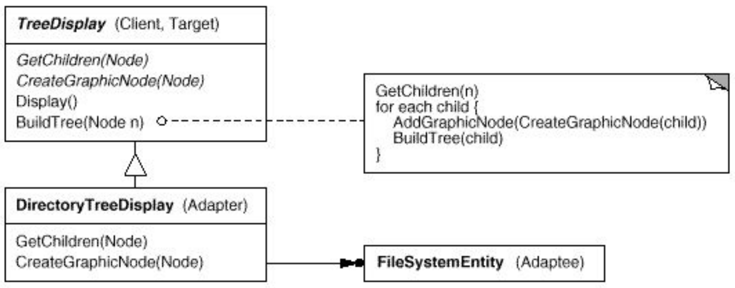
  - An object adapter relies on object composition:
    - 

- **Participants**
  - **Target** (Shape) - defines the domain-specific interface that Client uses
  - **Client** (DrawingEditor) - collaborates with objects conforming to the Target interface
  - **Adaptee** (TextView) - defines an existing interface that needs adapting
  - **Adapter** (TextShape) - adapts the interface of Adaptee to the Target interface
- **Collaborations**

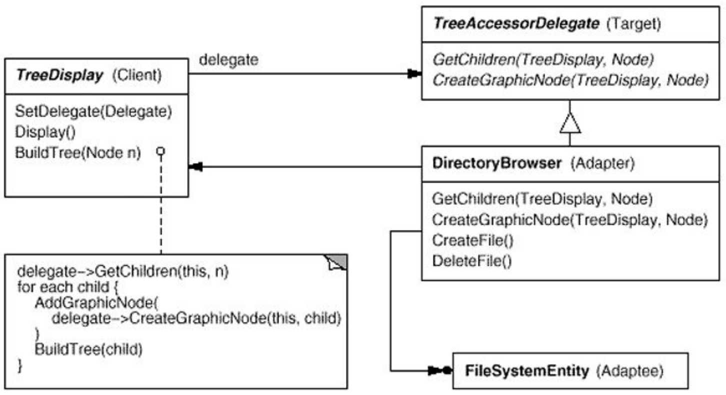
    - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.
- **Consequences**
    - Class and object adapters have different trade-offs. A class adapter
        - adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
        - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
        - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
    - An object adapter
        - lets a single Adapter work with many Adaptees - that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
        - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than Adaptee itself.
    - Here are other issues to consider when using the Adapter pattern:
        - *How much adapting does Adapter do?* - how similar the Target interface is to Adaptee's
        - *Pluggable adapters* - this term is used (by ObjectWorks, Smalltalk) to describe classes with built-in interface adaption
        - *Using two-way adapters to provide transparency* - multiple inheritance is a viable solution in this case because the interfaces of the adapter classes are **substantially different**



- **Implementation**
    - *Implementing class adapter in C++* - for class adapter, inherit publicly from Target and privately from Adaptee, so Adapter would only be subtype of Target
    - *Pluggable adapters* - 3 ways to implement pluggable adapters, the first step is to find a "narrow" interface for Adaptee (https://stackoverflow.com/questions/449424/can-anybody-explain-the-concept-of-pluggable-adapter-to-me-with-good-example)
    - Using narrow interfaces leads to 3 implementation approaches:
        - Using abstract operations



-
  - Using delegate object

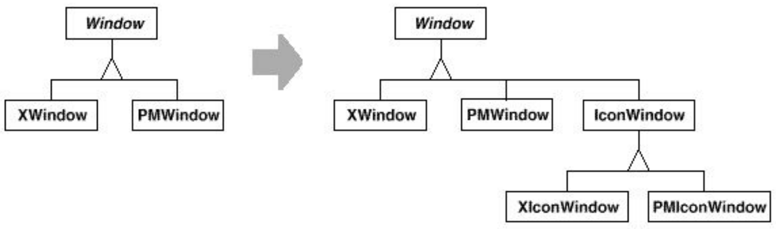

TreeAccessorDelegate (Target)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
delegate
TreeDisplay (Client)
SetDelegate(Delegate)
Display()
BuildTree(Node n)
DirectoryBrowser (Adapter)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
CreateFile()
DeleteFile()
delegate->GetChildren(this, n)
for each child {
AddGraphicNode(
delegate->CreateGraphicNode(this, child)
)
BuildTree(child)
}
FileSystemEntity (Adaptee)

-
  - *Parameterized adapters* - Parameterize an adapter with one or more blocks, the block construct supports adaptation without subclassing. If you are building interface adaptation into a class, this approach offers a convenient alternative to subclassing.
- **Related Patterns**
  - Bridge has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an *existing* object.
  - Decorator enhances another object without changing its interface. A 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.
  - Proxy defines a representative or surrogate for another object and does not change its interface.

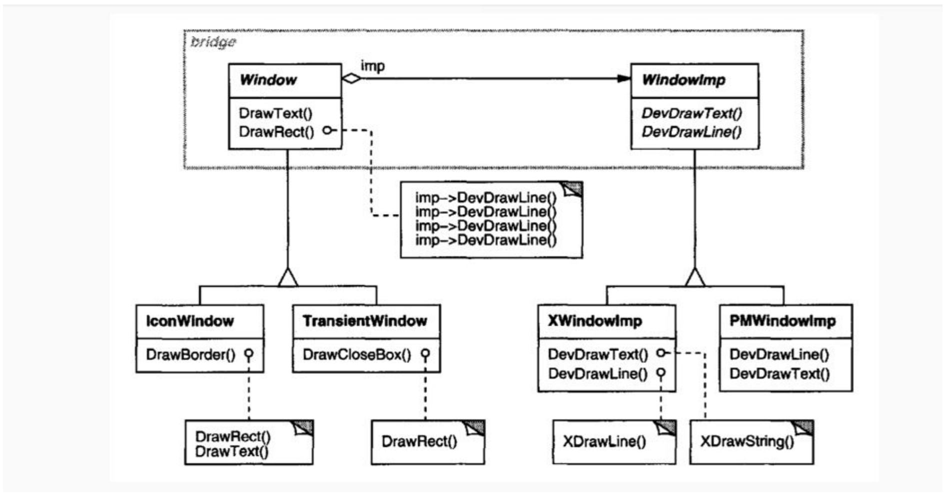
Bridge (Object Creational)

- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently
- **Also Known As**
  - Handle / Body
- **Motivation**

- Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementation independently
- Suppose there is an abstraction and it has different kinds, now if we have a specialized implementation for this abstraction then this needs to be implemented for all the kinds
- It makes the client code platform-dependent because the client instantiates a concrete class with a specific implementation

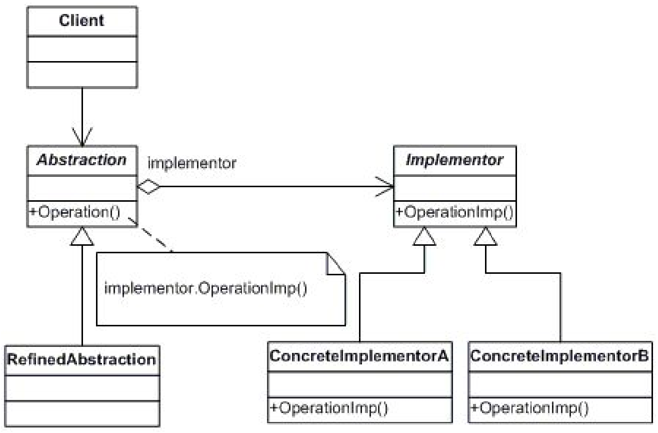


-
- Solution



-

- **Applicability**
  - avoiding a permanent binding b/w an abstraction & its implementation
  - both the abstractions & their implementations should be extensible by subclassing, and can be done independently
  - changes in the implementation of an abstract should have no impact on clients
  - (C++) you want to hide the implementation of an abstraction completely from clients
  - you have a proliferation of classes as shown earlier in the first Motivation diagram (**nested generalizations** is a term for such hierarchies)
  - you want to share an implementation among multiple objects (perhaps using reference counting) and this fact should be hidden from the client
- **Structure**

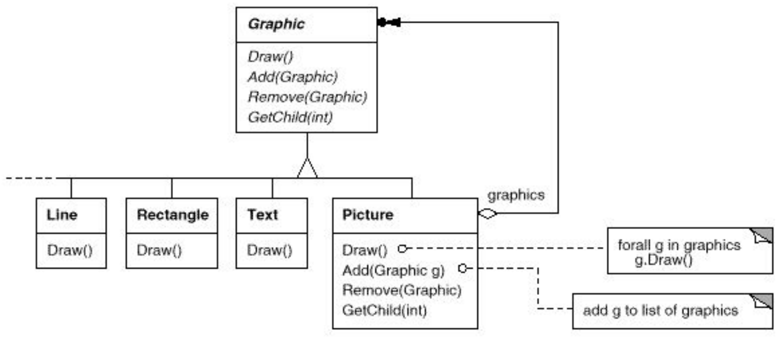
- 

- **Participants**
  - **Abstraction** (Window)
    - defines the abstraction's interface
    - maintains a reference to an object of type Implementor
  - **RefinedAbstraction** (IconWindow)
    - extends the interface defined by Abstraction
  - **Implementator** (WindowImp)
    - defines the interface for implementation classes. This doesn't have to correspond exactly to Abstraction's interface; in fact the 2 interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives
  - **ConcreteImplementor** (XWindowImp, PMWindowImp)
    - implements the IMplementor interface and defines its concrete implementation
- **Collaborations**
  - Abstraction forwards client requests to its Implementor object
- **Consequences**
  - *Decoupling interface and implementation* - Implementation is not bound permanently to an interface and implementation of an abstraction can be changed at runtime. This also eliminates compile-time dependencies on the implementation. Further, this decoupling encourages layering that can lead to a better structured system. The high-level part of a system only has to know about Abstraction and Implementor.
  - *Improved extensibility* - independent extension of Abstraction & Implementor
  - *Hiding implementation details from client*
- **Implementation**
  - *Only one Implementor* - In case there's only one implementation an abstract Implementor isn't necessary, nevertheless, this separation is still useful when a change in the implementation of a class must not affect its existing clients - that is, they shouldn't have to be recompiled, just relinked
  - *Creating the right Implementor object* - If Abstraction knows about all ConcreteImplementor then it can instantiate one of them in its constructor and decide b/w them based on parameters passed to its constructor (like

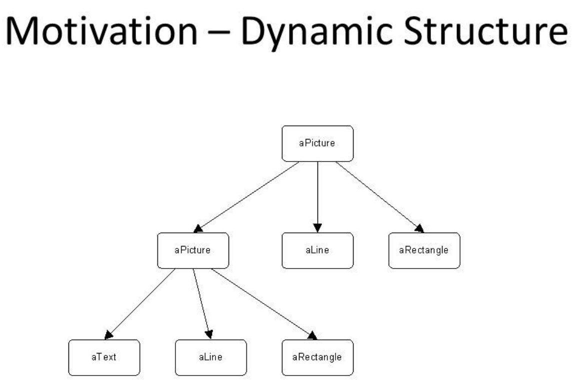
collection class can use Linked list / hash table based on size). This decision can also be delegated using Abstract Factory
    - *Sharing implementers* - Handle / Body idiom in C++
    - *Using multiple inheritance* - you can't implement a true Bridge with multiple inheritance in C++ because it relies on static inheritance, it binds an implementation permanently to its interface
- **Related Patterns**
    - An Abstract Factory can create and configure a particular Bridge
    - An Adaptor pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently

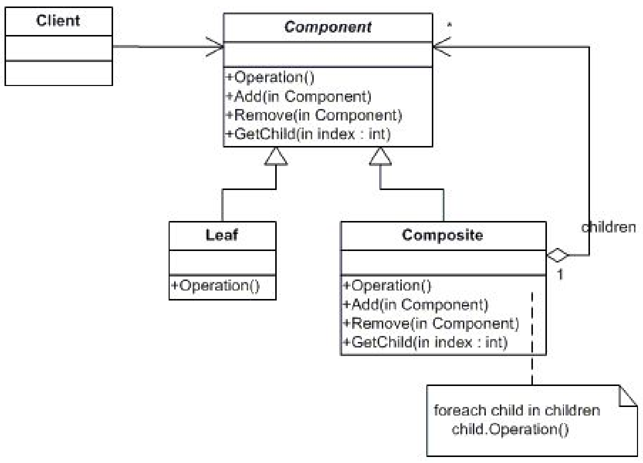
Composite (Object Structural)

- **Intent**
    - Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Motivation**
    - Graphic applications with several components where users treat them all as the same but the code needs to treat primitive and complex objects differently leading to complex code.



    -



- **Applicability**
    - want to represent part-whole hierarchies of objects
    - want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- **Structure**
    - 

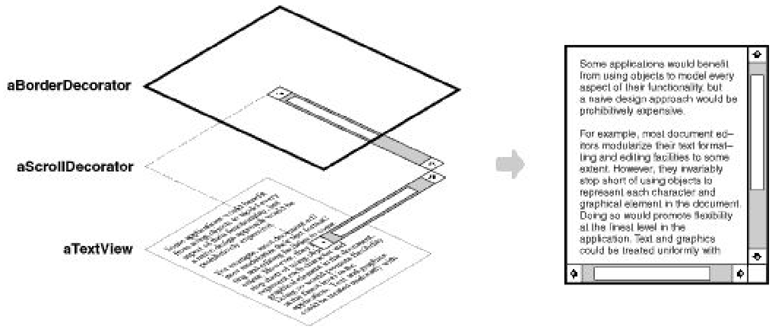
- **Participants**
    - **Component** (Graphic)
        - declares the interface for objects in the composition.
        - implements default behaviour for the interface common to all classes, as appropriate.
        - declares an interface for accessing and managing its child components.
        - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
    - **Leaf** (Rectangle, Line, Text, etc.)
        - represents leaf objects in the composition. A leaf has no children.
        - defines behaviour for primitive objects in the composition.
    - **Composite** (Picture)
        - defines behaviour for components having children.
        - stores child components.
        - implements child-related operations in the Component interface.
    - **Client**
        - manipulates objects in the composition through the Component interface.
- **Collaborations**
    - Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.
- **Consequences**

    - defines hierarchies consisting primitive/composite objects, both can be composed into more complex objects and client can take any one of them where it expects the other
    - makes clients simple by treating composite and individual structure objects uniformly
    - makes it easier to add new kinds of components. Clients need not be changed for newly defined Composite or Leaf subclasses
    - can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.
- **Implementation**
    - Issues to consider when implementing:
    - *Explicit parent references* - Maintaining parent references simplifies traversal, moving up the structure and deleting a component, it also helps support Chain of Responsibility pattern. It's essential that the parent is maintained correctly and this can be achieved using Add/Remove operations.
    - *Sharing components* - Sharing components to save space but it becomes difficult when a component can have no more than one parent. A possible solution is to store several patterns but that leads to ambiguities, the Flyweight pattern shows how to rework a design to avoid storing parents altogether. It works in cases when children can avoid sending parent requests by externalizing some or all of their state.
    - *Maximizing the Component interface* - One of the goals is to make clients unaware of the specific Leaf or Composite classes they're using by defining as many common operations for Composite and Leaf classes as possible. The component class usually provides overridable default implementation for these operations. But this conflicts with the principle of class hierarchy design that says a class should only define operations that are meaningful to its subclasses, this can be solved by viewing Leaf as a Component, like it can *never* have any children and thus returns no children but Composite class overrides this behaviour to return children.
    - *Declaring the child management operations* - Defining child management interface at the root of the class hierarchy gives transparency but compromises safety as clients might do meaningless things like add and remove objects from leaves, defining child management in the Composite class give you safety but this loses transparency as leaf and composite will have separate interfaces. Prefer transparency over safety, to provide safety you can add a method to check whether it's composite or not and use the operations accordingly (similar tests can be done by dynamic_cast check like instanceof). To provide transparency default Add and Remove operations need to be provided and it's better to fail these operations for Leaf (rather than leaving them unimplemented) because unwanted usage may indicate a bug in the client. Remove operation can be dealt with by changing its

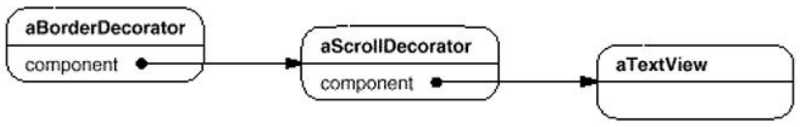
definition asking it to remove itself from its parent but still Add operation remains a problem.
    - *Should Component implement a list of Components* - Defining the set of children as an instance variable in the Component class where the child access and management operations are declared incurs a space penalty for every leaf this might be worthwhile if there are relatively few children in the structure
    - *Child ordering* - When ordering is crucial child access & management operations should be carefully declared, Iterator pattern can help here
    - *Caching to improve performance* - In case of frequent traversals results can be stored and to invalidate components can know their parents. You need to define an interface to invalidate the cache by telling composites that they are no longer valid.
    - *Who should delete components* - In languages without garbage collection, Composites should be responsible for its children when it's destroyed. Exception is when Leaf objects are immutable and thus can be shared.
    - *What's the best structure for storing components* - Different data-structures might be used for required efficiency, sometimes each subclass of Composite might implement its own management interface. See interpreter for an example.
- **Related Patterns**
    - Often the component-parent link is used for a Chain of Responsibility
    - Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
    - Flyweight lets you share components, but they can no longer refer to their parents.
    - Iterator can be used to traverse composites.
    - Visitor localizes operations and behaviour that would otherwise be distributed across Composite and Leaf classes.

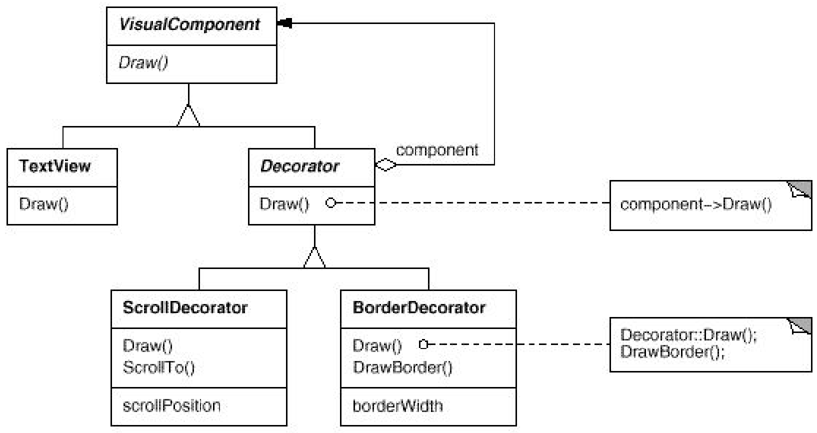
Decorator (Object Structural)

- **Intent**
    - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Also Known As**
    - Wrapper
- **Motivation**
    - When we want to add responsibilities to individual objects and not to the entire class.
    - Like border, Inheriting a border from another class puts border around every subclass instance
    - Inheritance is also inflexible because border is made statically and clients can't control it

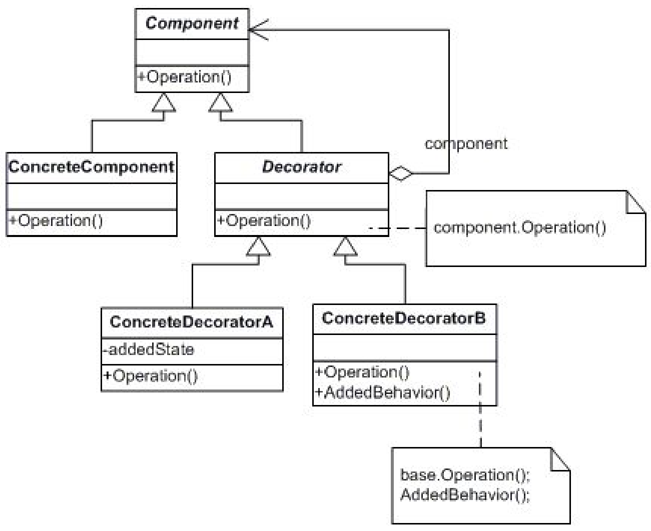
- Flexible approach is to enclose the component in another object (**decorator**) that adds the border
- Decorator confirms to the interface of the component it decorates so that its presence is transparent to the component's clients
- Decorator forwards requests to the component and may perform additional actions (before or after forwarding)
- Transparency lets you add nest decorator recursively, thereby allowing an unlimited number of added responsibilities
- They also provide additional method so if clients are aware of the decorator in the interface they can use the operations
- 

Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with
- 

- 

- **Applicability**
  - to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects
  - for responsibilities that can be withdrawn
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.
- **Structure**



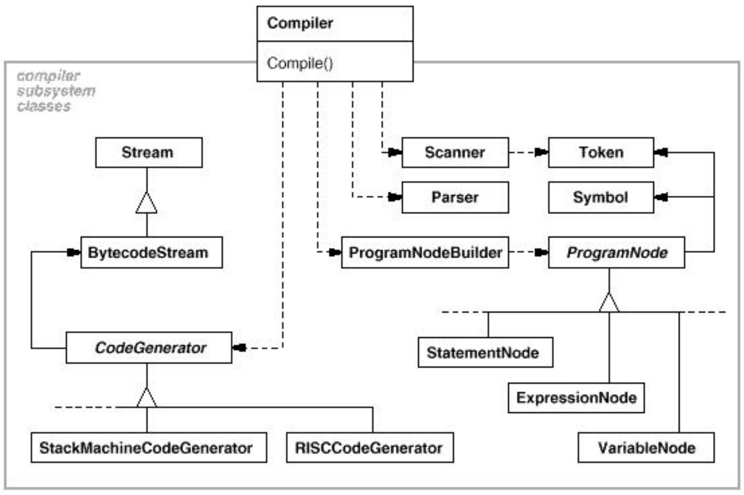
- **Participants**
    - **Component** (VisualComponent) - defines the interface for objects that can have responsibilities added to them dynamically.
    - **ConcreteComponent** (TextView) - defines an object to which additional responsibilities can be attached
    - **Decorator** - maintains a reference to a Component object and defines an interface that conforms to Component's interface
    - **ConcreteDecorator** (BorderDecorator, ScrollDecorator) - adds responsibilities to the component.
- **Collaborations**
    - Decorator forwards request to its Component object. It may optionally perform additional operations before and after forwarding the request.
- **Consequences**
    - 2 key benefits & 2 liabilities:
        - *More flexibility than static inheritance* - Responsibilities can be added/removed at run-time by simply attaching and detaching them. (Inheritance creates a lot many classes & increases the complexity of a system.) Decorator classes allow mix and match responsibilities. It also makes it easy to add a property twice.
        - *Avoids feature-laden classes high up in the hierarchy* - Instead of supporting all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions.
        - *A decorator and its component aren't identical* - A decorated component is not identical to the component itself (even though it's a transparent enclosure). Hence you shouldn't rely on object identity when you use decorators.
        - *Lots of little objects* - A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to

customize by those who understand them, they can be hard to learn and debug.

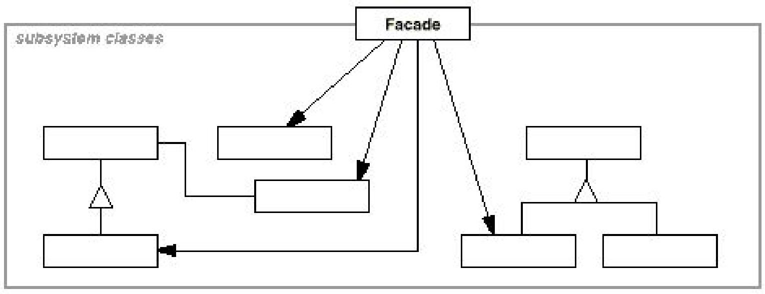
- **Implementation**
  - *Interface conformance*
  - *Omitting the abstract Decorator class* - If only one Decorator class is needed then abstract Decorator's responsibility of forwarding request can be merged with ConcreteDecorator
  - *Keeping Component classes lightweight* - Common class (interface) should be lightweight and focus on defining interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.
  - *Changing the skin of an object versus changing its guts* - Strategies are a better choice when the Component class is intrinsically heavyweight, thereby making the Decorator pattern costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.
- **Related Patterns**
  - Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.
  - Composite: A decorator can be viewed as a degenerate composite with only one component. However, decorator adds additional responsibilities - it isn't intended for object aggregation.
  - Strategy: A decorator lest you change the skin of an object; a strategy lets you change the guts. These are 2 alternative ways of changing an object.

Facade (Object Structural)

- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem. Faced defines a higher level interface that makes the subsystem easier to use.
- **Motivation**
  - Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems.
  - Example Compiler interface to hide details of it's components from clients that don't need to know about them

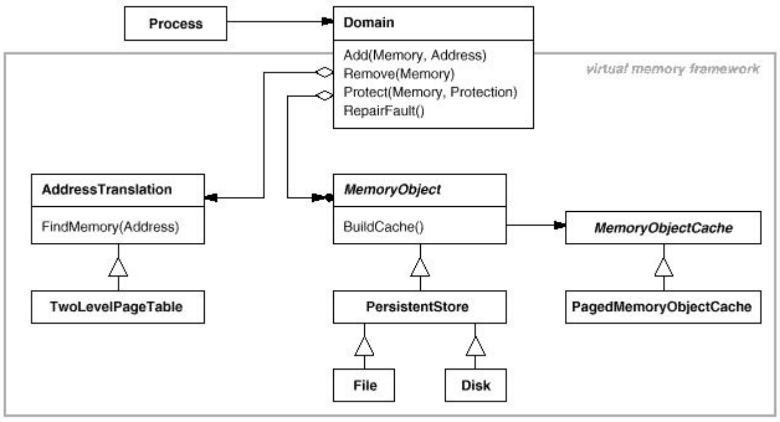


- **Applicability**
  - Use the Facade pattern when
    - provide a simple interface to a complex subsystem. Most patterns result in more and smaller classes.This makes the subsystem more reusable and easier to customize, but it becomes harder for clients that don't need to customize it.
    - there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from client and other subsystems, thereby promoting subsystem independence and portability
    - you want to layer your subsystem. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.
- **Structure**



- **Participants**
  - **Facade** (Compiler)
    - knows which subsystem classes are responsible for a request
    - delegates client request to appropriate substem objects
  - **subsystem classes** (Scanner, Parser, ProgramNode, etc)
    - implement subsystem functionality
    - handle work assigned by the Facade object
    - have no knowledge of the facade; that is, they keep no references to it
- **Collaborations**

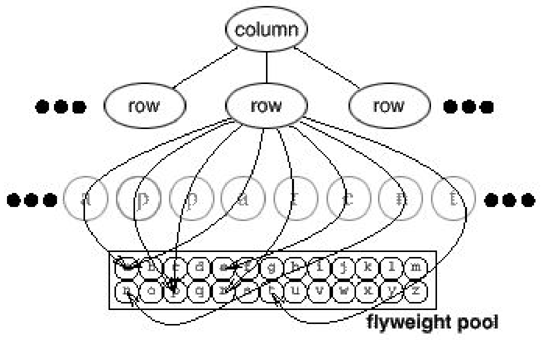
    - Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
    - Clients that use the faade don’t have to access its subsystem objects directly.
- **Consequences**
    - Shields clients from subsystem components, thus reducing the no. of objects clients deal with, making the subsystem easier to use
    - Promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.
      Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it’s less likely that building one subsystem requires building all others.
    - It doesn’t prevent application from using subsystem classes if they need to. Thus you can choose between ease of use and generality.
- **Implementation**
    - *Reducing client-subsystem coupling* - Coupling can be further reduced by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Clients communicate through the interface of the abstract Facade class which keeps them knowing the implementation of the subsystem.
      An alternative to subclassing is to configure a Facade object with different subsystems objects. To customize the facade, simply replace one or more of its subsystems objects.
    - *Public versus private subsystem classes* - The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, of course, but it’s not the only. Other subsystem classes are usually public as well. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.
- **Known Uses**

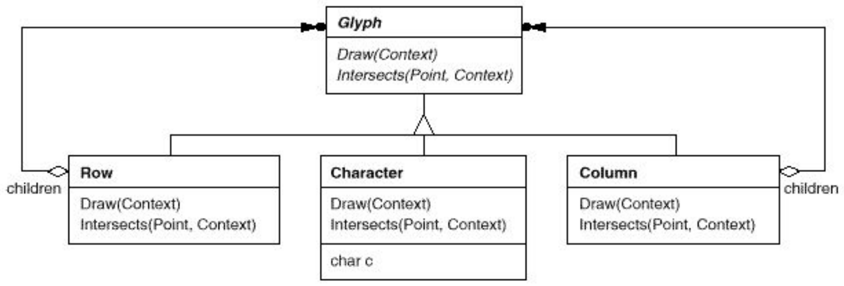


- **Related Patterns**
  - Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.
  - Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.
  - Usually only one Facade object is required. Thus Facade objects are often Singletons.

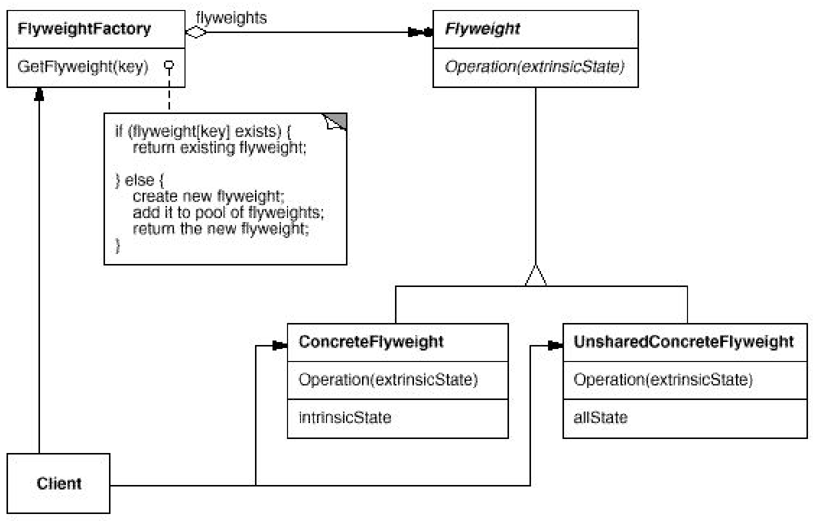
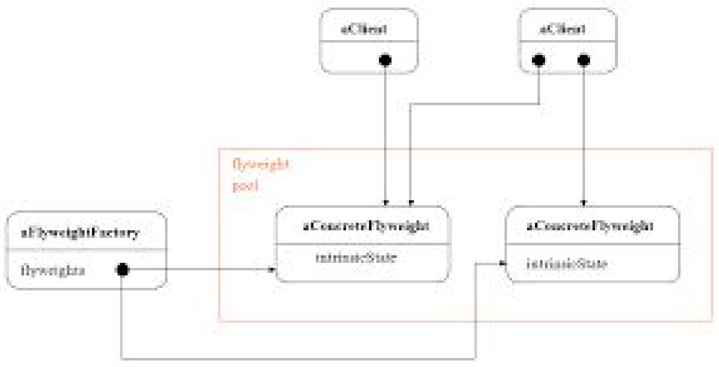
Flyweight (Object Structural)

- **Intent**
  - Use sharing to support large numbers of fine-grained objects efficiently
- **Motivation**
  - Document which uses objects to represent
  - Requires fine grained formatting and control but the cost should be low
  - A **flyweight** is a shared object that can be used in multiple contexts simultaneously
  - The flyweight acts as an independent object in each context - it's indistinguishable from an instance object that's not shared
  - Flyweights cannot make assumptions about the context
  - **Intrinsic** state is stored in the flyweight; it consists of information that's independent of the flyweights's context, thereby making it shareable
  - **Extrinsic** state depends on and varies with the flyweight's context and therefore can't be shared
  - Client objects are responsible for passing extrinsic state to the flyweight when it needs it

- Flyweight model concepts or entities that are too plentiful to represent with objects
- A document can create flyweight for each letter, coordinate position/typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears
- 

- 
Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
children
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
- A flyweight representing a letter only stores the corresponding character code; clients supply the context-dependent information

- **Applicability**
  - Apply when all of the following are true:
    - An application uses a large no. of objects
    - Storage costs are high because of the sheer quantity of objects
    - Most object sate can be made extrinsic
    - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
    - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.
- **Structure**

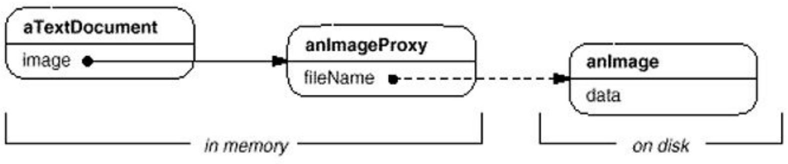
- 
FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client
- 

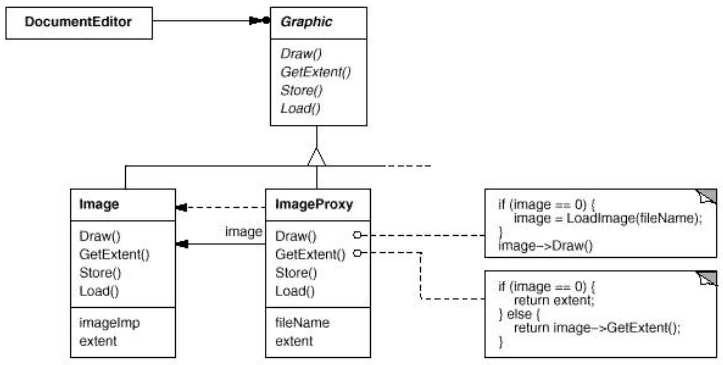
- **Participants**
  - **Flyweight** (Glyph)
    - declares an interface through which flyweights can receive and act on extrinsic state
  - **ConcreteFlyweight** (Character)
    - implements the Flyweigh interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be shareable.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

        - maintains a reference to flyweight(s).
        - computes or stores the extrinsic state of flyweight(s).
- **Collaborations**
    - States must be characterized as intrinsic/extrinsic. Intrinsic is stored in the ConcreteFlyweight object; extrinsic is stored or computed by Clients, clients pass this state to the flyweight when they invoke its operations.
    - Clients **must** obtain ConcreteFlyweight objects using FlyweightFactory
- **Consequences**
    - Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.
    - Storage savings are function of:
        - the reduction in the total no. of instances that comes from sharing
        - the amount of intrinsic state per object
        - whether extrinsic state is computed or stored
    - When intrinsic state is shared and extrinsic state is computed (instead of being stored) storage savings are substantial
    - The Flyweight pattern is often combined with the Composite pattern to represent a hierarchical structure as a graph with shared leaf nodes. A consequence of sharing is that flyweight leaf nodes cannot store a pointer to their parent. Rather, the parent pointer is passed to the flyweight as part of its extrinsic state. This has a major impact on how the objects in the hierarchy communicate.
- **Implementation**
    - *Removing extrinsic state* - Removing extrinsic state can help if it can be computed using a separate object structure and there aren't many different kinds of them
    - *Managing shared objects* - FlyweightFactory to create shareable objects and shareability implies reference counting or garbage collection to reclaim a flyweight's storage (when it's no longer is needed and space occupied is substantial)
- **Related Patterns**
    - The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes
    - It's often best to implement State and Strategy objects as flyweights

Proxy (Object Structural)

- **Intent**
    - Provide a surrogate or placeholder for another object to control access to it
- **Also Known As**
    - Surrogate
- **Motivation**

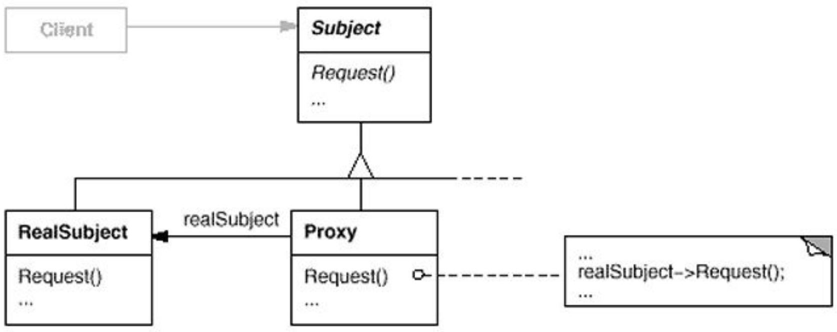
    - To control access to an expensive object so that it's creation and initialization can be deferred until used
    - Expensive object to be created on *demand* and we need to place something else in its place
    - We need to hide the fact that the object is created on demand so that we don't complicate the editor's implementation (this optimization shouldn't impact the rendering and formatting code)
    - Solution is to use **proxy**, that acts as a stand-in for the real object, it acts just like the object and takes care of instantiation when required
    - 

    - Once the object is instantiated the proxy forwards subsequent requests directly to the object and thus must retain a reference to it
    - The proxy can store additional data that can be required & supplied without instantiation
    - 

```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

- **Applicability**
    - Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer:
        - A **remote proxy** provides local representative for an object in different address space
        - A **virtual proxy** creates expensive objects on demand
        - A **protection proxy** are used when objects should have different access rights
        - A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typically:
            - counting the no. of references to the real object so that it can be freed automatically when there are no more references
            - loading a persistent object into memory when it's first referenced
            - check that the real object is locked before it's accessed to ensure that no other object can change it
- **Structure**



-
- **Participants**
    - **Proxy** (Image Proxy)
        - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
        - provides an interface identical to Subject's so that a proxy can be substituted for the real subject
        - controls access to the real subject and may be responsible for creating and deleting it.
        - other responsibilities depend on the kind of proxy:
            - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space
            - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
            - *protection proxies* check that the caller has the access permissions required to perform a request.
    - **Subject** (Graphic)
        - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
    - **RealSubject** (Image)
        - defines the real object that the proxy represents.
- **Collaborations**
    - Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.
- **Consequences**
    - A remote proxy can hide the fact that an object resides in a different address space
    - A virtual proxy can perform optimizations such as creating an object on demand
    - Both protection and smart references allow additional housekeeping tasks when an object is accessed
    - **Copy-on-write** - for this to work the subject must be reference counted, only when the client requests an operation that modifies the subject does the

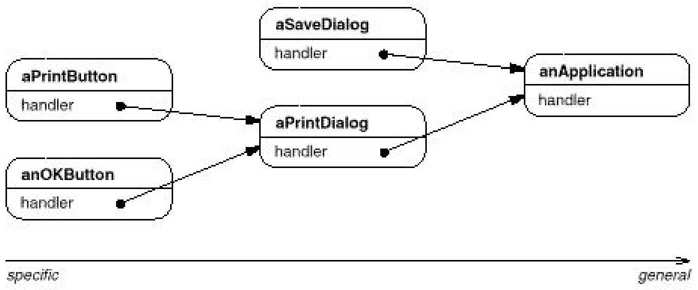
    proxy actually copy it (and decrements the subject's reference counter). When the reference count goes to zero, the subject is deleted.

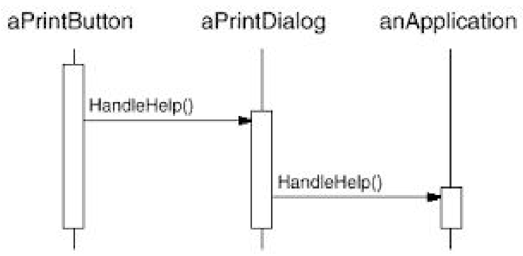
- **Implementation**
    - The proxy pattern can exploit the following language features:
        - *Overloading the member access operator (in C++)* - Overloading the member access operator (operator ->) lets you perform additional work whenever an object is dereferenced (the proxy behaves just like a pointer). This lets you call RealSubject operations through Proxy without going to the trouble of making the operations part of the Proxy interface. In this case both Proxy and RealSubject needs to be handled differently.
        - *Using* **doesNotUnderstand** *in Smalltalk* - Provides a hook that can be used to support automatic forwarding of requests. Smalltalk calls doesNotUnderstand: aMessage when a client sends a message to a receiver that has no corresponding method. The Proxy class can redefine doesNotUnderstand so that the message is forwarded to its subject. (See book for more details)
        - *Proxy doesn't always have to know the type of real subject* - If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.
        - Another implementation issue is how to refer to the subject before it's instantiation (use address space-independent object identifier like file name)
- **Related Patterns**
    - Adapter: An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.
    - Decorator: Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.
    - Proxies vary in the degree to which they are implemented like a decorator. A protection proxy might be implemented exactly like a decorator. On the other hand, a remote proxy will not contain a direct reference to its real subject but only an indirect reference, such as "host Id and local address on host". A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

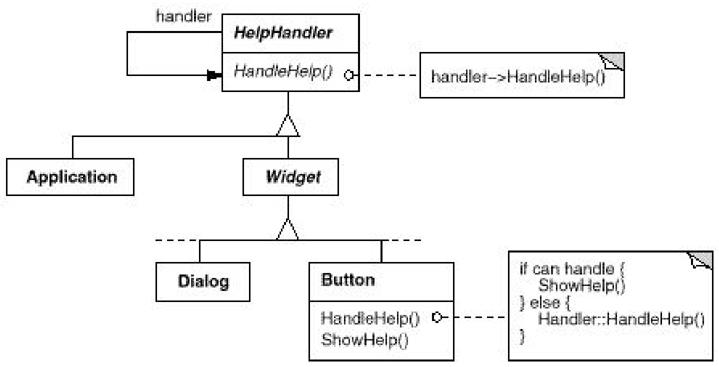
Discussion of Structural Patterns

5. Behavioral Patterns

Chain of Responsibility (Object Behavioral)

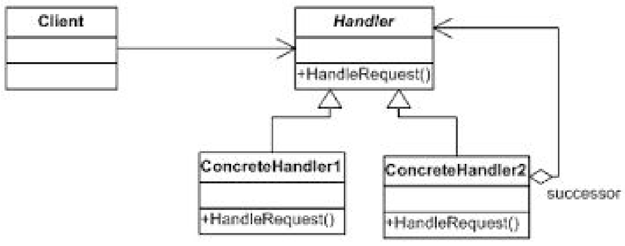
- **Intent**
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chaining the receiving objects and passing the request along the chain until an object handles it.
- **Motivation**
  - Context-sensitive help facility for a gui where the user can obtain help information on any part of the interface and if it's not available then a more general help information can be given
  - 

  - 

  - 

- **Applicability**
  - more than 1 object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically

- you want to issue a request to one of several objects without specifying the receiver explicitly
- the set of objects that can handle a request should be specified dynamically

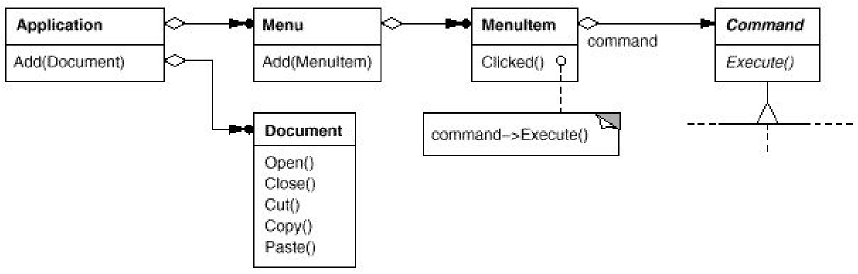
- **Structure**
  - 

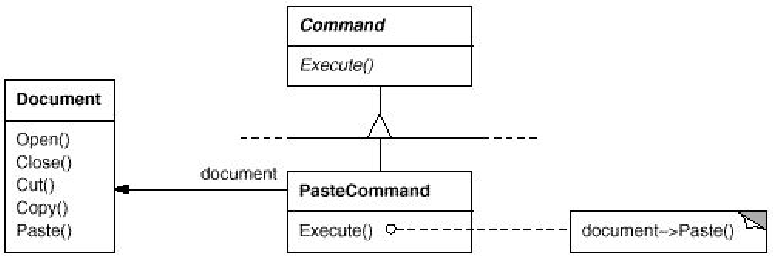
- **Participants**
  - **Handler** (HelpHandler)
    - defines an interface for handling requests
    - (optional) implements the successor link
  - **ConcreteHandler** (PrintButton, PrintDialog)
    - handles requests it is responsible for
    - can access its successor
    - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to it successor
  - **Client**
    - initiates the request to a ConcreteHandler object on the chain
- **Collaborations**
  - When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.
- **Consequences**
  - *Reduced coupling* - Receiver & Sender doesn't each other explicitly, an object in the chain doesn't have to know about the chain's structure. Chain of Responsibility can simplify object interconnection. Instead of objects maintaining references to all candidate receivers, they keep a single reference to their successor.
  - *Added flexibility in assigning responsibilities to objects* - You can distribute, add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialize handlers statically.
  - *Receipt isn't guaranteed* - A request can go unhandled when the chain is not configured properly or it falls off the end without being handled
- **Implementation**
  - *Implementing the successor chain* - 2 ways to implement successor chain:
    - Define new links (usually in the Handler, but ConcreteHandlers could define them instead)
    - Use existing links - existing links can be used if such a hierarchy exists (composite pattern) and is similar to required chain of responsibility
  - *Connecting successors* - If defining new links then Handler not only defines the interface for requests but usually maintains the successor as well. That lets the handler provide a default implementation of HandleRequest that

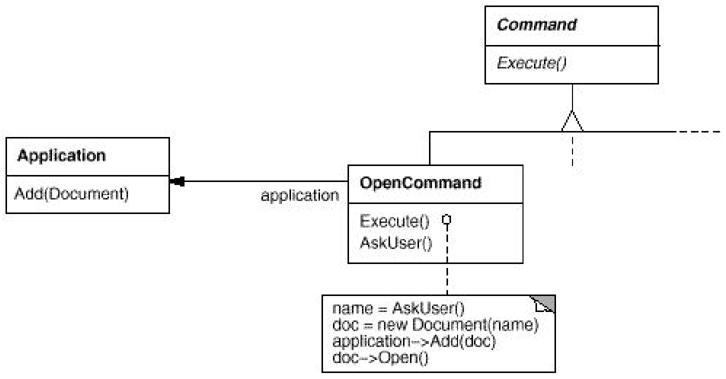
forwards the request to the successor (if any). If a ConcreteHandler subclass isn't interested in the request, it doesn't have to override the forwarding operation since its default implementation forwards unconditionally.
    - *Representing requests* - Either have hard-coded operations which are convenient and safe but only a fixed set of requests can be defined or have request code (sender and receiver need to agree) & pass that. The second option is flexible but it requires conditional statements for dispatching the request based on its code. Moreover, there's no type-safe way to pass parameters, so they must be packed and unpacked manually. We can have separate request objects that bundle request parameters. Requests can be subclassed and provide an identifier (or the receiver can use runt-time information).
    - *Automatic forwarding in Smalltalk* - doesNotUnderstant mechanism to forward requests
- **Related Patterns**
    - Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

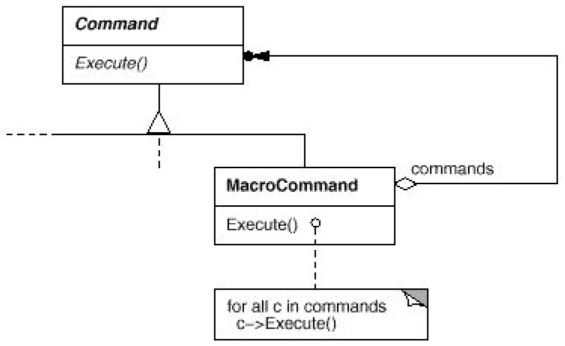
Command (Object Behavioral)

- **Intent**
    - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- **Also Known As**
    - Action, Transaction
- **Motivation**
    - Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request
    - User interface toolkits include objects like buttons and menus that carry out a request in response to user input. But the toolkit can't implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which object. As toolkit designers have no way of knowing the receiver of the request or the operations that will carry it out.
    - The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects.
    - The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation.
    - Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing Execute to invoke the request.
    - The receiver has the knowledge required to carry out the request.

- 

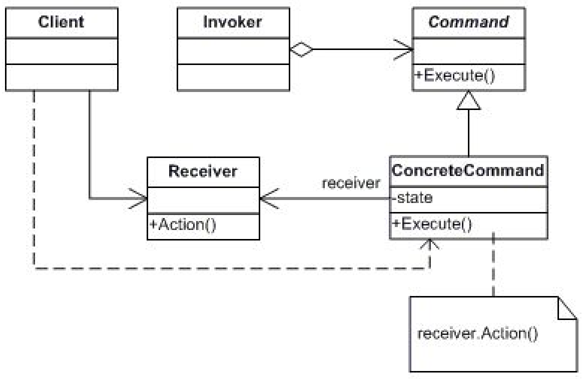
- 

- 

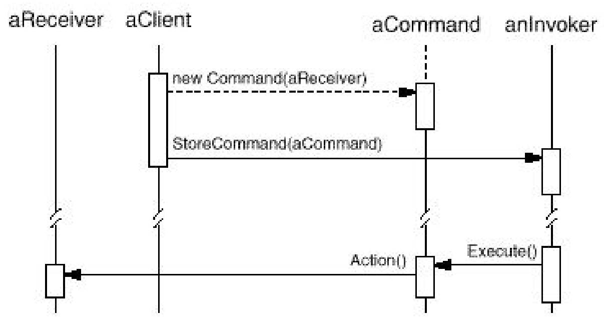
- MacroCommand to execute a sequence of Commands
- 

- In the examples, above the Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it
- We can replace commands dynamically, which would be useful for implementing context-sensitive menus

- **Applicability**
  - parameterize objects by an action to perform. You can express such parameterization in a procedural language with a **callback** function, *i.e.*, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement of callbacks.

    - specify, queue and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
    - support undo. The Commands' Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute.
    - support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and re-executing them with the Execute operation.
    - structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support **transactions**. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.
- **Structure**
    - 

- **Participants**
    - **Command** - declares an interface for executing an operation
    - **ConcreteCommand** (PasteCommand, OpenCommand)
        - defines a binding between a Receiver object and an action
        - implements Execute by invoking the operation(s) on Receiver
    - **Client** (Application) - creates ConcreteCommand object and sets its receiver
    - **Invoker** (MenuItem) - asks the command to carry out the request
    - **Receiver** (Document, Application) - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.
- **Collaborations**
    - The client creates a ConcreteCommand object and specifies the receiver.
    - The Invoker object stores the ConcreteCommand object.

- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.
- 

- **Consequences**
  - Command decouples the object that invokes the operation from the one that knows how to perform it
  - Commands are first-class objects. They can be manipulated and extended.
  - You can assemble commands into a composite command like, MacroCommand or in general, instances of Composite pattern.
  - Easy to add new Commands, because existing class is not to be changed.
- **Implementation**
  - *How intelligent should a command be?* - Command can just define a binding between a receiver and the actions that carry out the request or it can implement everything itself without delegating to a receiver at all.
  - *Supporting undo and redo* - ConcreteCommand class needs to store additional state to do the reverse. The state includes:
    - the Receiver object, which actually carries out operations in response to the request
    - the arguments to the operation performed on the receiver
    - any original values in the receiver that can change as a result

    An undoable command might have to be copied before it can be placed on the history list. That's because the command object that carried out the original request will perform other requests at later times. Copying is required to distinguish different invocations of the same command if its state can vary across invocations. Example DeleteCommand.
  - *Avoiding error accumulation in the undo process* - Error can accumulate due to repeated un-execution and re-execution of commands and state might diverge from the original values. The Memento pattern can be applied to give the command access to this information without exposing the internal of other objects.
  - *Using C++ templates* - For commands that (1) aren't undoable and (2) don't require arguments can use C++ templates. See the book for more.
- **Related Patterns**
  - A Composite can be used to implement MacroCommands
  - A Memento can keep state the command requires to undo its effect

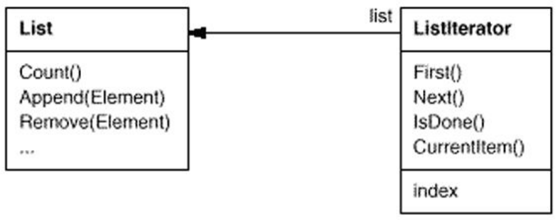
    - A command that must be copied before being placed on the history list acts as a Prototype

Interpreter (Class Behavioral) Skipped

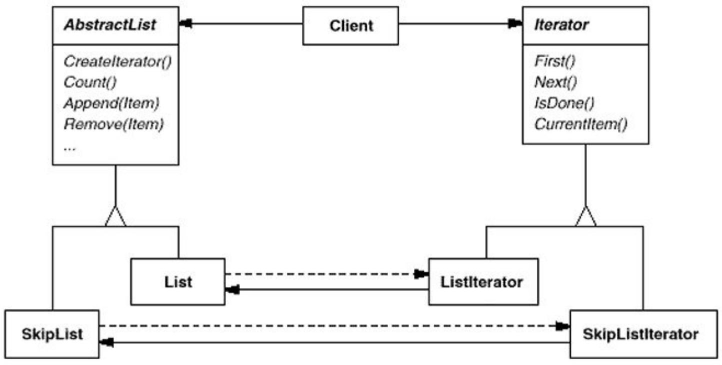
- **Intent**
    - Given a language, define a representations for its grammar along with an interpreter that uses the representation to interpret sentences in the language
- **Motivation**
    - The pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences
- **Applicability**
- **Structure**
- **Participants**
- **Collaborations**
- **Consequences**
- **Implementation**
- **Related Patterns**

Iterator (Object Behavioral)

- **Intent**
    - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- **Also Known As**
    - Cursor
- **Motivation**
    - The key idea is to take the responsibility for access and traversal out of the object (list) and put it in an iterator object

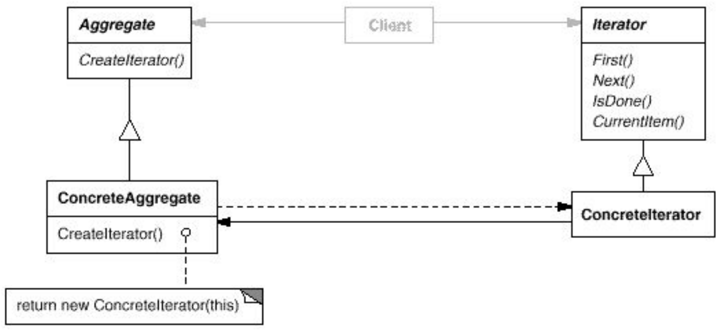


    -
    - To decouple list and iterator we need to generalize the iterator concept to support polymorphic iteration
    - We define an AbstractList class for a common interface and define Iterator subclasses for different list implementations. As a result, the iteration mechanism becomes independent of concrete aggregate classes.



- ○
- ○ So, we make the list objects responsible for creating their corresponding iterator

- **Applicability**
    - to access an aggregate object's contents without exposing its internal representation
    - to support multiple traversal of aggregate objects
    - to provide a uniform interface for traversing different aggregate structure (that is, to support polymorphic iteration).
- **Structure**



- ○

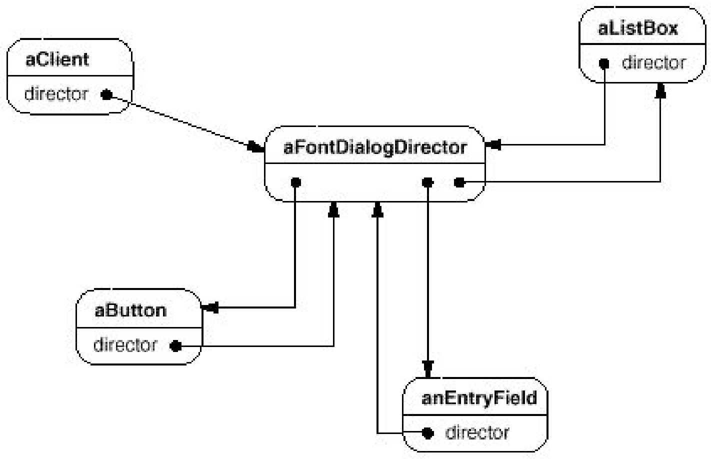
- **Participants**
    - **Iterator** - defines an interface for accessing and traversing elements
    - **ConcreteIterator**
        - implements the Iterator interface
        - keeps track of the current position in the traversal of the aggregate
    - **Aggregate**
        - defines an interface for creating an Iterator object
    - **ConcreteAggregator**
        - implements the Iterator creation interface to return an instance of the proper ConcreteIterator
- **Collaborations**
    - A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- **Consequences**
    - *It supports variations in the traversal of an aggregate* - Complex aggregates may be traversed in many ways (eg preorder or inorder)

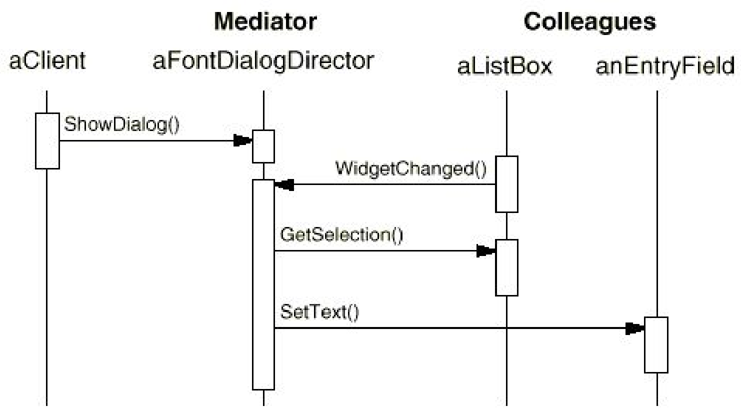
    - *Iterators simplify the Aggregate interface* - Iterators' traversal interface obviates the need for a similar interface in Aggregate, thereby simplify the aggregate's interface
    - *More than one traversal can be pending on an aggregate* - An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.
- **Implementation**
    - *Who controls the iteration?* - The iterator or the client? When the client controls the iteration, the iterator is called an **external iterator**, otherwise **internal iterator**. Internal iterators are easier to use but certain use cases are restricted like comparing 2 lists.
    - *Who defines the traversal algorithm?* - The iterator is not the only place where the traversal algorithm can be defined. The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration (this is known as **cursor**).
      If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates. On the other hand, the traversal algorithm might need to access the private variables of the aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.
    - *How robust is the iterator?* - Dangerous to modify an aggregate while traversing. A **robust iterator** ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate. Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.
    - *Additional Iterator operations* - Minimal interface - First, Next, IsDone, CurrentItem. Previous operation might also be supported, SkipTo for sorted and indexed collections.
    - *Using polymorphic iterators in C++* - Polymorphic iterators have their cost. They require the iterator object to be allocated dynamically by a factory method. Hence they should be used only when there's a need for polymorphism. Otherwise use concrete iterators, which can be allocated on the stack.
      Polymorphic iterators have another drawback: the client is responsible for deleting items. This is error-prone, because it's easy to forget to free a heap-allocated iterator object when you're finished with it. That's especially likely when there are multiple exit points in an operation. And if an exception is triggered, the iterator object will never be freed.
      Proxy pattern provides a remedy. We can use a stack allocated proxy as a stand-in for the real iterator. The proxy deletes the iterator in its destructor. Thus when the proxy goes out of scope, the real iterator will get deallocated along with it. The proxy ensures proper cleanup, even in the face of exceptions.

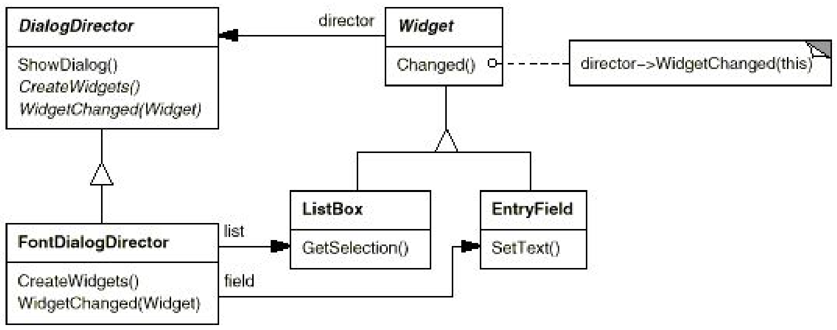
    - *Iterators may have privileged access* - An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled.
      We can express the close relationship in C++ by making the iterator a *friend* of its aggregate. Then you don't need to define aggregate operations whose sole purpose is to let iterators implement traversal efficiently.
      However, such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend. To avoid the problem, the Iterator class can include *protected* operations for accessing important but publicly unavailable members of the aggregate. Iterator subclasses (and *only* Iterator subclasses) may use these protected operations to gain privileged access to the aggregate.
    - *Iterators for composites* - External iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates. Therefore an external iterator has to store a path through the Composite to keep track of the current object.
    - *Null iterators* - **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is *always* done with traversal.
- **Related Patterns**
    - Composite: Iterators are often applied to recursive structures such as Composites.
    - Factory Method: Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.
    - Memento is often used in conjunction with the Iterator pattern. An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.

Mediator (Object Behavioral)

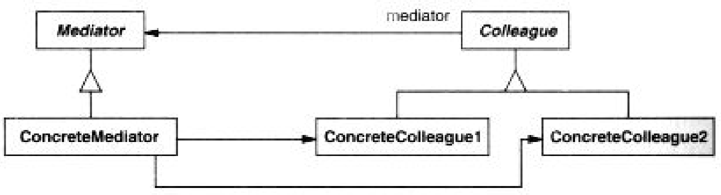
- **Intent**
    - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Motivation**
    - Object-oriented design has distribution of behaviour among objects, this can result in a structure with many connections & every object ends up knowing about every other
    - A lot of connections make it less likely that an object can work without the support of others - the system acts as though it were monolithic.
      It can be difficult to change the system's behavior in any significant way, since behavior is distributed among many objects.
    - You may be forced to define many subclasses to customize the system's behaviour
    - Collective behavior can be encapsulated in a separate **mediator** object

- The objects know only the mediator thereby reducing the no. of interactions
- 

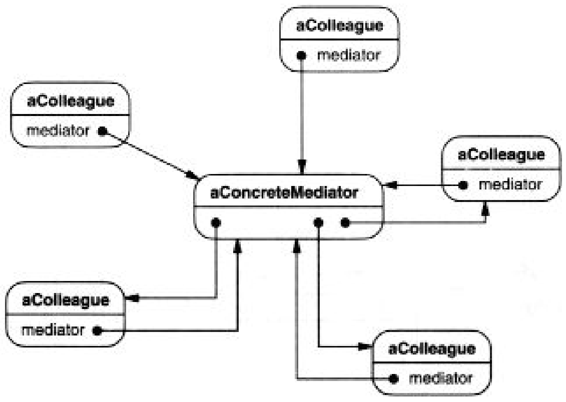
- 

- 

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



A typical object structure might look like this:

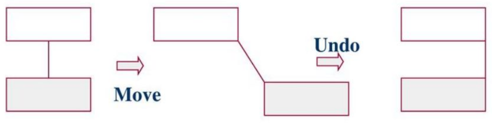


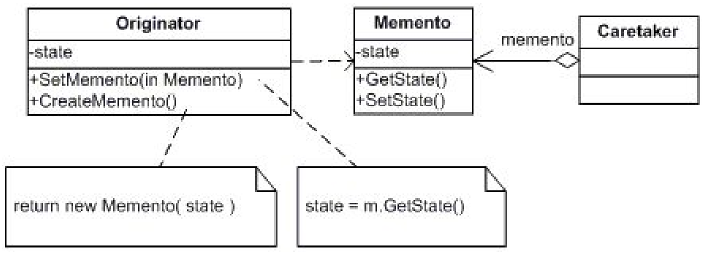
- ○
- **Participants**
  - **Mediator** (DialogDirector)
    - defines an interface for communicating with Colleague objects
  - **ConcreteMediator** (FontDialogDirector)
    - implements cooperative behavior by coordinating Colleague objects
    - knows and maintains its colleagues
  - **Colleague classes** (ListBox, EntryField)
    - each Colleague class knows its Mediator object
    - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague
- **Collaborations**
  - Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).
- **Consequences**
  - *It limits subclassing* - Behavior can be changed by subclassing Mediator only
  - *It decouples colleagues*
  - *It simplifies object protocols* - Many-to-many interactions are simplified to One-to-many
  - *It abstracts how objects cooperate* - Clarifies how objects interact apart from their individual behavior
  - *It centralizes control* - Mediator pattern trades complexity of interaction for complexity n the mediator, hence, sometimes it can become hard to maintain monolith
- **Implementation**
  - *Omitting the abstract Mediator class*
  - *Colleague-Mediator communication* - Colleagues have to communicate with their mediator when an event of interest occurs so Mediator can be

implemented by Observer pattern. Another approach is delegation, colleague passes itself as an argument and mediator identifies it.

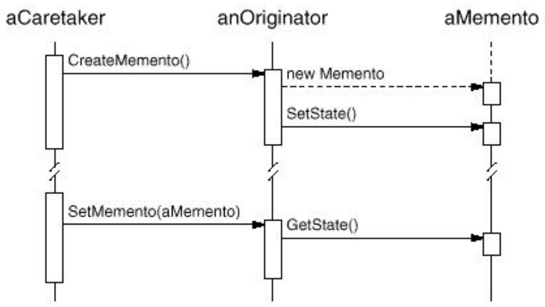
- **Related Patterns**
    - Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.
    - Colleagues can communicate with the mediator using the Observer pattern

Memento (Object Behavioral)

- **Intent**
    - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
- **Also Known As**
    - Token
- **Motivation**
    - Saving internal state of an object to set checkpoints for error recovery & undo
    - Sometimes undo mechanism to restore previous state through calculations might be much harder
    - 

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

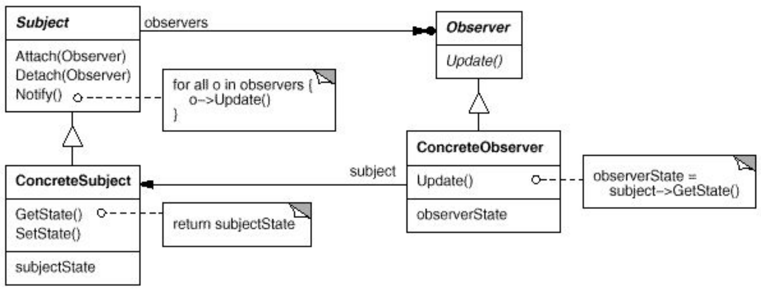
- **Participants**
    - **Memento** (SolverState)
        - store internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion
        - protects against access by objects other than the originator. Mementors have effectively two interfaces. Caretaker sees a *narrow* interface to the Memento - it can only pass the memento to other objects. Originator, in contrast, sees a *wide* interface, one that lets it

        access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.
    - **Originator** (ConstraintSolver)
        - creates a memento containing a snapshot of its current internal state
        - uses the memento to restore its internal state
    - **Caretaker** (undo mechanism)
        - is responsible for the memento's safekeeping
        - never operates on or examines the contents of a memento
- **Collaborations**
    - A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator:
    - 

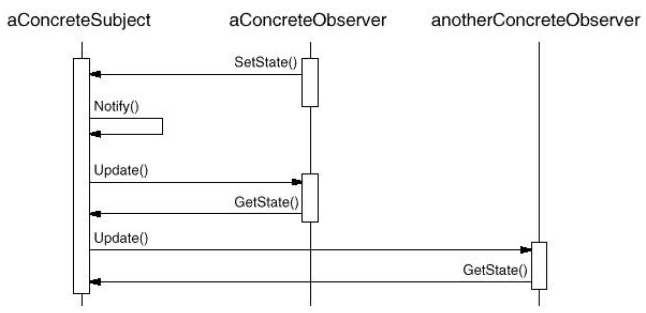
    - Mementos are passive. Only the originator that created a memento will assign or retrieve its state.
- **Consequences**
    - *Preserving encapsulating boundaries*
    - *It simplifies Originator*
    - *Using mementos might be expensive*
    - *Defining narrow and wide interfaces*
    - *Hidden costs in caring for mementos*
- **Implementation**
    - *Language support* - Only narrow interface to be exposed publicly other fields to be private (originator could be a friend in C++)
    - *Storing incremental changes* - When mementos get created & passed back to their originator in a predictable sequence, then it can save just the *incremental change*.
- **Related Patterns**
    - Command: Commands can use mementos to maintain state for undoable operations
    - Iterator: Mementos can be used for iteration

Observer (Object Behavioral)

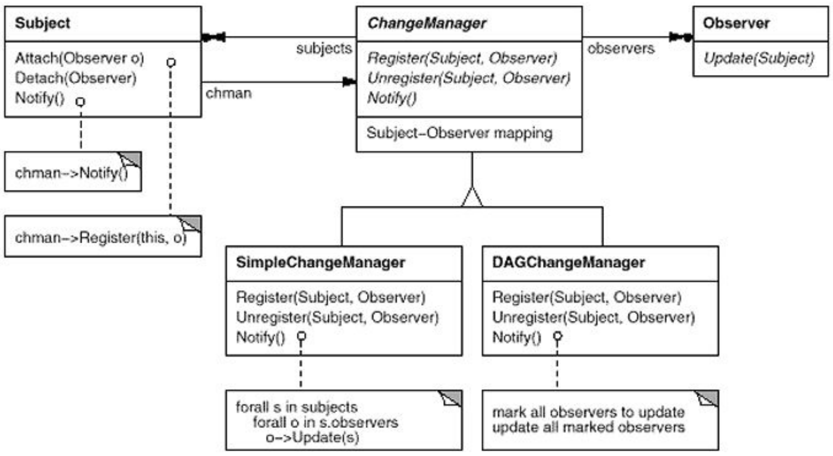
- **Intent**
    - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Also Known As**

- Dependents, Publish-Subscribe
- **Motivation**
  - **Subject** and **observer** relationship without tight coupling
  - Changing values changes multiple graphs - bar, chart
- **Applicability**
  - When an abstraction has 2 aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
  - When a change to 1 object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.
- **Structure**
  - 

- **Participants**
  - **Subject**
    - knows its observers. Any number of Observer object may observe it
    - provides an interface for attaching and detaching Observer objects
  - **Observer**
    - defines an updating interface for objects that should be notified of changes in a subject
  - **ConcreteSubject**
    - stores state of interest to ConcreteObserver objects
    - sends a notification to its observer when its state changes
  - **ConcreteObserver**
    - maintains a reference to a ConcreteSubject object
    - stores state that should stay consistent with the subject's
    - implements the Observer updating interface to keep its state consistent with the subject's
- **Collaborations**
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own
  - After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject

    - 

    - “Notify” is not always called by the subject. It can be called by an observer or by another kind of object entirely
- **Consequences**
    - *Abstract coupling between Subject and Observer* - Subject & observer aren’t tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system’s layering intact.
    - *Support for broadcast communication* - Subject just broadcasts and doesn’t care who are subscribed to it, observers can be added & removed at any time
    - *Unexpected updates* - Because observers have no knowledge of each other’s presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren’t well-defined or maintained usually lead to spurious updates, which can be hard to track down.
- **Implementation**
    - *Mapping subjects to their observers* - maintaining a global map, map in individual subjects will be expensive
    - *Observing more than one subject* - If observer depends on more than one subject then subject may pass itself as a parameter in the Update operation
    - *Who triggers the update?*
        - Either several state setting operations can call notify. Advantage is that clients don’t have to remember to call Notify on the subject. The disadvantage is that several consecutive operations will cause several updates
        - Or Clients need to call Notify, this will avoid needless updates but this makes it error prone because clients might skip this
    - *Dangling references to deleted subjects*
    - *Making sure Subject state is self-consistent before notification* - In case of overridable methods, notify should only be called after the subject is consistent. Template methods can be used to avoid this.
    - *Avoiding observer-specific update protocols: the push and pull models* - Push model requires subject to have some knowledge about observer’s needs which makes subject less reusable. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

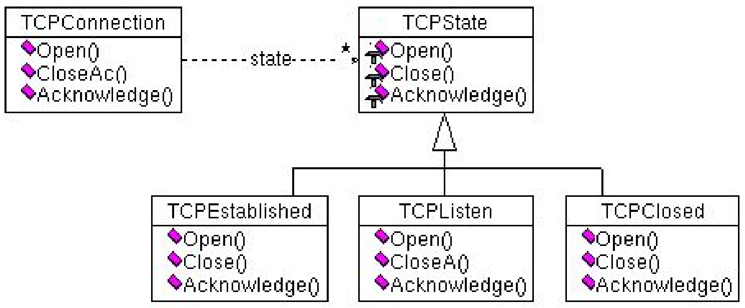
- *Specifying modifications of interest explicitly* - All subject registration interface to allow registering observers only for specific events. This can be done using **aspects** for Subject objects.
- *Encapsulating complex update semantics* - When the dependency relations between subjects and observers is particularly complex, an object that maintains these relationships might be required. We call such an object a **ChangeManager** (mediator pattern). Its purpose is to minimize the work required to make observers reflect a change in their subject. For example if an operation involves changes to several interdependent subjects, you might have to ensure that their observers are notified only after *all* the subjects have been modified to avoid notifying observers more than once.
  Change manager has 3 responsibilities:
  - it maps a subject to its observer
  - if defines a particular update strategy
  - it updates all dependent observer at the request of a subject
  - 

- *Combining the Subject and Observer classes*

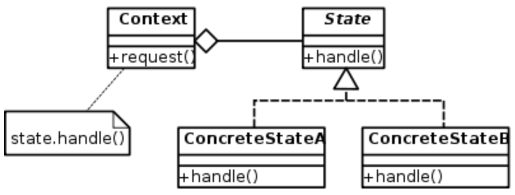
- **Related Patterns**
  - Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers
  - Singleton: The ChangeManager may use the Singleton pattern to make it unique and globally accessible

State (Object Behavioral)

- **Intent**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Also Known As**
  - Objects for States
- **Motivation**
  - Class can exhibit different behavior in each state by delegating state specific requests to a state object



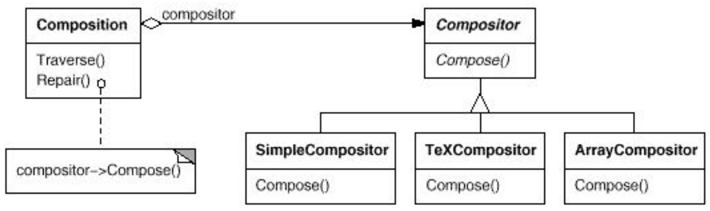
- **Applicability**
  - An object's behavior depends on its state, and it must change it at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several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



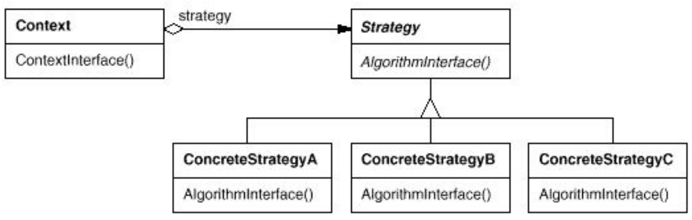
- **Participants**
  - **Context** (TCPConnection)
    - defines the interface of interest to clients
    - maintains an instance of a ConcreteState subclass that defines the current state
  - **State** (TCPState)
    - defines an interface for encapsulating the behavior associated with a particular state of the Content
  - **ConcreateState subclasses** (TCPEstablished, TCPListen, TCPClosed)
    - each subclass implements a behavior associated with a state of the Context
- **Collaborations**
  - Context delegates state-specific requests to the current ConcreteState object
  - A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
  - Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
  - Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.
- **Consequences**
  - *It localizes state-specific behavior and partitions behavior for different states*

  - *It makes state transitions explicit* - Can also protect the Context from inconsistent internal states, because state transitions are atomic from the Context's perspective - they happen after rebinding *one* variable
  - *State objects can be shared*
- **Implementation**
  - *Who defines the state transitions?* - It is generally more flexible to let the State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.
    Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses. A disadvantage of decentralization is that one State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.
  - *A table-based alternative*
  - *Creating and destroying State objects*
  - *Using dynamic inheritance*
- **Related Patterns**
  - The Flyweight pattern explains when and how State objects can be shared. State objects are often Singleton.

Strategy

- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Also Known As**
  - Policy
- **Motivation**
  - 

- **Applicability**
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - you need different variants of an algorithm.
  - an algorithm uses data that clients shouldn't know about. Use this to avoid exposing complex, algorithm-specific data structures.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
- **Structure**

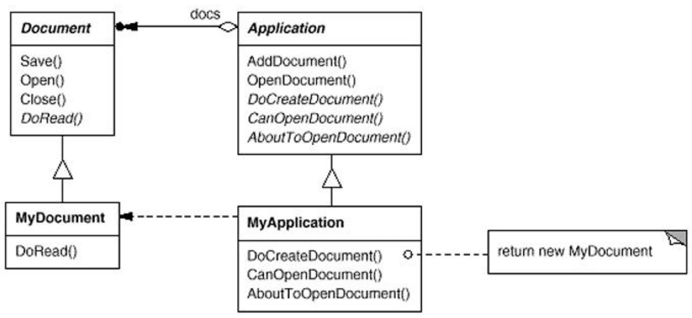
- 

- **Participants**
  - **Strategy** (Compositor)
    - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
  - **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
    - implements the algorithm using the Strategy interface
  - **Context** (Composition)
    - is configured with a ConcreteStrategy object
    - maintains a reference to a Strategy object
    - may define an interface that lets Strategy access its data
- **Collaborations**
  - Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass, itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
  - A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.
- **Consequences**
  - *Families of related algorithms* - Hierarchies of Strategy classes define a family of algorithms or behaviors for context to reuse. Inheritance can help factor out common functionality of the algorithms.
  - *An alternative to subclassing* - You can subclass a Context class directly to give different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context’s, making Context harder to understand, maintain and extend. And you can’t vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
  - *Strategies eliminate conditional statements*
  - *A choice of implementations*
  - *Clients must be aware of different Strategies*
  - *Communication overhead between Strategy and Context* - All concrete strategies share a common interface, so if the strategy is simple and doesn’t need much information from the context it still follows the interface.
  - *Increased number of objects* - Strategies increase the no. of objects in an application. Sometimes you can reduce this overhead by implementing

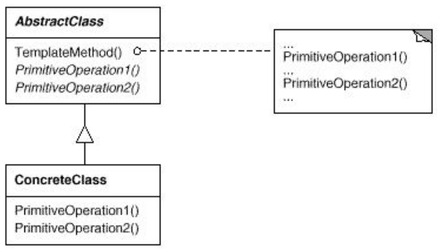
strategies as stateless objects that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy object.

- **Implementation**
  - *Defining the Strategy and Context interfaces* - The Strategy & Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
    Once approach is to have Context pass data in parameters to Strategy operations (take the data to the strategy). This keeps Strategy & Context decoupled. On the other hand, Context might pass data the Strategy doesn't need.
    Another technique has a context pass *itself* as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all. Either way, the strategy can request exactly what it needs. But not Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.
  - *Strategies as template parameters*
  - *Making Strategy objects optional*
- **Related Patterns**
  - Flyweight: Strategy objects often make good flyweights

Template Method (Class Behavioral)

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

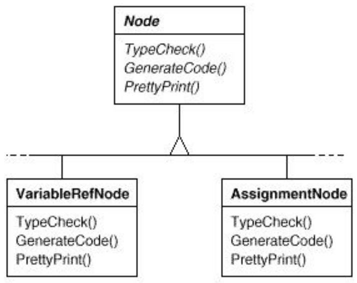
- **Applicability**
  - to implement the invariant parts of an algorithms once and leave it up to subclasses to implement the behavior that can vary
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
  - to control subclasses extensions.
- **Structure**

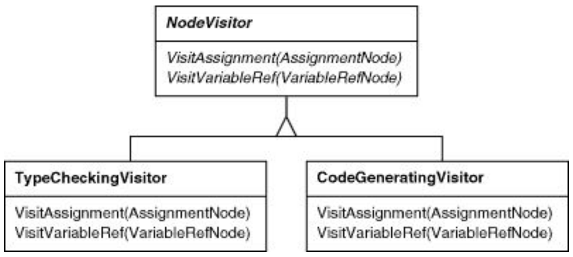
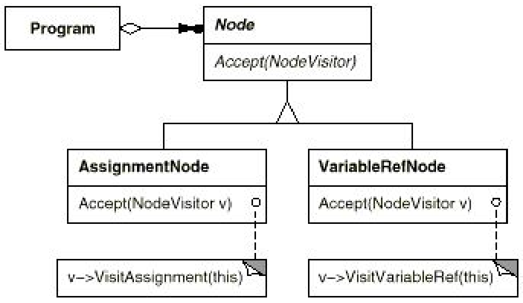
- 

- **Participants**
  - **AbstractClass** (Application)
    - defines abstract **primitive operations** that concrete subclasses define to implement steps of algorithms.
    - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
  - **ConcreteClass** (MyApplication)
    - implements the primitive operations to carry out subclass-specific steps of the algorithm.
- **Collaboration**
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm
- **Consequences**
  - Fundamental technique for code reuse
  - Template methods lead to an inverted control structure that's sometimes referred to as "the Hollywood principle" *i.e.* "Don't call us, we'll call you"
  - It calls the following kind of operations:
    - concrete operations (either on the ConcreteClass or on client classes)
    - concrete AbstractClass operations (i.e., operations that are generally useful to subclasses)
    - primitive operations (i.e. abstract operations)
    - factory methods
    - **hook operations**, which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.
- **Implementation**
  - *Using C++ access control*
  - *Minimizing primitive operations*
  - *Naming conventions*
- **Related Patterns**
  - Factory Method are often called by template methods
  - Strategy: Template methods use inheritances to vary part of an algorithm. Strategies use delegation to vary the entire algorithm

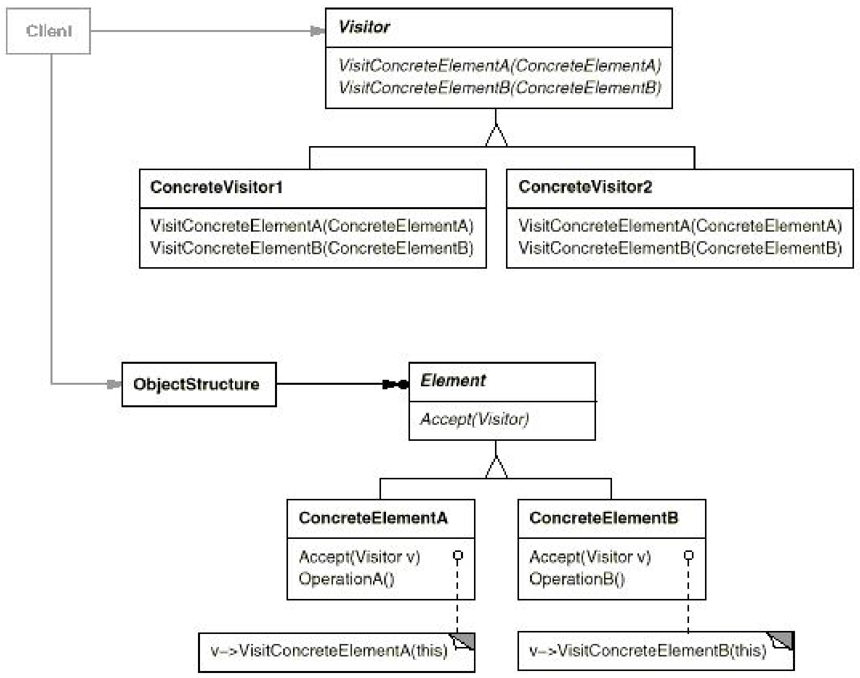
Visitor (Object Behavioral)

- **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.
- **Motivation**
    - 

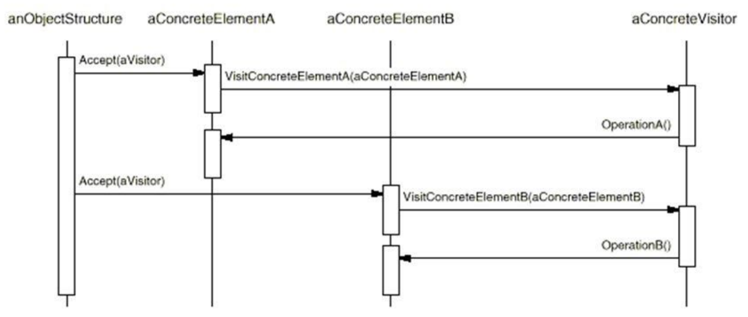
    - 
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
    - 

- **Applicability**
    - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
    - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid “polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
    - the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it’s probably better to define the operations in those classes.
- **Structure**


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

- **Participants**
  - **Visitor** (NodeVisitor)
    - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
  - **ConcreteVisitor** (TypeCheckingVisitor)
    - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
  - **Element** (Node)
    - defines an Accept operation that takes a visitor as an argument.
  - **ConcreteElement** (AssignmentNode, VariableRefNode)
    - implements an Accept operation that takes a visitor as an argument.
  - **ObjectStructure** (Program)
    - can enumerate its elements.
    - may provide a high-level interface to allow the visitor to visit its elements.
    - may either be a composite (see Composite) or a collection such as a list or a set.
- **Collaborations**
  - A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
  - When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

Visitor: collaborations



5

-
- **Consequences**
    - *Visitor marks adding new operations easy*
    - *A visitor gather related operations and separate unrelated ones*
    - *Adding new ConcreteElement classes is hard* - Sometimes a default implementation can be provided in Visitor that can be inherited by most of the ConcreteVisitors
    - *Visiting across class hierarchies* - Visitor doesn't restrict that objects have a parent class whereas iterator requires that, so iterator can be used sometimes
    - *Accumulating state* - Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would be passed as extra arguments to the operations that perform the traversal, or they might appear as global variables.
    - *Breaking encapsulation* - Visitors' require ConcreteElement interface to do their job thus required public operations to access element's internal state
- **Implementation**
    - Each operation can be named same (using overloading) in ConcreteVisitor as arguments are different
    - *Double dispatch* - (see CLOS) The operation that gets executed depends on the type of the Visitor and the type of Elements it visits
    - *Who is responsible for traversing the object structure?* - 3 places: the object structure, the visitor, or separate iterator object. Often the object structure is responsible like: for collection, for composite.
      In case of an iterator, an internal iterator will not cause double-dispatching - it will call an operation on the *visitor* with an *element* as an argument as opposed to calling an operation on the *element* with the *visitor* as an argument.
      Visitor traversal will require you to copy the same code in each Concrete Visitor, mostly this is done when the traversal is complex and depends on the results of the operation

- **Related Patterns**
    - Composite: Visitors can be used to apply an operation over an object structure defined by the Composite pattern.
    - Interpreter: Visitor may be applied to do the interpretation.

Discussion of Behavioral Patterns

6. Conclusion

6.1 What to Expect from Design Patterns

6.2 A Brief History

6.3 The Pattern Community

6.4 An Invitation

6.5 A Parting Thought