Design Patterns

Y. Narahari

Computer Science and Automation

Indian Institute of Science

Bangalore - 560 012

DESIGN PATTERNS

- Simple and elegant solutions to specific, commonly occurring problems in software design
- Each design pattern addresses an important and recurring design problem in object oriented design
- DPs make it easier to reuse successful designs and architectures of experienced and professional designers
- DPs improve documentation and maintenance of software
- *Each pattern describes a problem that occurs over and over again and sketches a solution for the problem in a way that the solution can be reused in numerous contexts*

What Constitutes a DP?

- Pattern name and classification
- Intent of the pattern
- Motivation for the pattern
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known uses and related patterns

Catalog of Patterns

- Gamma, Helm, Johnson, and Vlissides describe 23 patterns in their book - *Design Patterns: Elements of Reusable Object-Oriented Software*
- Patterns keep evolving and are updated: `http://st-www.cs.uiuc.edu/users/patterns`
- Three types of Patterns:
  1. **Creational**: concerned with creation of objects
  2. **Structural**: concerned with composition of classes or objects
  3. **Behavioral**: characterize the ways in which classes and objects interact and distribute responsibility

Creational Patterns

- **Abstract Factory**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Builder**: Separate the construction of a complex object from its representation so that the same construction process can create different representations
- **Factory Method**: Define an interface to create an object, but let subclasses decide which class to instantiate (Instantiation is deferred to subclasses)
- **Prototype**: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- **Singleton**: Ensure a class has only one instance and provide a global point of access to it

Structural Patterns

- **Adapter**: Convert the interface of a class into another interface that clients expect
- **Bridge**: Decouple an abstraction from its implementation so that the two can vary independently
- **Composite**: Compose objects into tree structures to represent part- whole hierarchies. (Let clients treat individual objects and compositions of objects in a uniform way)
- **Decorator**: Attach additional responsibilities to an object dynamically to provide a flexible alternative to subclassing
- **Facade**: Provide a unified interface to a set of interfaces in a subsystem, enabling easier use of the subsystem
- **Flyweight**: Use sharing to support large numbers of fine-grained objects efficiently
- **Proxy**: Provide a surrogate or placeholder for another object to control access for it

Behavioral Patterns - 1

- **Chain of Responsibility**: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request (chain the receiving objects and pass the request along until an object handles it)
- **Command**: Encapsulate a request as an object, thereby enabling to parameterize clients with different requests, queue or log requests, and support special operations
- **Interpreter**: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
- **Iterator**: Provide a way to access the elements of an aggregate object sequentially without exposing its representation

Behavioral Patterns - 2

- **Mediator**: Define an object that encapsulates how a set of objects interact, to promote loose coupling by keeping objects from referring to each other explicitly and enabling to vary their interaction independently
- **Memento**: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
- **Observer**: Define to one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **State**: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class

Behavioral Patterns - 3

- **Mediator**: Define an object that encapsulates how
- **Strategy**: Define a family of algorithms, encapsulate each one, and make them interchangeable, enabling the algorithm to vary independently from clients that use it
- **Template Method**: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. This will let subclasses redefine certain steps of an algorithm without changing the structure of the algorithm
- **Visitor**: Represent an operation to be performed on the elements of an object structure, enabling to define a new operation without changing the classes of the elements on which it operates.

Why Design Patterns? - 1

- Enable best practices in design to be used commonly
  - *Program to interface, not implementation*
    - `Abstract Factory, Builder, Factory Method, Prototype, Singleton`
  - *Prefer interface inheritance over implementation inheritance*
    - `Chain of Resp, Composite, Command, Observer, State, Strategy`
  - *Favour object composition and delegation over class inheritance*
    - `State, Strategy, Visitor, Mediator, Chain of Resp, Bridge`
  - *Use small inheritance hierarchies, avoid complex compositions, use a judicious mix*
    - `Composite, Strategy, Abstract Factory`

Why Design Patterns? - 2

- Find appropriate objects during design
  - `Strategy, Composite, State`
- Determine object granularity
  - `Facade, Flyweight, Abstract Factory, Builder, Visitor, Command`
- Specify object interfaces
  - `Memento, Decorator, Proxy, Visitor`
- Specify object implementations
  - `Composite, Chain of Responsibility, Command, Observer, State, Strategy`

Why Design Patterns? - 3

- Enable effective use of reuse mechanisms
  - Inheritance
  - Composition
  - Delegation
  - Parameterized types
- Relating run-time and compile-time structures: DPs enable a better mapping between the two
  - `Composite`, `Decorator`, `Chain of Responsibility`, `Observer`
- Enable design for change: DPs ensure that a system can change in specific and structured ways only
- Lead to high quality designs of application programs, toolkits, and frameworks

Design for Change - 1

- Create objects indirectly and avoid creating and object by explicitly specifying a class
  - `Abstract Factory, Factory Method, Prototype`
- Design so as to limit platform dependencies
  - `Abstract Factory, Bridge`
- Reduce dependence on object representation or implementation by hiding the internals from clients
  - `Abstract Factory, Bridge, Memento, Proxy`
- Reduce algorithmic dependencies by isolating algorithms that are likely to change
  - `Builder, Iterator, Strategy, Template Method, Visitor`

Design for Change - 2

- Promote loosely coupled classes
  - `Abstract Factory, Bridge, Chain of Responsibility, Facade, Mediator, Observer`
- Avoid subclassing as a means to extend functionality; use composition instead
  - `Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy`
- Enable convenient alteration of classes
  - `Adapter, Decorator, Visitor`

Toolkits and Frameworks

- Lead to high quality application programs by promoting
  - internal reuse, maintainability, extensibility
- Lead to better Toolkits
  - A toolkit is a set of related and reusable classes designed to provide useful, general purpose functionality
  - Code reuse and flexibility are important
- Lead to better Frameworks
  - A set of cooperating classes that make up a reusable design for a specific class of software
  - Comprises abstract classes from which application-specific subclasses can be created to build a particular customized application
  - Design reuse and flexibility are important

Creational Patterns

- Abstract the instantiation process and help make a system independent of how its objects are created, composed, and represented.
- Class Creational Patterns
  - Factory Method
  - use inheritance to vary the class that is instantiated
- Object creational Patterns
  - Abstract Factory, Builder, Prototype, Singleton
  - use delegation to instantiate classes
- Creational patterns gain importance as systems evolve to depend more on object composition rather than class inheritance

Creational Patterns

- Recurring Themes:
  - encapsulate knowledge about which concrete classes the system uses
  - hide how instances of these classes are created and put together
- Provide flexibility in *what* gets created, *who* creates it, *how* it gets created, and *when*.
- Competitive: Abstract Factory and Prototype
- Complementary: Builder and Prototype; Builder and Abstract Factory; Abstract Factory and factory Method; Prototype and Singleton; etc.

Abstract Factory Pattern

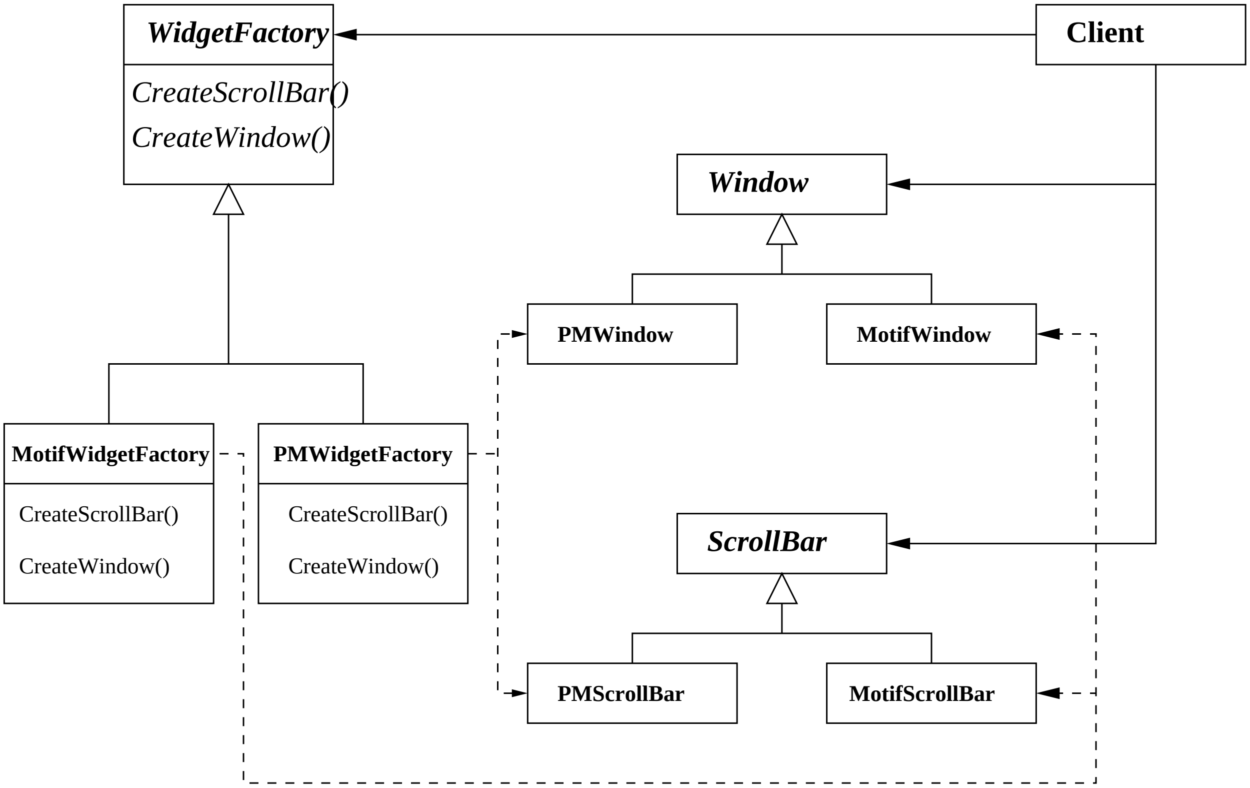
- **Intent**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Motivation**:
  - Provide portability across look and feel standards (InterViews)
  - Provide portability across different window systems (ET++)
- **Applicability**: Use this pattern 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 there is a need to enforce this constraint

  - it is required to provide a class library of products where only interfaces are to be revealed

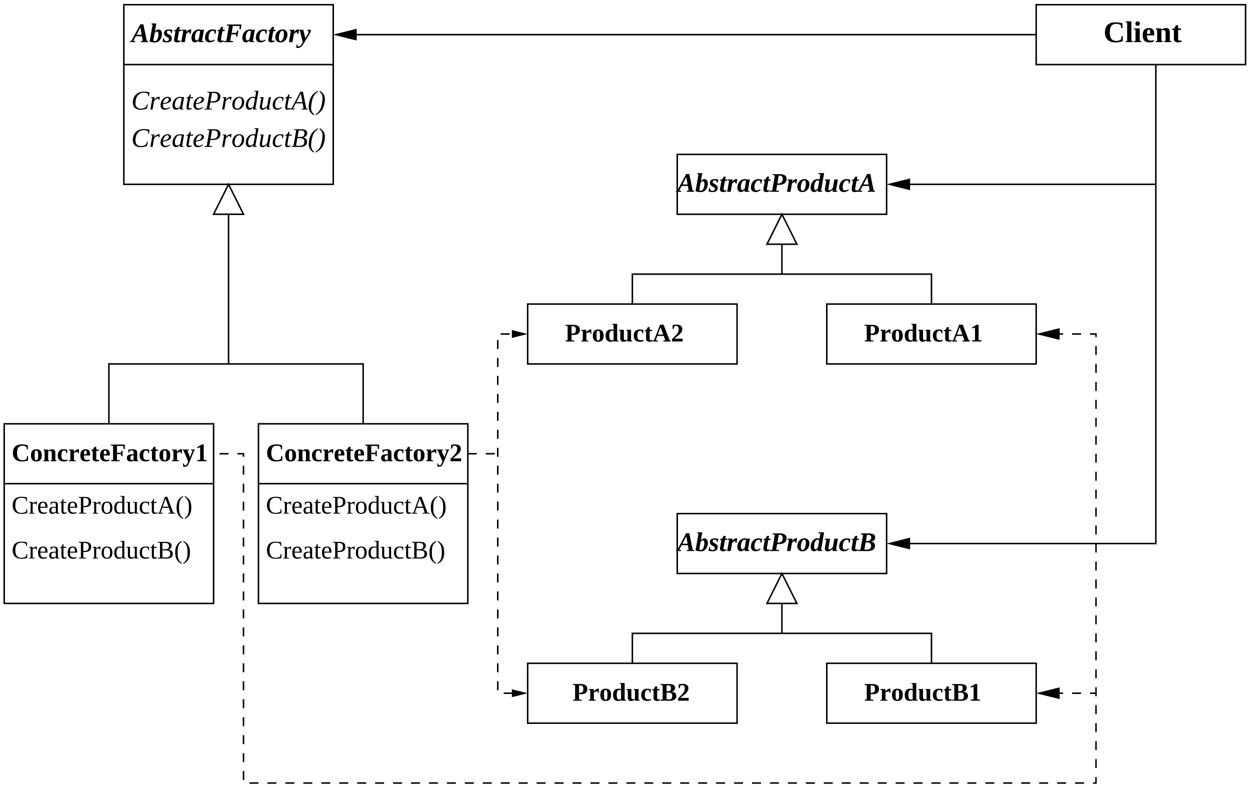
- **Participants**:
  1. **AbstractFactory**: declares an interface for operations that create abstract product objects
  2. **ConcreteFactory**: implements the operations to create concrete product objects
  3. **AbstractProduct**: declares an interface for a type of product object
  4. **ConcreteProduct**: defines a product object to be created by the corresponding factory; implements the interface of the AbstractProduct class
  5. **Client**: uses only interfaces declared by AbstractFactory and AbstractProduct classes

- **Collaborations**:
  1. A single instance of ConcreteFactory class is created at run time. This creates product objects having a particular implementation
  2. AbstractFactory defers creation of product objects to its ConcreteFactory subclass
- **Consequences**:
  + isolates concrete classes
  + makes exchanging product families easy
  + promotes consistency among products
  – makes it cumbersome to support new kinds of products at run time.
- **Implementation**:
  1. Each ConcreteFactory is best implemented using a Singleton pattern
  2. ConcreteProduct subclasses create the products; the Factory Method pattern or the Prototype pattern can be used for this purpose
  3. Defining extensible factories is not that easy

Motivational Example for Abstract Factory



Structure of Abstract Factory Pattern



Builder Pattern

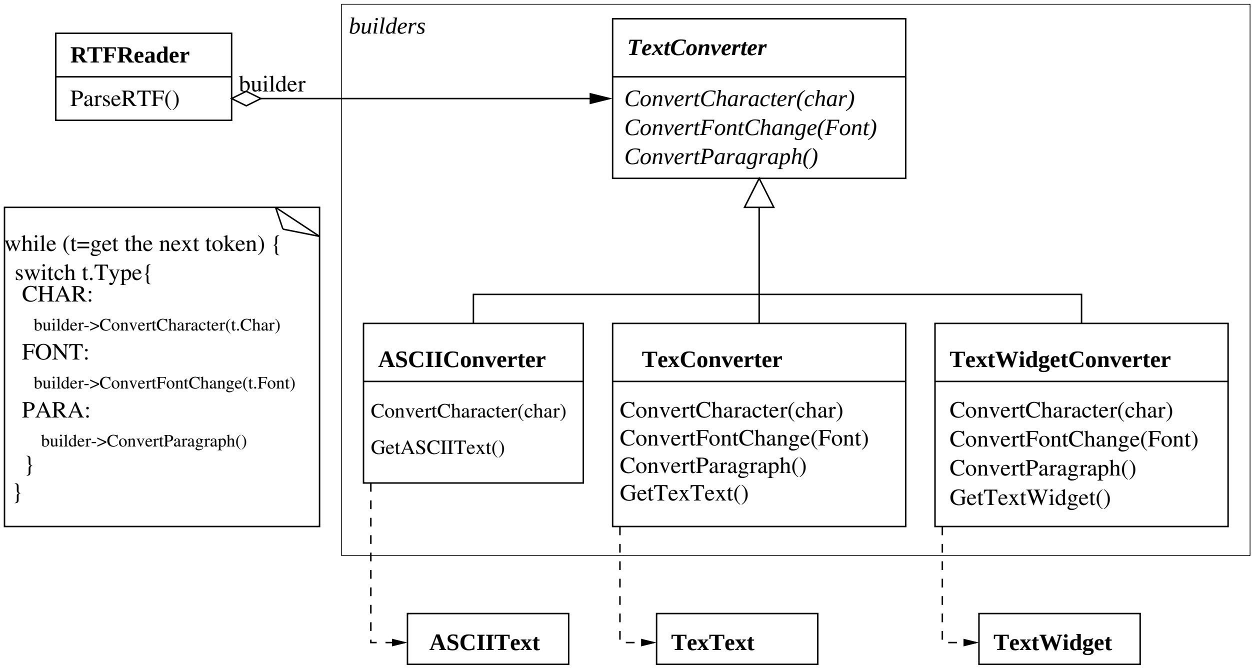
- **Intent**: Separate the construction of a complex object from its representation so that the same construction process can create different representations
- **Applicability**: Use this pattern when
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - the construction process must allow different representations for the object that is constructed

- **Participants**:
  1. **Builder**:specifies an abstract interface for creating parts of a Product object
  2. **ConcreteBuilder**:
     - 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
  3. **Director**: constructs an object using the Builder interface
  4. **Product**:
     - represents the complex object under construction.
     - 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 Director and adds parts to the Product
  - The client retrieves the product from the Builder
- **Consequences**:
  - + enables to vary a product's internal representation
  - + isolates code for construction and representation
  - + gives finer control over the construction process since the pattern constructs a product step by step under the Director's control.

- **Implementation**
  - The Builder class interface must be general enough to allow construction of products for all kinds of concrete builders
  - The products produced by the concrete builders can differ so greatly that there is little justification for having an abstract class for products
- **Related Patterns**
  - **Composite**: A composite is what a Builder often builds
  - **Abstract Factory**: Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes families of product objects. Builder returns the product as a final step whereas the Abstract Factory returns the product immediately

Motivational Example for Builder



builders

RTFReader

ParseRTF()

builder

TextConverter

ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()

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

ASCIIConverter

ConvertCharacter(char)
GetASCIIText()

TexConverter

ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTexText()

TextWidgetConverter

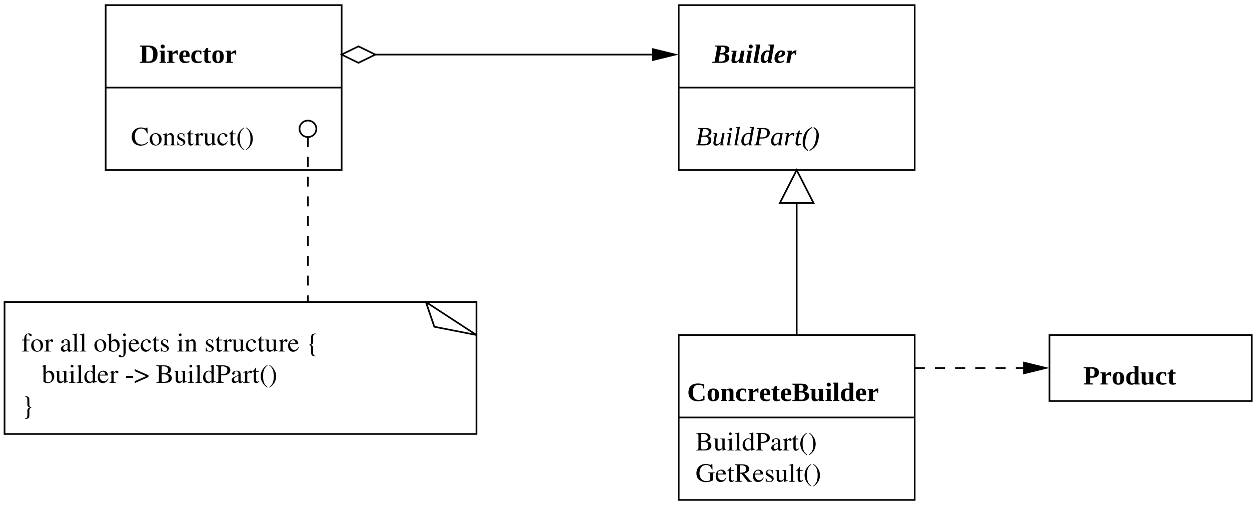
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()

ASCIIText

TexText

TextWidget

Builder Pattern



Director

Construct()

Builder

BuildPart()

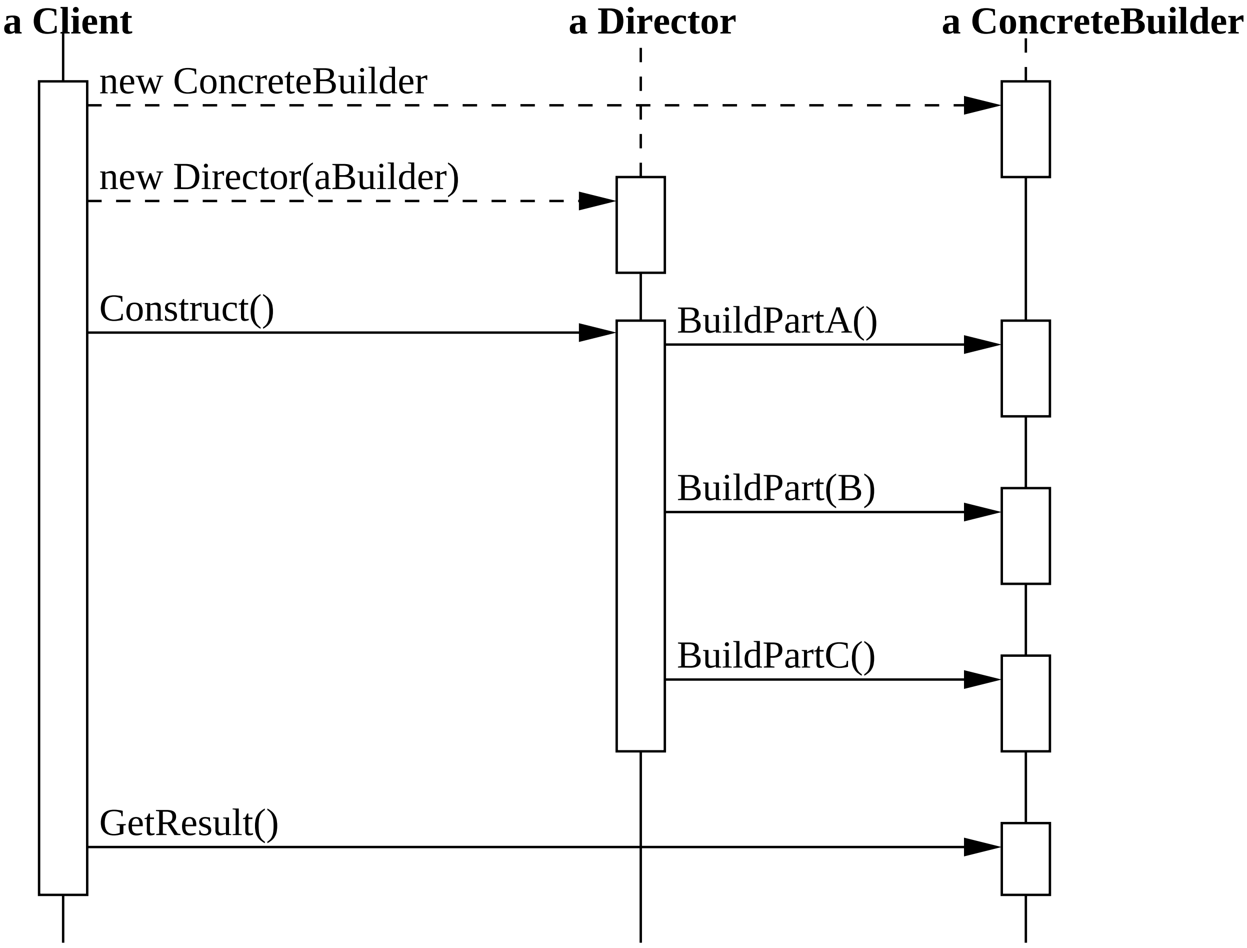
```
for all objects in structure {
  builder -> BuildPart()
}
```

ConcreteBuilder

BuildPart()
GetResult()

Product

Interactions in Builder Pattern



Factory Method (Virtual Constructor)

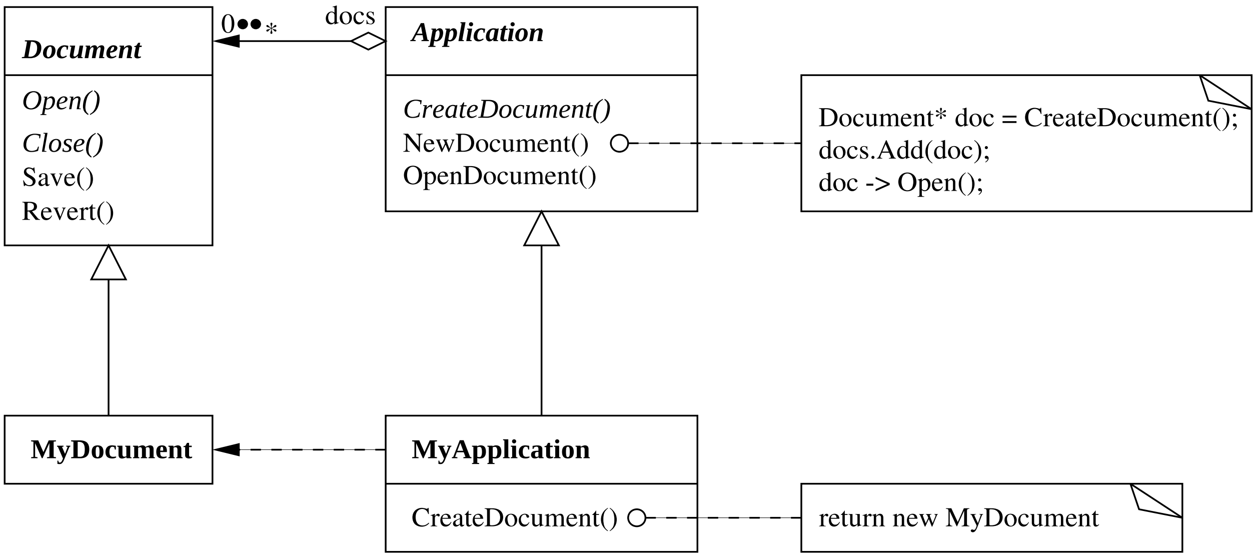
- **Intent**: Define an interface for creating an object but let subclasses decide which class to instantiate (defer class instantiation to subclasses)
- **Motivation**:
  - Creating objects through Frameworks
  - Classes to be instantiated may often be application specific
- **Applicability**: Use this pattern when
  - a class cannot 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 it is desirable to localize the knowledge of which helper subclass is the delegate

- **Participants**:
  1. **Product**: defines the interface of objects the factory method creates
  2. **ConcreteProduct**: implements the product interface
  3. **Creator**:
     - declares the factory method, which returns an object of type Product
     - may define a default implementation of the factory method to return a default ConcreteProduct object
     - may call the factory method to create a Product object
  4. **ConcreteCreator**: 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

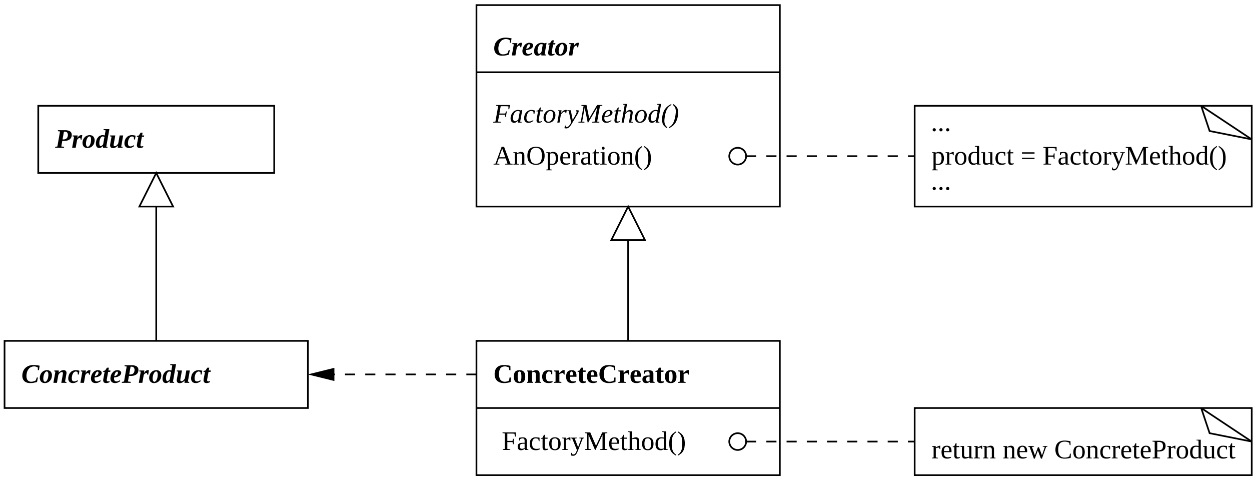
- **Consequences**:
  - + eliminate the need to bind application-specific classes into our code
  - + gives subclasses a hook for providing an extended version of an object
  - + define connection between parallel class hierarchies
- **Implementation**:
  - There could be two main variations of Factory Methods:
    - Creator is abstract and does not provide an implementation for the factory method it declares
    - Creator is concrete and provides a default implementation for the factory method
  - Parameterized Factory Method: The pattern can be enabled to create multiple kinds of products

- **Related Patterns**:
  - **Abstract Factory**: is often implemented with factory methods
  - **Template Method**: Factory methods are often called within Template Methods
  - **Prototype**: Prototypes don't require subclassing Creator (required by Factory Method), but often require an Initialize operation on the Product class (not required by Factory Method).

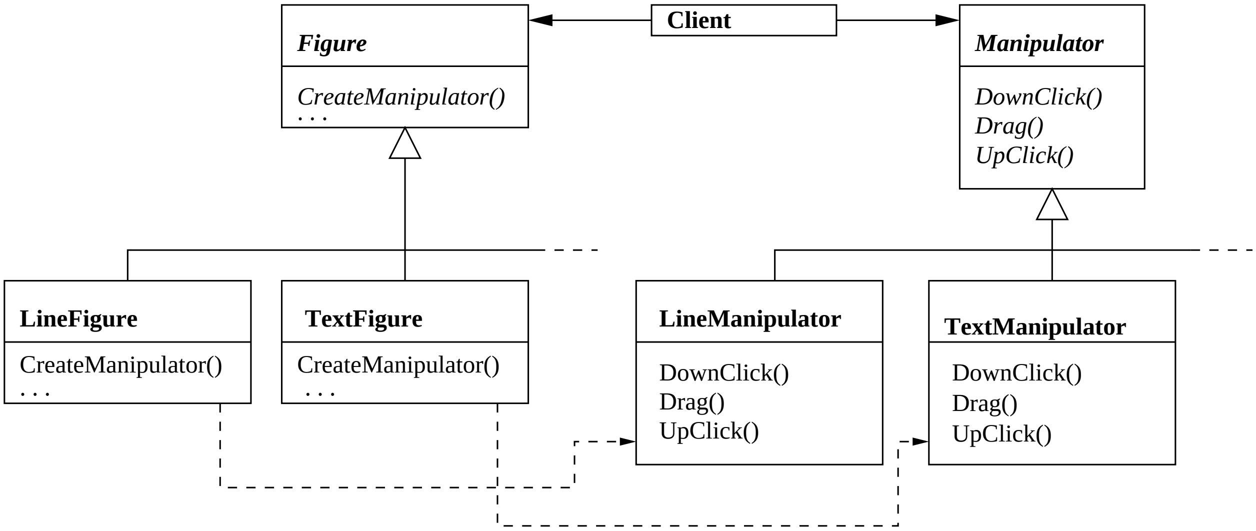
Motivational Example: Factory Method



Factory Method Pattern



Factory Method Pattern



Prototype Pattern

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

- **Participants**:
  1. **Prototype**: declares an interface for cloning itself
  2. **ConcretePrototype**: implements an operation for cloning itself
  3. **Client**: creates a new object by asking a prototype to clone itself

- **Collaboration**: A client asks a prototype to clone itself

- **Consequences**:
  + hides concrete product classes from the client, thereby reducing the number of names clients know about
  + enables a client to work with application-specific classes without modification
  + clients can install and remove prototypes at run times

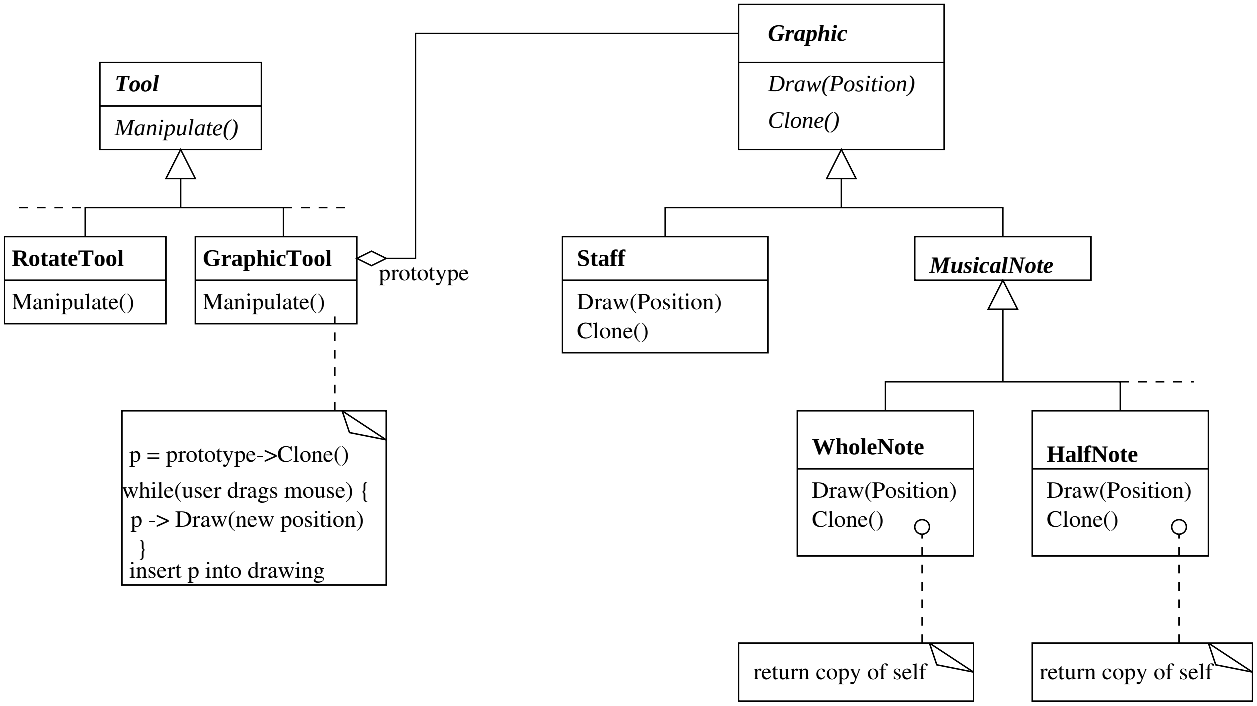
+ reduces greatly the number of classes a system needs

+ obviates the need for a separate Creator class hierarchy as in factory methods

– each subclass of Prototype must implement the clone operation

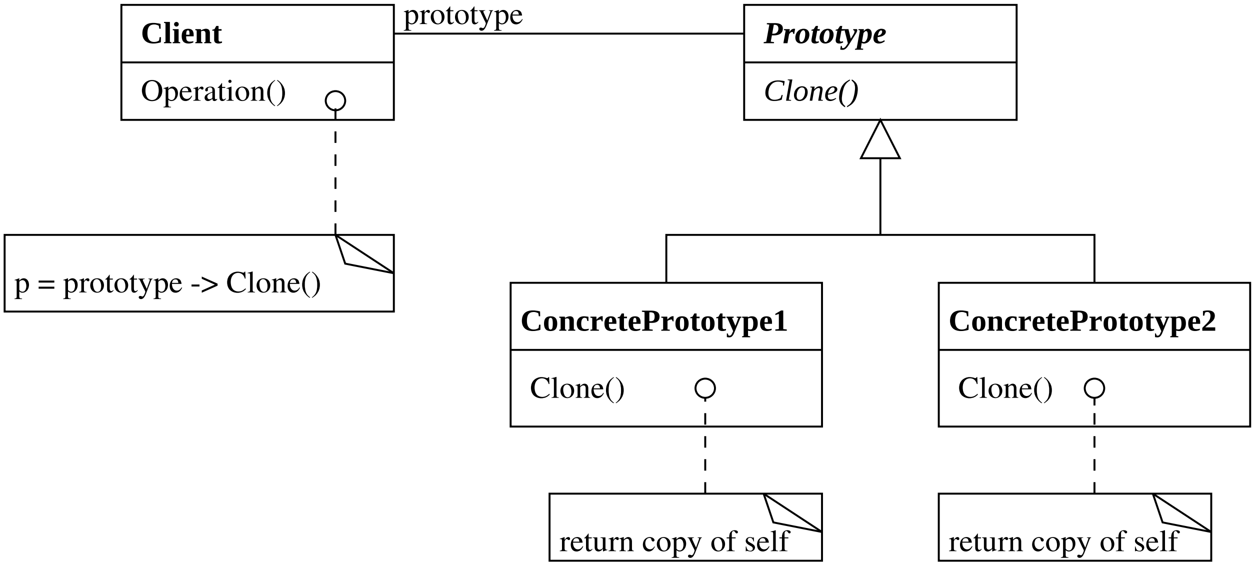
- **Related Patterns**:
  - **Abstract Factory**: A prototype is often a competitor for Abstract Factory; however the two patterns can often be used together
  - **Composite and Decorator**: Designs that make intensive use of Composite and Decorator can benefit from Prototype

Motivational Example for Prototype



```
p = prototype->Clone()
while(user drags mouse) {
 p -> Draw(new position)
  }
insert p into drawing
```

Prototype Pattern



Singleton

- **Intent**: Ensure that a class has only one instance and provide a global point of access to it.
- **Applicability**: Use this pattern when
  - there must be exactly one instance of a class, and it must be accessible from a well-known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
- **Key Idea**: Make the class itself responsible for keeping track of its sole instance. The class can intercept requests to create new objects and thus guarantee that no other instance can be created; it can also provide a way to access the instance.

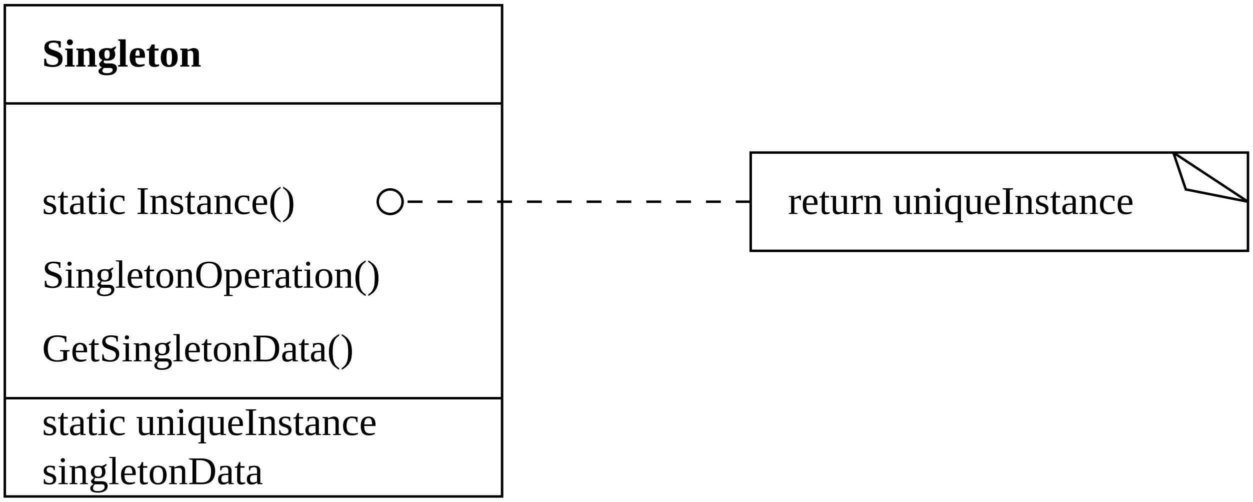
- **Participants**:
  1. **Singleton**:
     - defines an Instance operation that lets clients access to its unique instance
     - may be responsible for creating its own unique instance.
- **Collaboration**: Clients access a Singleton instance solely through Singleton's instance operation
- **Consequences**:

  + improvement over global variables: controlled access

  + permits refinement of operations and representation

  + permits a variable number of instances

Singleton Pattern



Singleton

static Instance()

SingletonOperation()

GetSingletonData()

static uniqueInstance

singletonData

return uniqueInstance

Summary of Creational Patterns

- Factory Method uses subclassing: subclass the class that creates the objects
  - can require a new subclass just to change the class of the product; such changes can cascade
- Abstract Factory, Prototype, and Builder use object composition
  - all three patterns involve creating a new factory object whose responsibility is to create product objects
  - Abstract Factory has the factory object producing objects of several classes
  - Builder has the factory object building a complex product
  - Prototype has the factory object building a product by copying a prototype object

- Factory Method makes a design more customizable and only a little more complex since it only requires a new operation
- Other patterns are even more flexible ; however, require new classes and are more complex

Structural Patterns

- Concerned with classes and objects can be assembled to form larger structures
- **Class Structural Patterns** use inheritance to assemble interfaces or implementations
  - Class version of `Adapter`
  - Multiple inheritance
- **Object Structural Patterns** describe ways to compose objects to realize new functionality
  - Composite, Decorator, Bridge, etc.

Composite Pattern

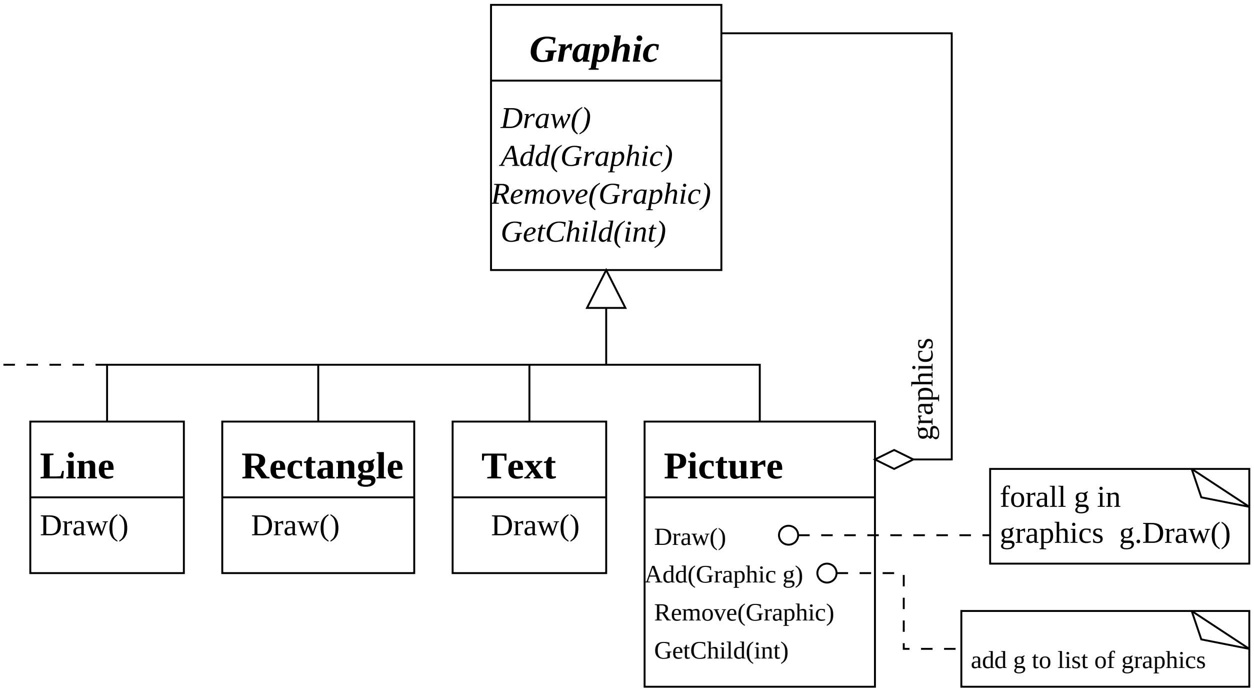
- **Intent**: Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly
- **Motivation**:
  - Graphics applications which use a variety of simple to complex diagrams made of smaller components
  - Financial applications where a portfolio aggregates individual assets
  - Parse trees in compilers which are composites of subclasses
- **Applicability**: Use this pattern when
  - there is a need to represent part-whole hierarchies
  - there is a need for clients to be able to ignore the difference between compositions of objects and individual objects

- **Participants**:
  1. **Component**:
     - declares the interface for objects in the composition
     - implements default behavior for the interface, common to all classes, as appropriate
     - declares an interface for accessing and managing its child components
  2. **Leaf**: represents leaf objects in the composition and defines behavior for primitive objects in the composition
  3. **Composite**:
     - defines behavior for components having children
     - stores child components
     - implements child-related operations in the Component interface
  4. **Client** manipulates objects in the composition through the Component interface

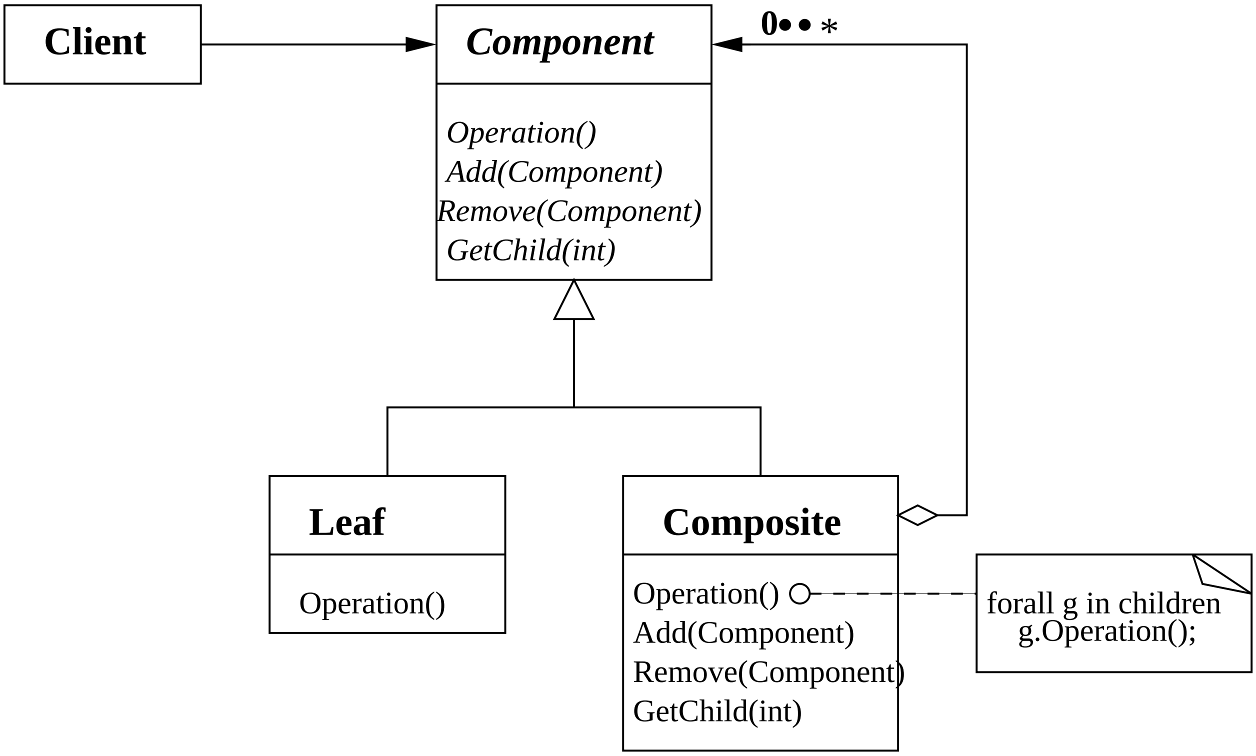
- **Collaborations**:
  - Clients use the Component interface to interact with objects in the composite structure
  - If the recipient is a Leaf, the request is handled directly
  - If the recipient is a Composite, it usually forwards requests to child components, with possible preprocessing and post processing.
- **Consequences**:
  - + defines class hierarchies consisting of primitive objects and composite objects
  - + makes the client simple
  - + makes it easier to add new types of components
  - – can make the design overly general - makes it harder to restrict the components of a composite

- **Implementation**
  - Explicit parent references can simplify the traversal and management of a composite structure
  - To make clients unaware of the specific leaf or composite classes they are using, the Component class should define as many common operations as possible
- **Related Patterns**
  - **Decorator**: Often decorators and composites are used together
  - **Iterator**:can be used to traverse composites
  - **Visitor**:localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes

Motivational Example for Composite



Structure of the Composite Pattern



Decorator Pattern

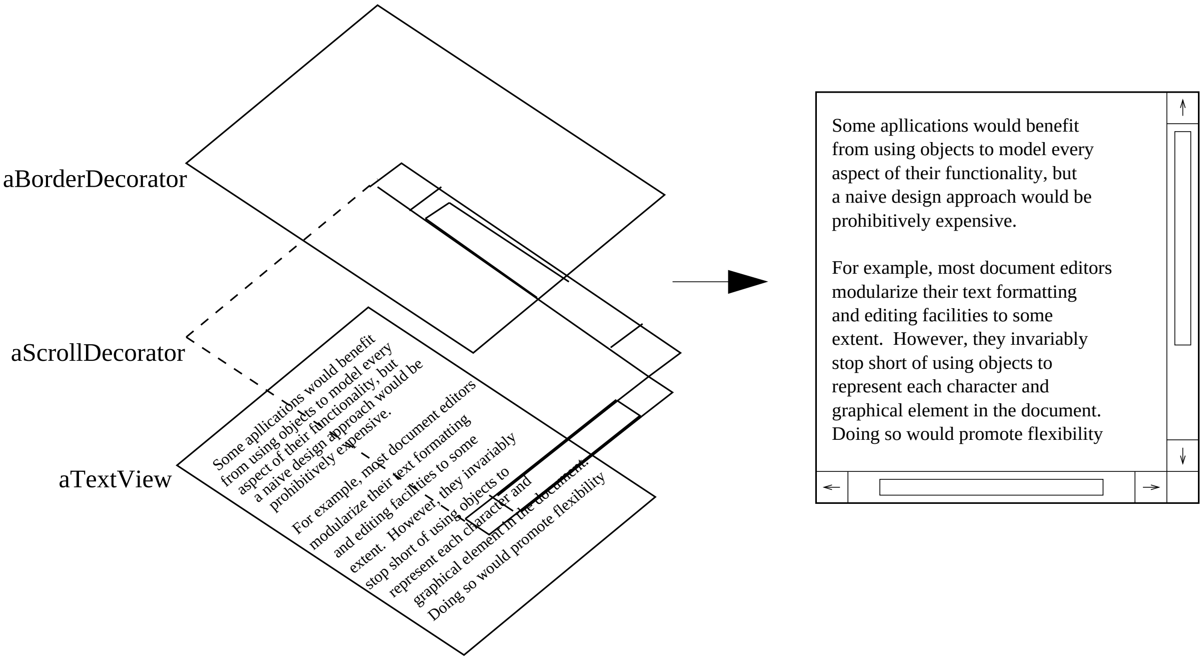
- **Intent**: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- **Motivation**:
  - In GUI applications, properties like borders and scrolling need to be provided in a flexible way to individual user interface components
  - In general, add additional responsibilities to individual objects, not to an entire class
  - Inheritance can be used but is inflexible because it is static.

- **Applicabilty**: Use this pattern
  - 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, there could be an explosion of subclasses to support every combination.
- **Participants**:
  1. **Component**: defines an interface for objects that can have responsibilities added to them dynamically
  2. **ConcreteComponent**: defines an object to which additional responsibilities can be attached
  3. **Decorator**: maintains a reference to a Component object and defines an interface that conforms to Component's interface
  4. **ConcreteDecorator**: adds responsibilities to the component

- **Collaborations**:
  - Decorator forwards requests to its Component object
  - Decorator may optionally perform additional operations before and after forwarding the request
- **Consequences**:
  - + Provides more flexibility than inheritance: responsibilities can be added and removed at run time by simply attaching and detaching them; properties can be added multiple times
  - + Enables functionalities to be incrementally added through composition, thus avoiding feature-laden classes high up in the hierarchy
  - – A decorator acts as a transparent enclosure; a decorated component is not to be mistaken for the component itself
  - – Use of Decorators can often result in systems composed of numerous little objects whose interactions are difficult to learn, understand, and debug.

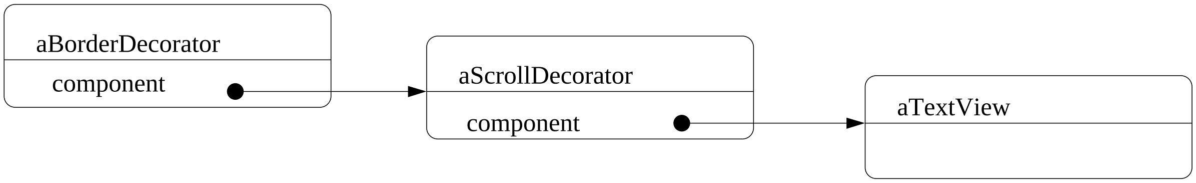
- **Implementation**:
  1. A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class
  2. To ensure a conforming interface, components and decorators must descend from a common *lightweight* Component class
- **Related Patterns**:
  - **Adapter**: A decorator only changes an object's responsibilities, not its interface;an adapter will give an object a totally new interface
  - **Composite**: A decorator is like a degenerate composite with only one component. It is not intended for object aggregation. On the other hand, it adds additional responsibilities
  - **Strategy**: A decorator enables to change the skin of an object; a strategy enables to change the guts

Motivational Example for Decorator

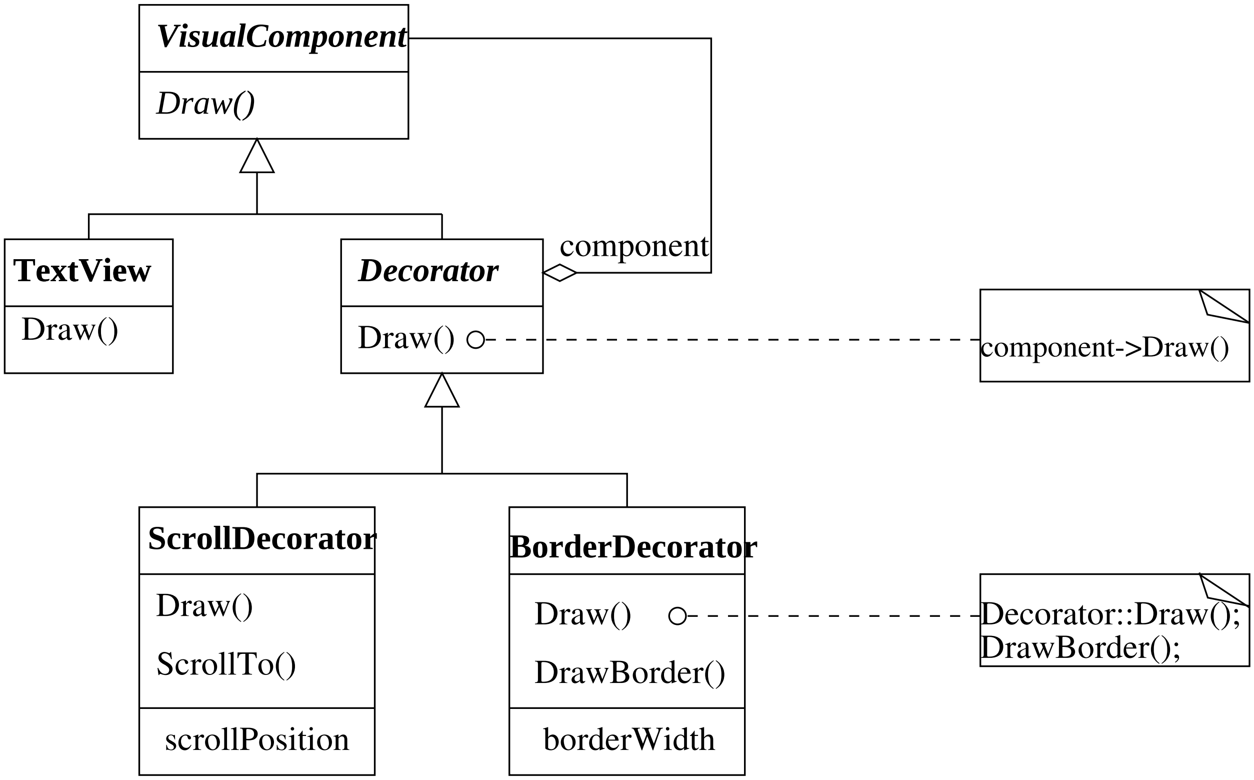


Some apllications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

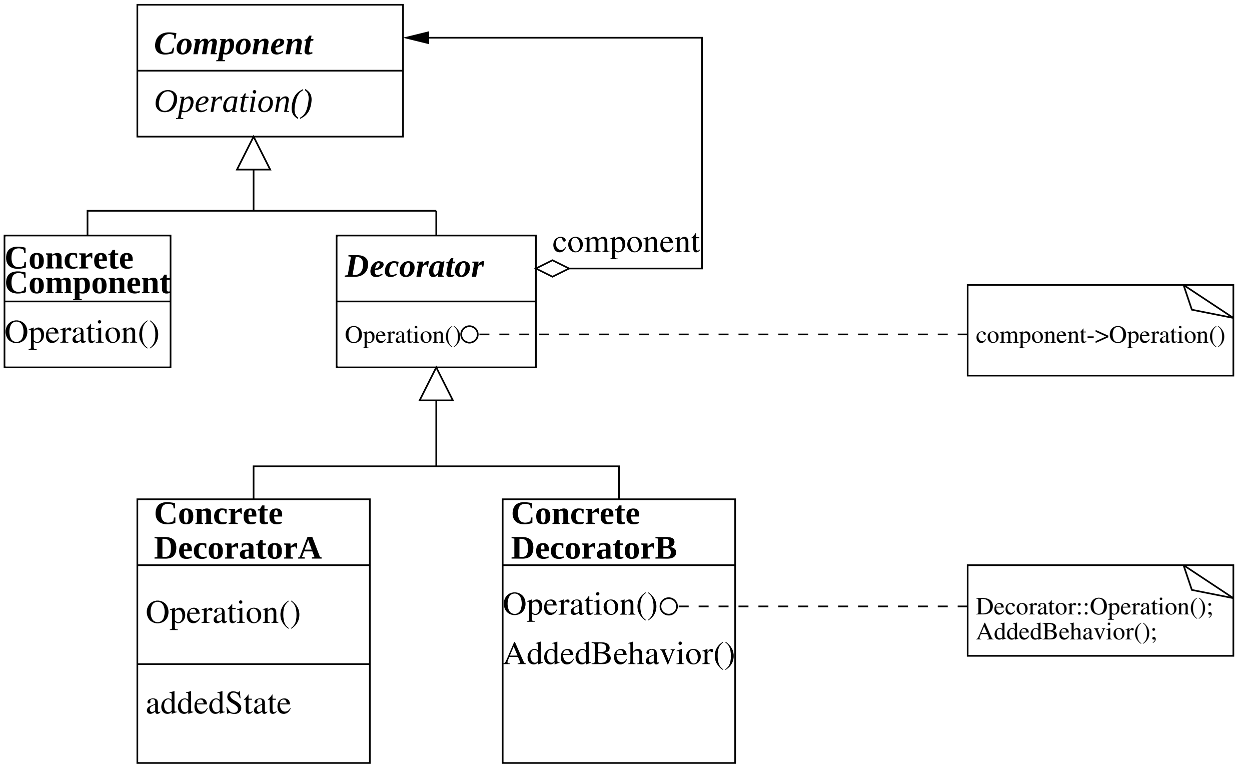
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility



Decorator for Motivational Example



Structure of Decorator Pattern



Bridge (Handle/Body) Pattern

- **Intent**: Decouple an abstraction from its implementation so that the two can vary independently
- **Motivation**:
  - When an abstraction can have several possible implementations, one can use inheritance; however, this will bind an implementation to the abstraction permanently
  - To make client code platform independent
- **Applicability**: Use this pattern when
  - it is required to avoid a permanent binding between an abstraction and its implementation (for example when the implementation must be selected or switched at run time)
  - both abstractions and implementations should be extensible by subclassing

  - changes in the implementation of an abstraction should have no impact on the clients
  - there is a proliferation of classes which indicates the need for splitting an object into two parts
  - you want to share an implementation among multiple objects

- **Participants**:
  1. **Abstraction** : defines the abstraction's interface and maintains a reference to an object of type Implementor
  2. **RefinedAbstraction**: extends the interface defined by Abstraction
  3. **Implementor**: defines the interface for implementation classes (this interface could be different from that of Abstraction)
  4. **ConcreteImplementor**: implements the interface of the Implementor

- **Collaborations**:

  1. Abstraction forwards client requests to its Implementor object

- **Consequences**:

  + decouples interface and implementation; consequently,

  - ∗ implementation of an abstraction can be configured at run time
  - ∗ an object can change its implementation at run time
  - ∗ eliminates compile-time dependencies on the implementation
  - ∗ encourages layering and better structuring

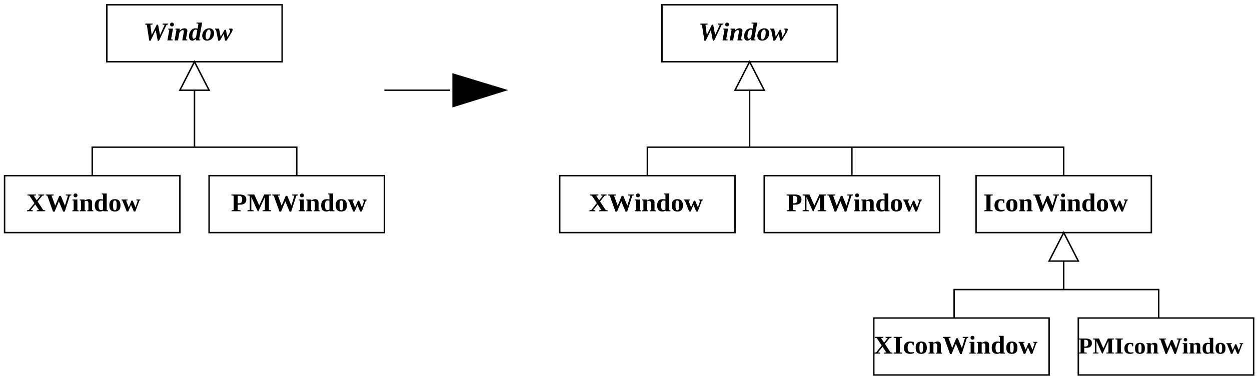
  + the Abstraction and Implementor hierarchies can be extended independently

  + hide implementation details from clients

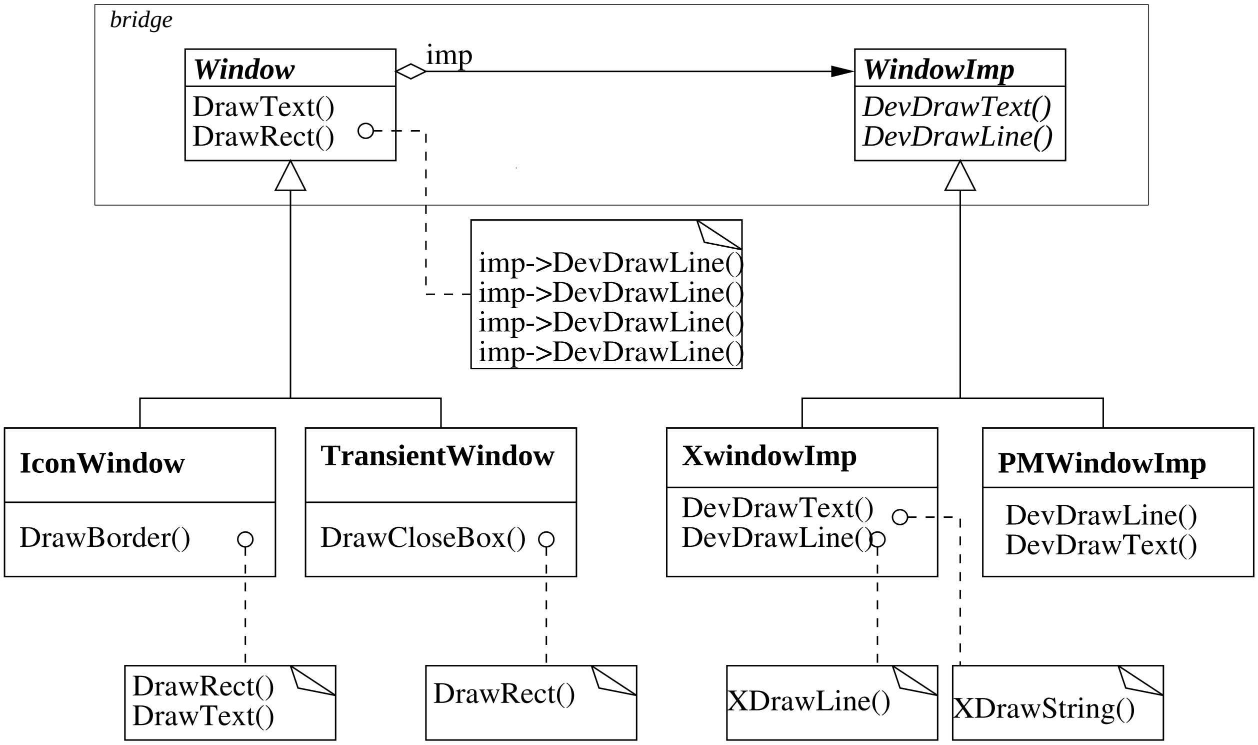
- **Implementation**:
  1. In situations where there is only one implementation, creating an abstract Implementor class is not necessary, but is stil
  2. It is important to resolve how, when, and where to decide which implementor class to instantiate
  3. A true Bridge cannot be implemented with multiple inheritance

- **Related Patterns**
  - **Abstract Factory**: An Abstract Factory ca create and configure a particular bridge
  - **Adapter**: The Adapter pattern makes unrelated classes work together and is applie to systems after they are designed. Bridge is used up-front.

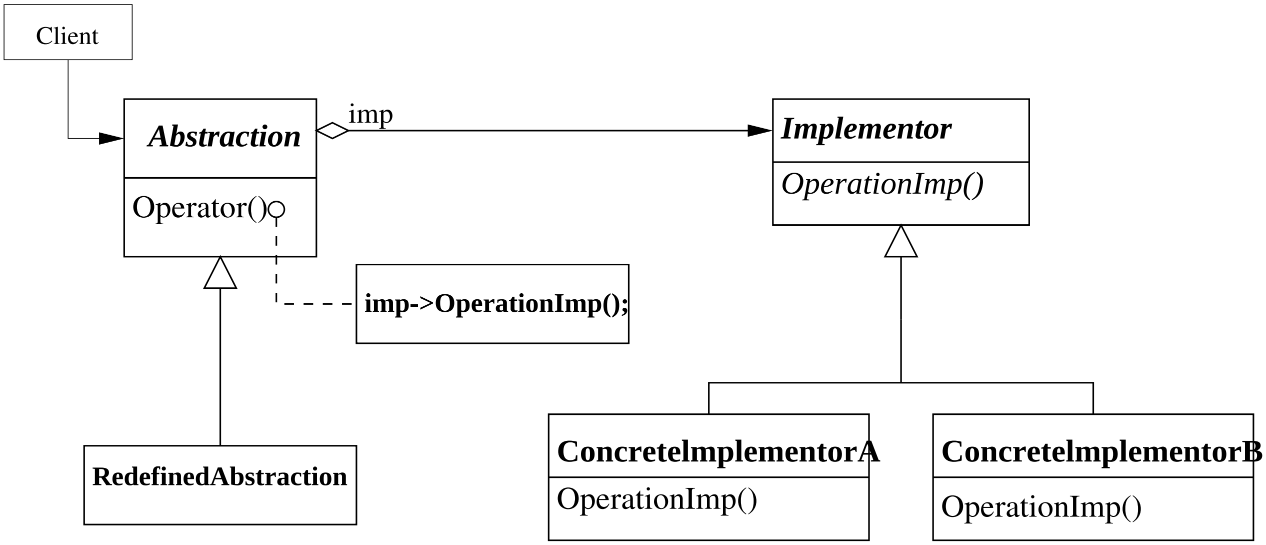
Motivation for Bridge Pattern



Example of Bridge Pattern



Structure of Bridge Pattern



Adapter (Wrapper)

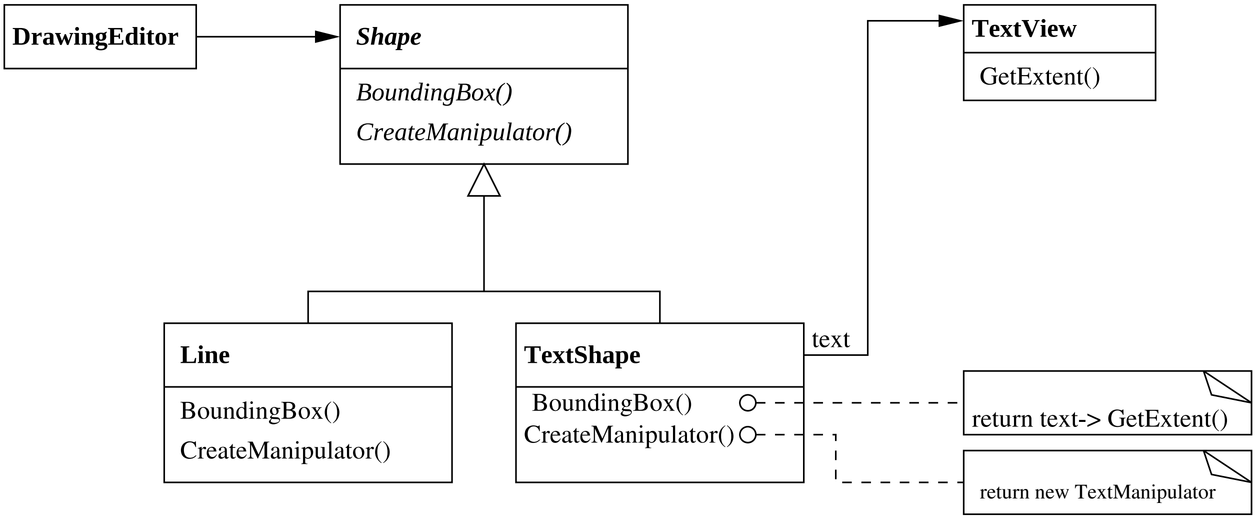
- **Intent**:
  - Convert interface of a class into another interface that clients expect
  - Lets classes work together that couldn't otherwise because of incompatible interfaces
- **Motivation**: Often toolkit classes are not reusable only because the interface does not match the domain-specific interface an application requires
- **Key Idea**: To make a class adapt to another, use either inheritance or composition

- **Applicability**: Use this pattern when:
  - one wants to use an existing class whose interface does not match the one you need
  - it is needed to create a reusable class that cooperates with unrelated or unforeseen classes (classes with incompatible interfaces)
  - (applies to object adapters only) you need to use several existing subclasses but it is impractical to adapt their interface by subclassing everyone
- **Participants**:
  - **Target**: defines a domain-specific interface
  - **Client**: collaborates with objects conforming to the Target interface
  - **Adaptee**: defines an existing interface that needs adopting
  - **Adapter**: adapts interface of Adaptee to the target interface

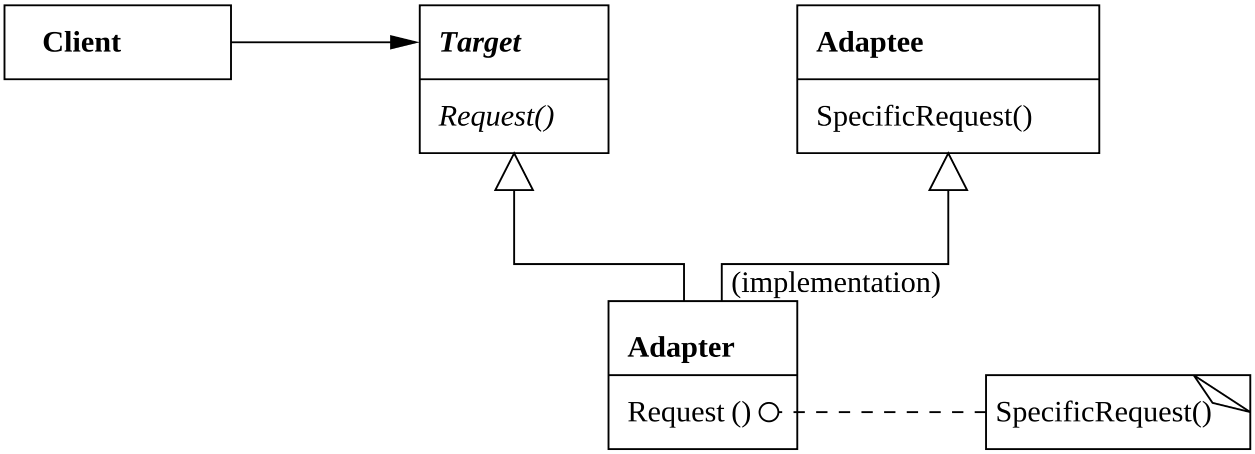
- **Collaborations**:
  - Clients call operations on an Adapter instance
  - In turn, Adapter calls Adaptee operations that carry out the request
- **Consequences**:
  - **Class Adapters**
    - ∗ adapts Adaptee to Target by committing to a concrete Adapter class; hence will not work if we need to adapt a class and all its subclasses
    - ∗ lets Adapter override some of Adaptee's behavior
    - ∗ introduces only one object, and no additional pointer indirection is needed to get to the adaptee
  - **Object Adapters**
    - ∗ lets a single Adapter work with many Adaptees
    - ∗ Adapter can also add functionality to all Adaptees at once
    - ∗ it is harder to override Adaptee behavior

- **Related Patterns**:
  - **Bridge**: Has a structure similar to that of Adapter, but has a very different intent
  - **Decorator**: enhances another object without changing its interface and is thus more transparent
  - **Proxy**: defines a surrogate for another object and does not change its interface

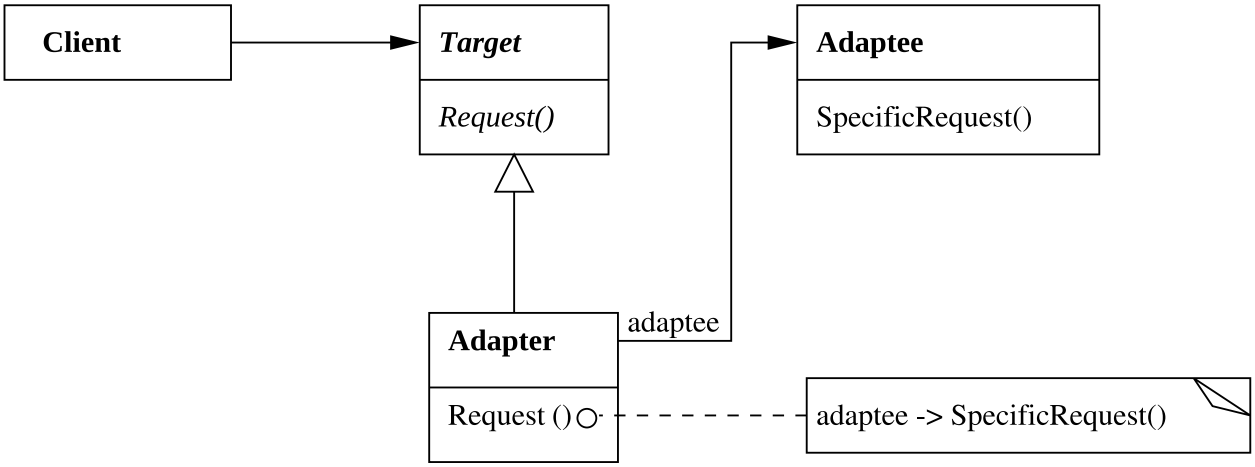
Motivational Example for Adapter



Adapter Pattern



Adapter Pattern



Client

Target

Request()

Adaptee

SpecificRequest()

Adapter

Request ()

adaptee

adaptee -> SpecificRequest()

Facade Pattern

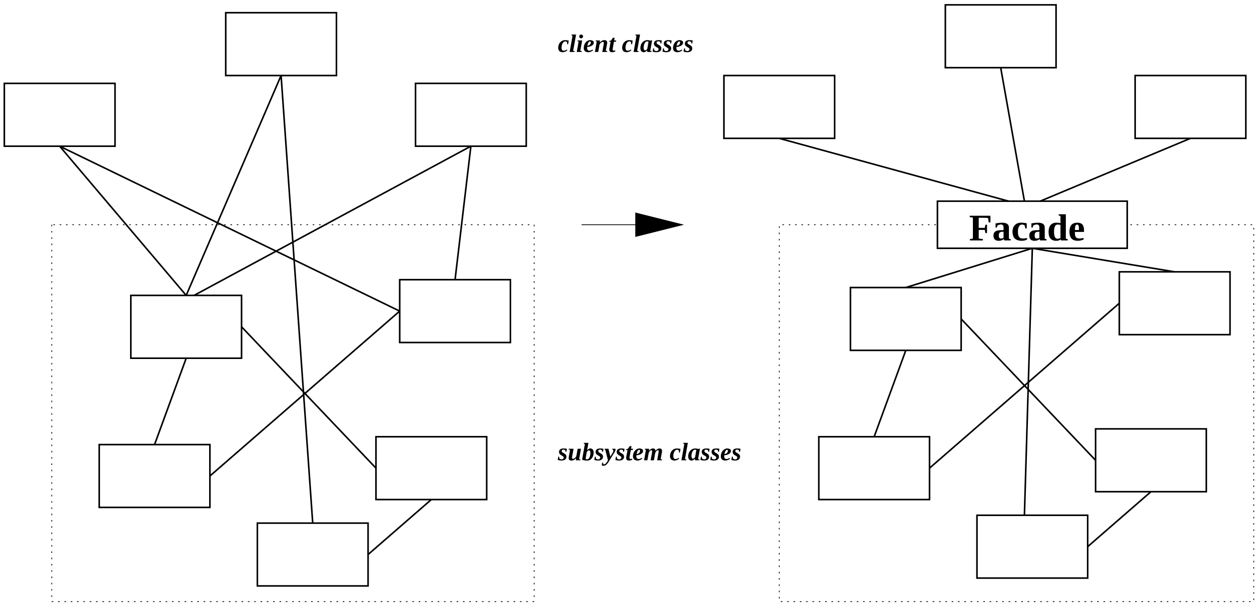
- **Intent**:
  - Provide a unified interface to a set of interfaces in a subsystem
  - Defines a higher-level interface that makes the subsystem easier to use
- **Motivation**: Often applications need direct access to different subsytems of a system, for example, a compiler.
- **Key Idea**: To make a class adapt to another, use either inheritance or composition

- **Applicability**: Use this pattern when:
  - you want to provide a simple interface to a complex subsystem (a simple default view of the subsystem that is good enough for most clients)
  - there are many dependencies between clients and the implementation classes of an abstraction; facade will decouple the subsystem from clients and other subsystems thus promoting subsystem independence and portability
  - you want to layer your subsystems; facade will define an entry point to each subsystem level
- **Participants**:
  - **Facade**: knows which subsystem classes are responsible for a request and delegates client requests to appropriate subsystem objects.
  - **Subsystem Classes**: implement subsystem functionality and handle work assigned by the Facade object; do not have knowledge of the facade to the Target interface

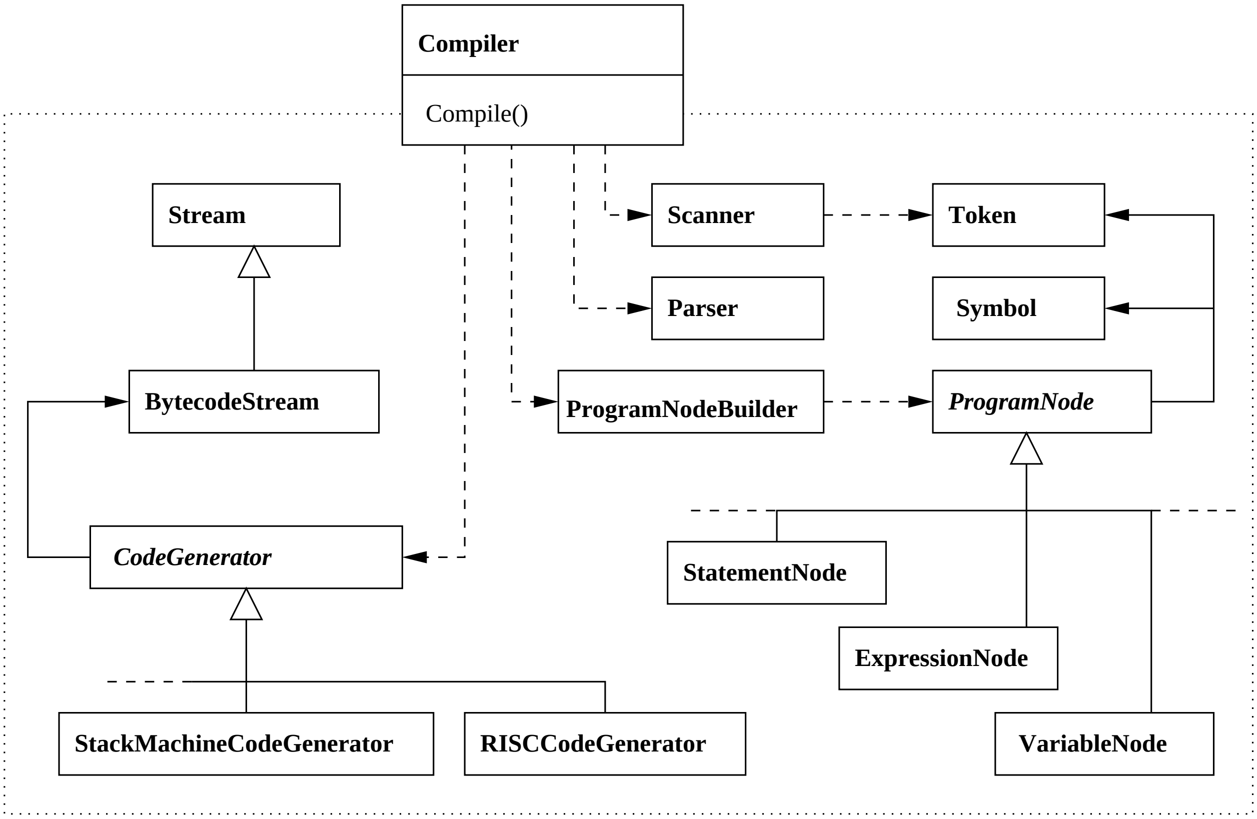
- **Collaborations**:
  - Clients communicate with the subsystem by sending requests to Facade, which forwards them to appropriate subsystem object(s).
  - Facade may have to do work of its own to translate its its interface to subsystem interfaces
- **Consequences**:
  - Facade shields clients from subsystem components, making the client simple and subsystem easier to use
  - Facade promotes weak coupling between the subsystem and its clients; this lets you vary the components of subsystem without affecting clients significantly
  - Facade does not prevent applications from using subsystem classes if they need to; thus one can choose between ease of use and granularity

- **Related Patterns**:
  - **Abstract Factory**: can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Can also be used as an alternative to facade to hide platform specific classes.
  - **Mediator**: abstracts functionality of existing classes, like the Facade does; however, a Mediator's colleagues are aware of the Mediator

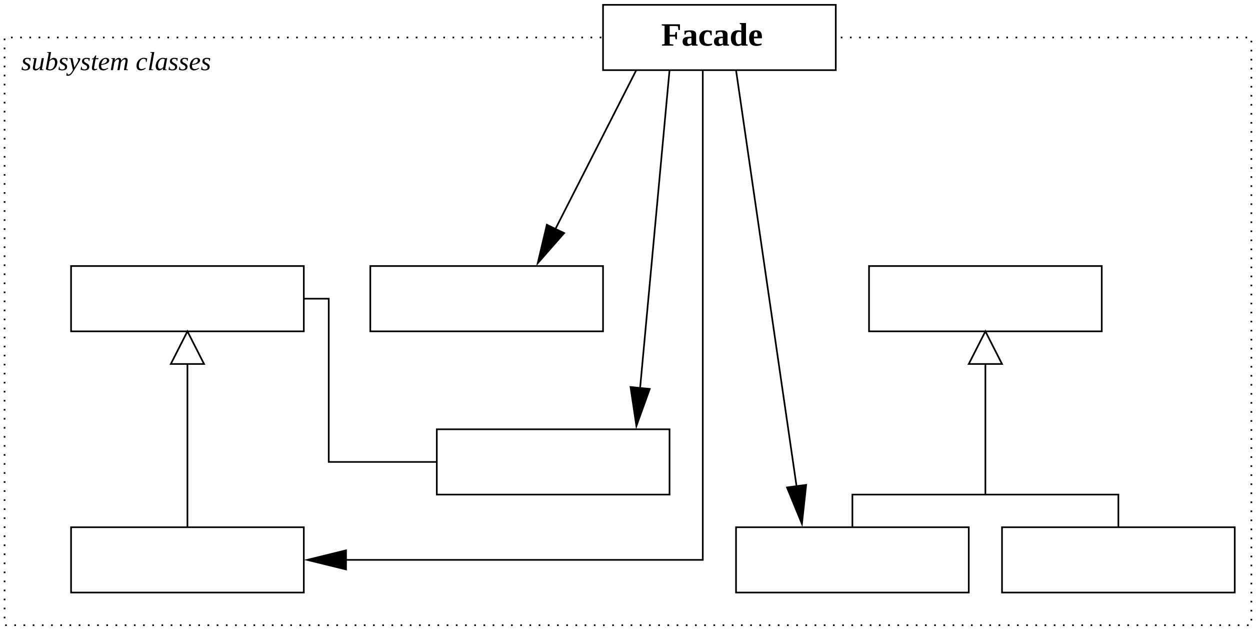
Motivational Example for Facade Pattern



Facade Pattern



Facade Pattern



Flyweight Pattern

- **Intent**: Use sharing to support large numbers of fine-grained objects efficiently
- **Motivation**: Some applications such as document editors can benefit immensely by using a large number of objects to promote flexibility at the finest levels in the application; however this makes run time overheads prohibitive
- **Key Idea**: Define shared objects which can be used in multiple contexts simultaneously through distinction between intrinsic (context-independent) state and extrinsic (context-dependent) state

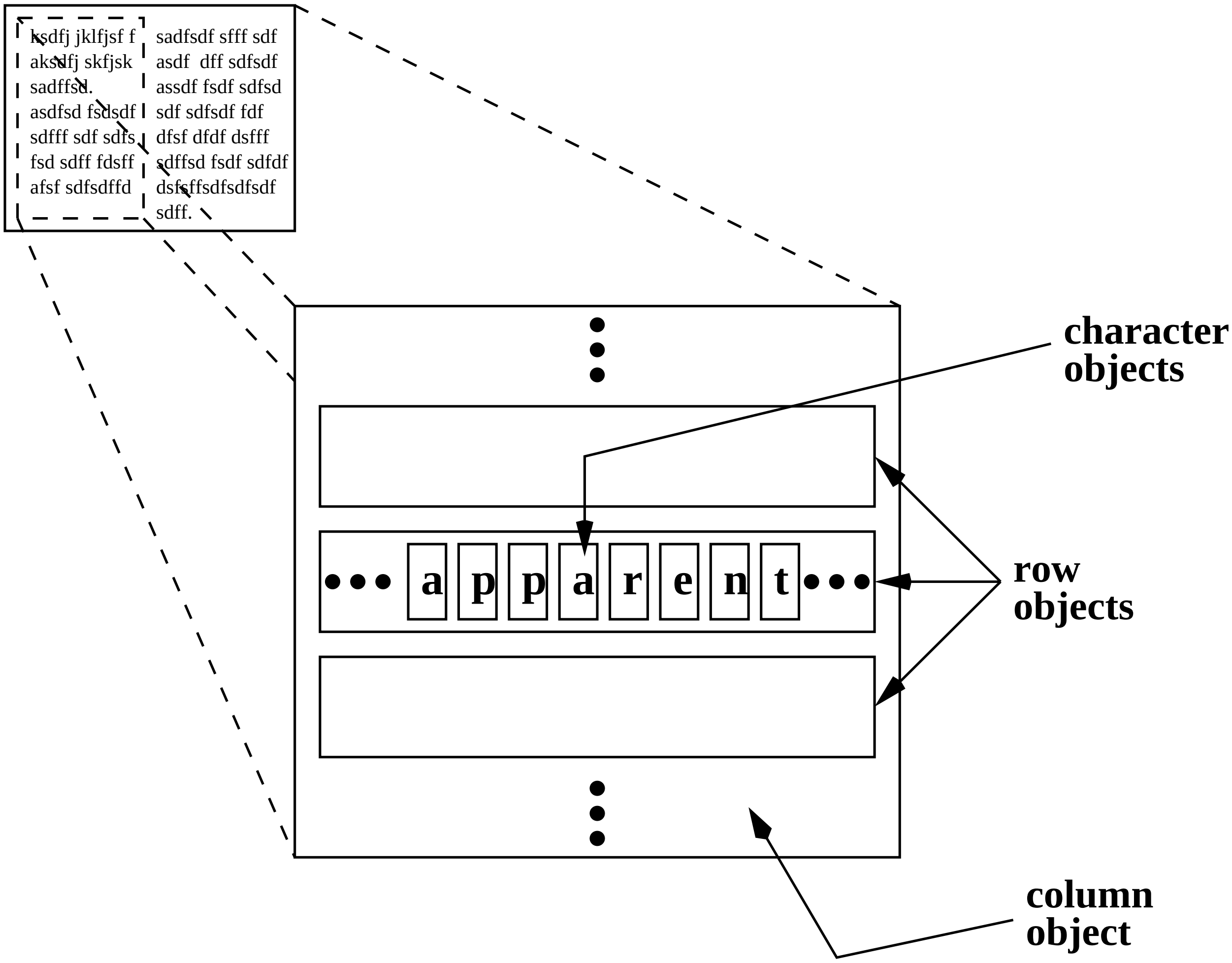
- **Applicability**: Use this pattern when all the following are true:
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic.
  - Many groups of objects can be replaced by relatively few shared objects once extrinsic state is removed
  - The application does not depend on object identity
- **Participants**:
  - **Flyweight**: declares an interface through which flyweights can receive and act on extrinsic state
  - **ConcreteFlyweight**: implements the Flyweight interface and adds storage for intrinsic state

- **UnsharedConcreteFlyweight**: implements the Flyweight and represents a Flyweight subclass that is not shared
- **FlyweightFactory**: creates and manages Flyweight objects; ensures that flyweights are shared properly
- **Client**: maintains a reference to flyweight(s) computes or stores the extrinsic state of flyweight(s).

- **Collaborations**:
  - State that a flyweight needs to function must be characterized as either intrinsic or extrinsic
  - Clients pass the extrinsic state to the flyweight when they invoke its operations
  - Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly

- **Consequences**:
  - Flyweights may introduce some run time costs which are however offset by space savings due to sharing of flyweights
  - Storage savings are enhanced by:
    - more flyweights being shared
    - increasing the amount of shared state
    - computing rather than storing the extrinsic state
- **Related Patterns**:
  - **Composite**: Flyweight is often combined with the Composite to implement a logically hierarchical structure
  - It is best to implement **State** and **Strategy** objects as flyweights

Motivational Example for Flyweight

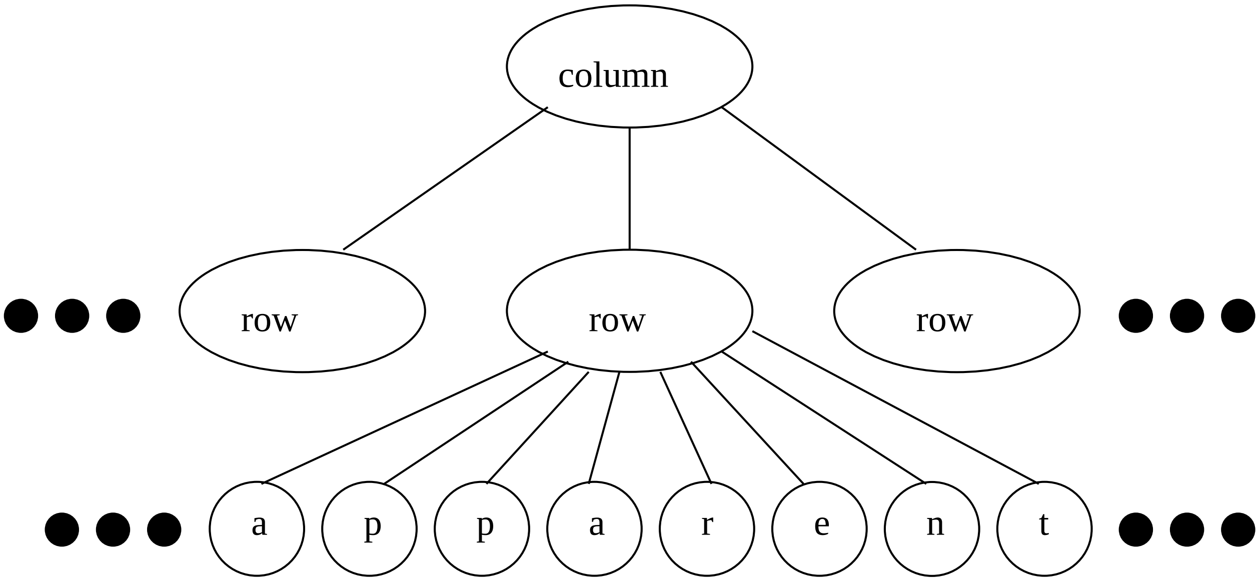


character objects

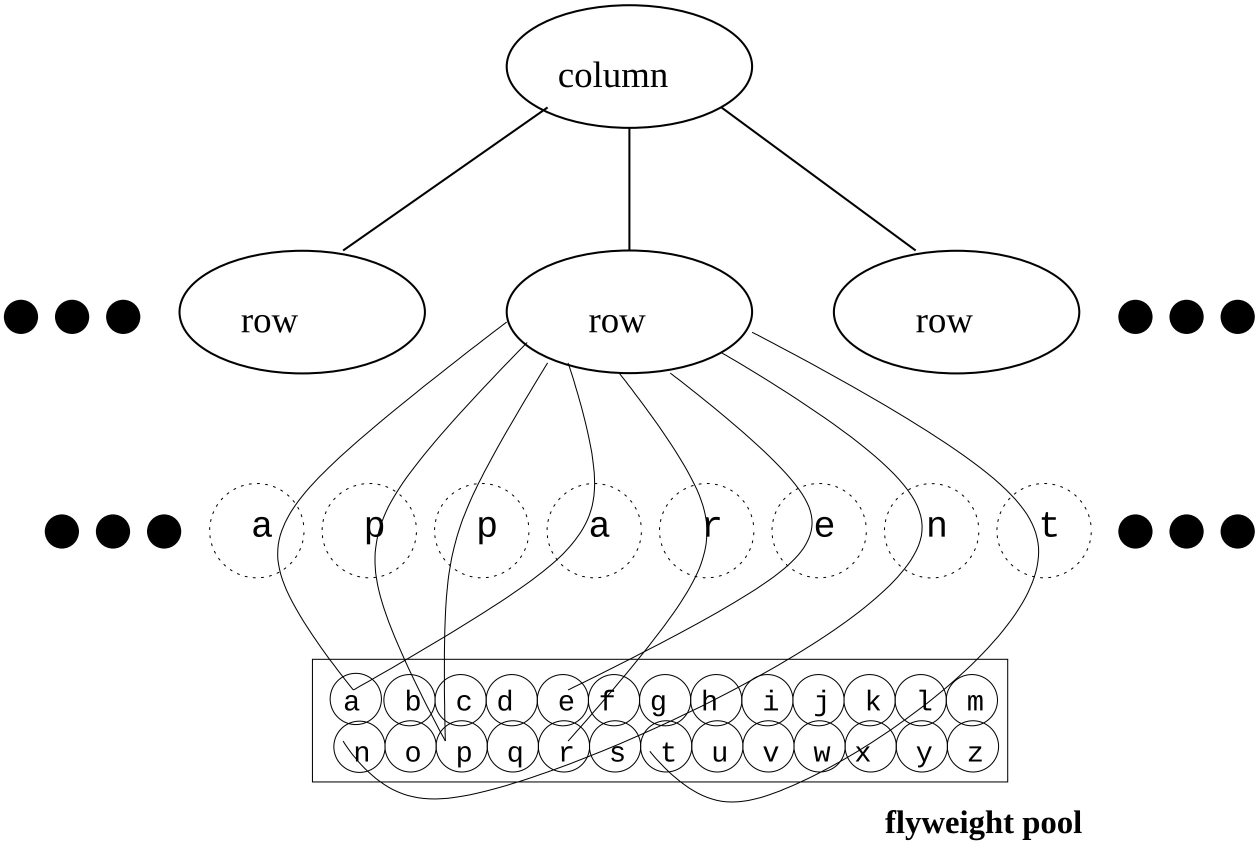
row objects

column object

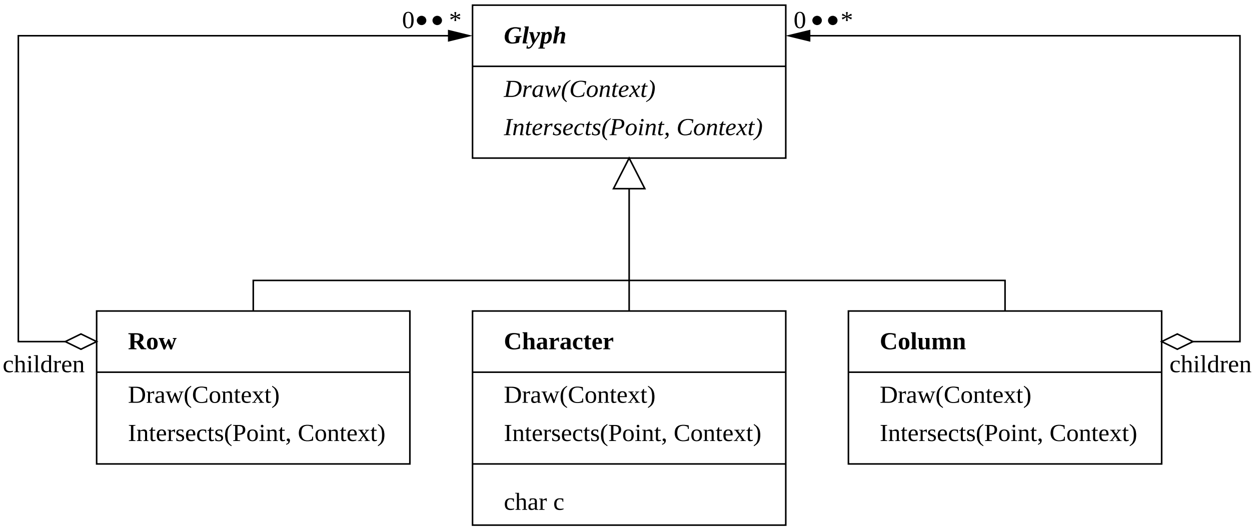
Flyweight Pattern



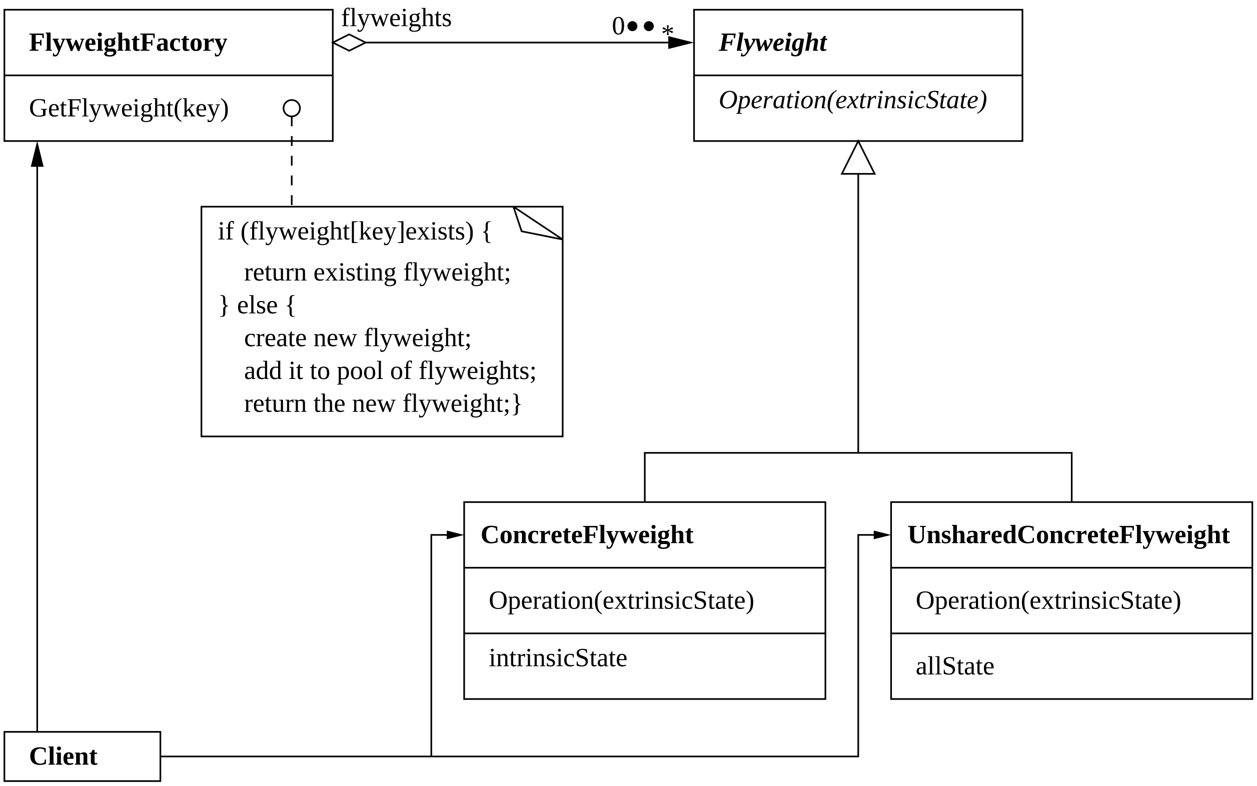
Flyweight Pattern



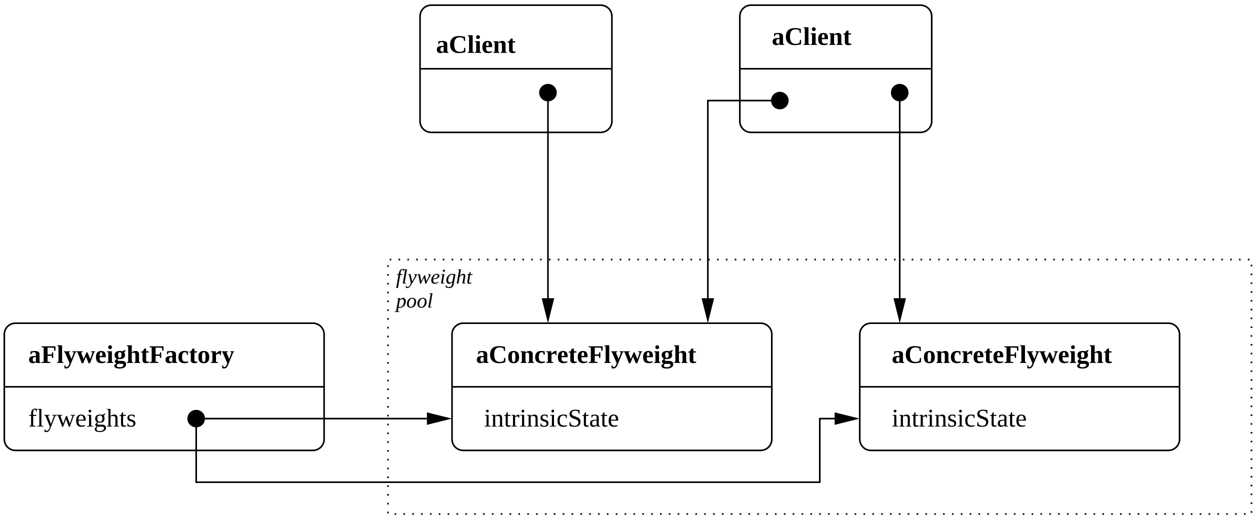
Flyweight Pattern



Flyweight Pattern


FlyweightFactory
GetFlyweight(key)
flyweights
0•• *
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

Flyweight Pattern



Proxy (Surrogate) Pattern

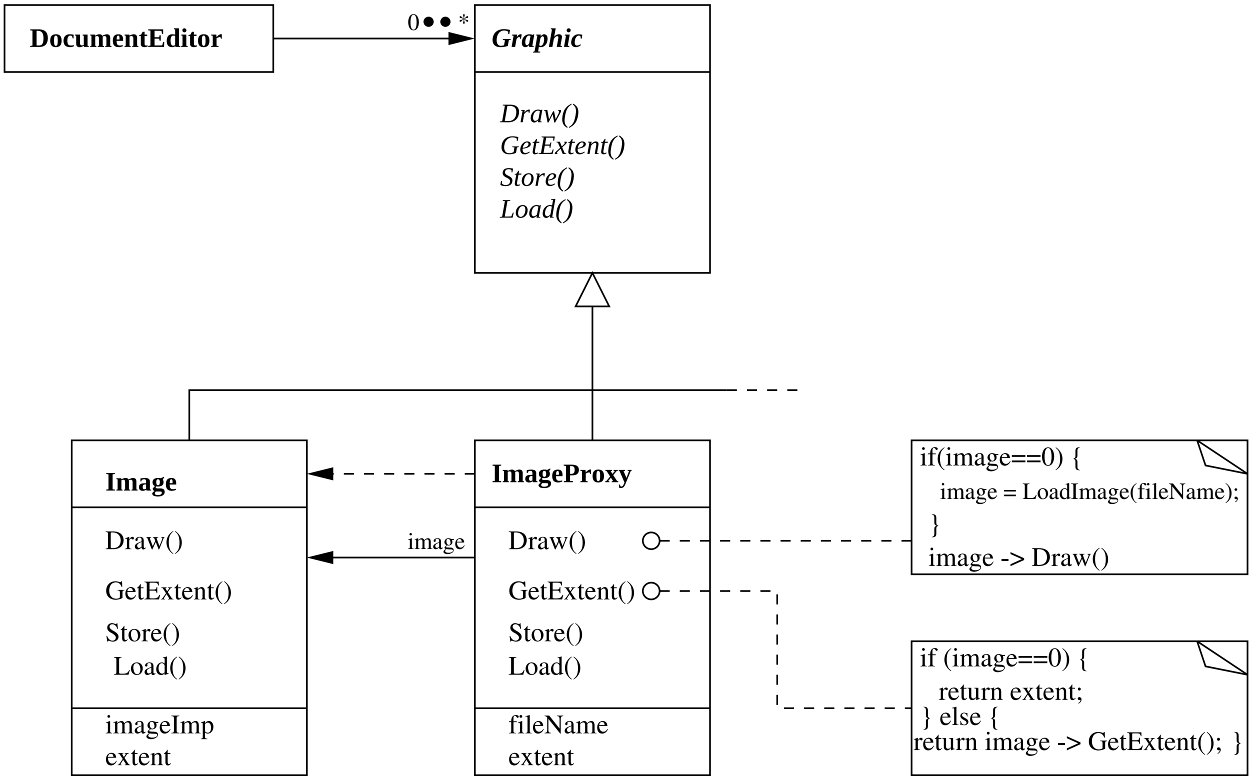
- **Intent**: provide a surrogate or placeholder for another object to control access to it.
- **Motivation**: Often expensive (heavy) objects are involved in applications (such as a large raster image in a document editor) which are best created only on demand
- **Key Idea**: Use a `proxy` object that acts as a stand-in for the real object and invoke the real object only when required

- **Applicability**: This is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Some such situations:
  - `remote proxy` provides a local representative for an object in a different address space (also called as `ambassador`)
  - `virtual proxy` creates expensive objects on demand
  - `protection proxy` controls access to the original object
  - `smart reference` is a replacement for a pointer that performs that performs additional actions when an object is accessed. Typical uses include: counting number of references, loading a persistent object on first reference, etc.

- **Participants**:
  - **Proxy**:
    - maintains a reference that lets the proxy access the real subject
    - 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 based on the kind of proxy
  - **Subject**: defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
  - **RealSubject**: 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
  - Protection proxies and smart references allow additional housekeeping tasks when an object is accessed
  - By using a proxy to postpone the copying process, we ensure that we pay the price of copying the object only if it is modified
- **Related Patterns**:
  - **Adapter**: provides a different interface to the object it adopts
  - **Decorator**: adds additional responsibilities to a class

Motivational Example for Proxy Pattern



DocumentEditor

0 • • *

Graphic

Draw()
GetExtent()
Store()
Load()

Image

Draw()
GetExtent()
Store()
Load()

imageImp
extent

image

ImageProxy

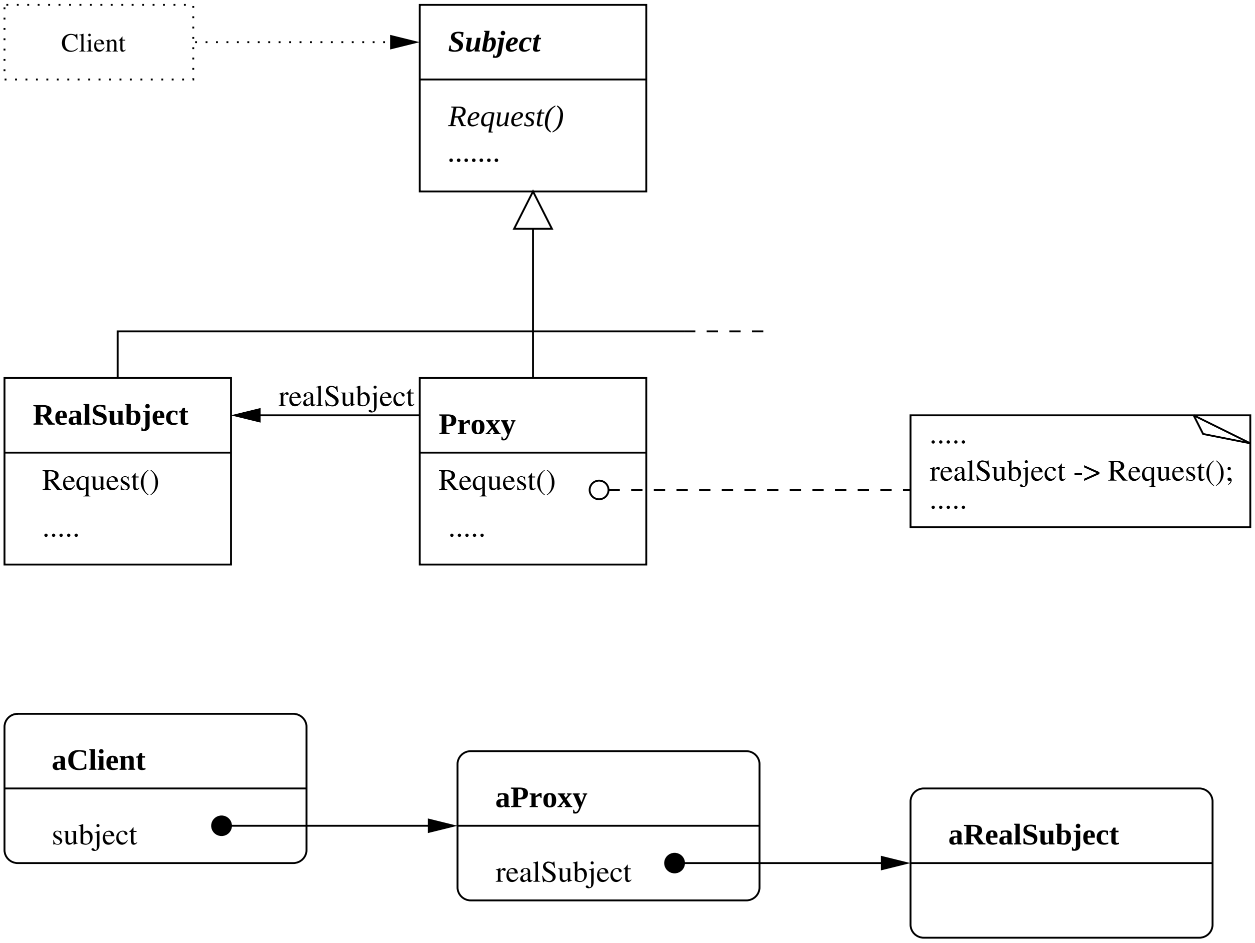
Draw()
GetExtent()
Store()
Load()

fileName
extent

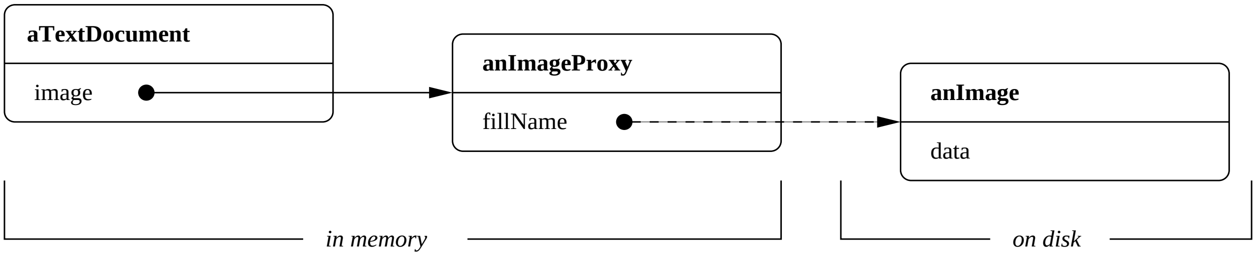
```
if(image==0) {
  image = LoadImage(fileName);
 }
 image -> Draw()
```

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

Proxy Pattern



Proxy pattern



Structural Patterns: Discussion

- Structural patterns often look alike since they rely on a small set of mechanisms for structuring classes and objects together
- Adapter and Bridge look similar. However,
  - Adapter makes things work `after` the classes are designed
  - Bridge plans up-front and makes the classes work `before` they are designed
- Composite and Decorator both use recursive decomposition but have vastly different intents
- Decorator and Proxy look alike, however have widely differing purposes
- Patterns can be combined:
  - A Proxy-Decorator can add functionality to a proxy
  - Decorator-Proxy can embellish remote objects

Behavioral Patterns

- Concerned with algorithms and the assignment of responsibilities among objects
- Describe both patterns of objects or classes `and` patterns of communication between them
- Characterize complex control flow that is difficult to follow at run time
- **Bahavioral Class Patterns** use inheritance to distribute behavior between classes
  - Template Method
  - Interpreter
- **Behavioral Object Patterns**
  - use object composition
  - encapsulate behavior in an object
  - delegating requests to other objects

Strategy (Policy) Pattern

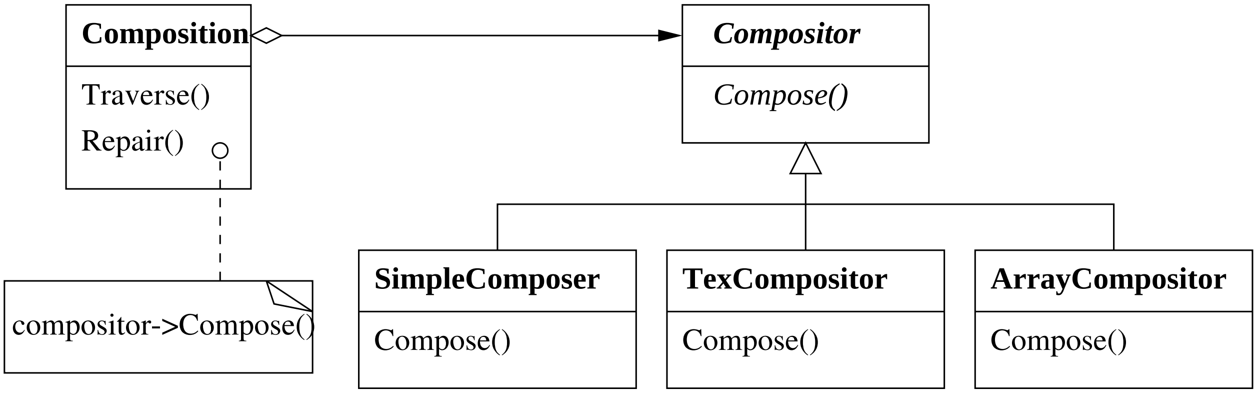
- **Intent**:
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Lets the algorithm vary independently of the client
- **Motivation**:
  - Availability of many alternative algorithms for a given problem
  - Different algorithms may be appropriate for different contexts
  - It is difficult to add new algorithms and vary existing ones if the logic is an integral part of the client

- **Applicability**: Use this pattern when
  - many related classes differ only in their behavior; strategies provide a way to configure a class with one of many behaviors
  - different variants of an algorithm are needed
  - an algorithm uses data that clients shouldn't know about; this pattern can be used to avoid exposing complex, algorithm specific data structures
  - a class defines many behaviors and these appear as multiple conditional statements in its operations

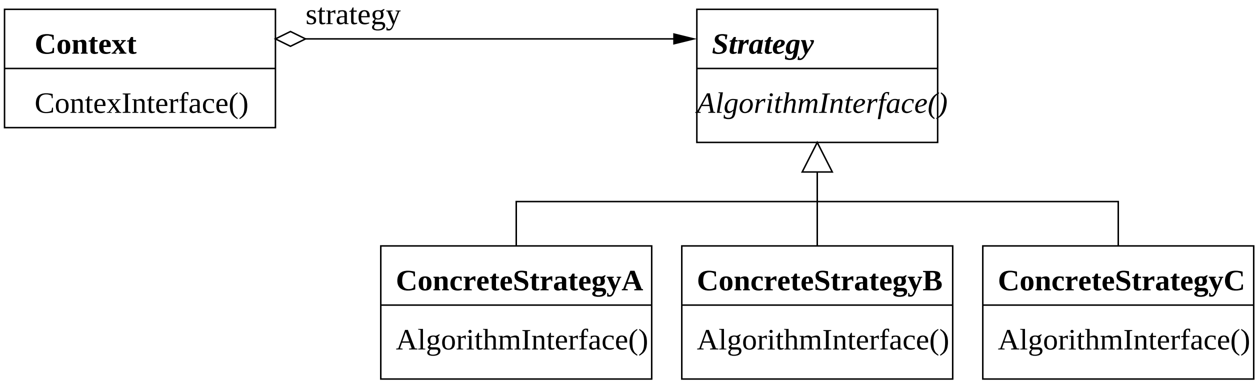
- **Participants**:
  1. **Strategy**:
     - declares an interface common to all supported algorithms
     - uses the interface to call the algorithm defined by a ConcreteStrategy
  2. **ConcreteStrategy**: implements the algorithm using the Strategy interface
  3. **Context**:
     - is configured with a ConcreteStrategy object
     - keeps a reference to a Strategy object
     - may define an interface that lets Strategy access its data
- **Collaborations**:
  - Strategy and Context collaborate to implement the chosen algorithm
  - A Context forwards requests from its clients to its strategy

- **Consequences**:
  - + elegantly implements families of related algorithms or behaviors for contexts to reuse
  - + effective alternative to subclassing
  - + eliminate conditional statements
  - + provides a choice of implementations
  - – clients must be aware of different Strategies
  - – communication overhead between Strategy and Context
  - – increased number of objects
- **Implementation**:
  - – The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice-versa
  - – Strategy objects may often be made optional

Motivational Example for Strategy



Structure of Strategy Pattern



Command (Action, Transaction) Pattern

- **Intent**: Encapsulate a request as an object, thereby enabling to parameterize clients with different requests, queue or log requests, and support undoable operations
- **Motivation**:
  - Often it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. e.g.: buttons, or menus in user interface toolkits
- **Applicability**: Use this pattern when
  - there is a need to parameterize objects by an action to perform
  - it is required to specify, queue, and execute requests at different times
  - there is a need to support undo.

- **Participants**:
  1. **Command**: declares an interface for executing an operation
  2. **ConcreteCommand**: defines a binding between a Receiver object and an action and implements Execute by invoking the corresponding operation(s) on Receiver.
  3. **Client**: creates a ConcreteCommand object and sets its receiver.
  4. **Invoker**: asks the command to carry out the request.
  5. **Receiver**: knows how to perform the operations associated with carrying out a request

- **Collaborations**:
  - The client creates a ConcreteCommand object and specifies its receiver
  - An Invoker object stores the ConcreteCommand object
  - The Invoker issues a request by calling Execute on the command.
  - 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, being objects, can be manipulated and extended like any other object
  - + Commands can be assembled into composite commands
  - + New Commands can be easily added

- **Implementation**:
  1. A command can have a wide range of abilities: from merely defining a binding between a receiver and the actions to implementing complex functionality
  2. To support undo and redo capabilities, a ConcreteCommand object needs to store additional information regarding the Receiver object, arguments to the operations, etc.
- **Related Patterns**:
  - **Composite** can be used to implement MacroCommands
  - **Memento** can keep state the command requires to undo its effect
  - **Prototype**: A command that must be copied before being placed on the history list acts as a Prototype

Motivational Example for Command

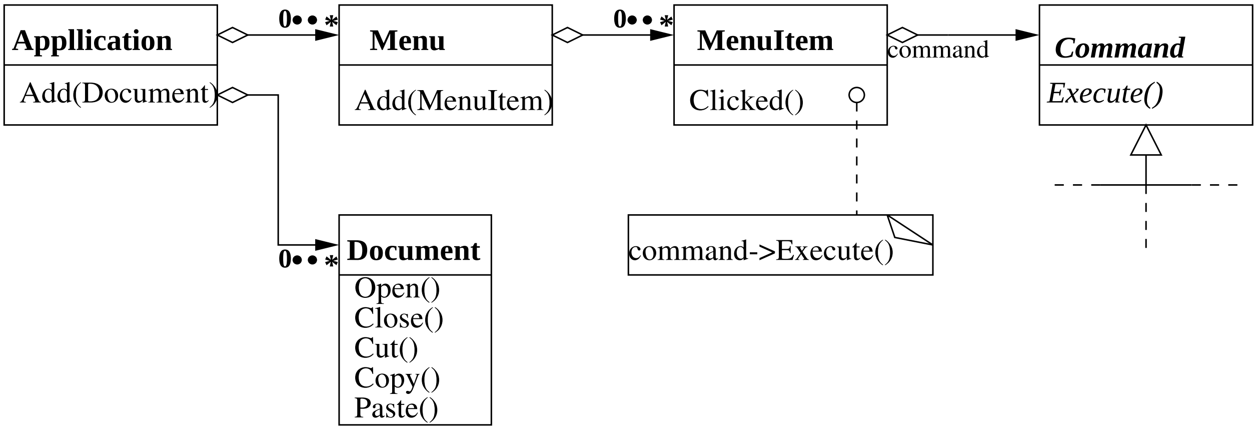
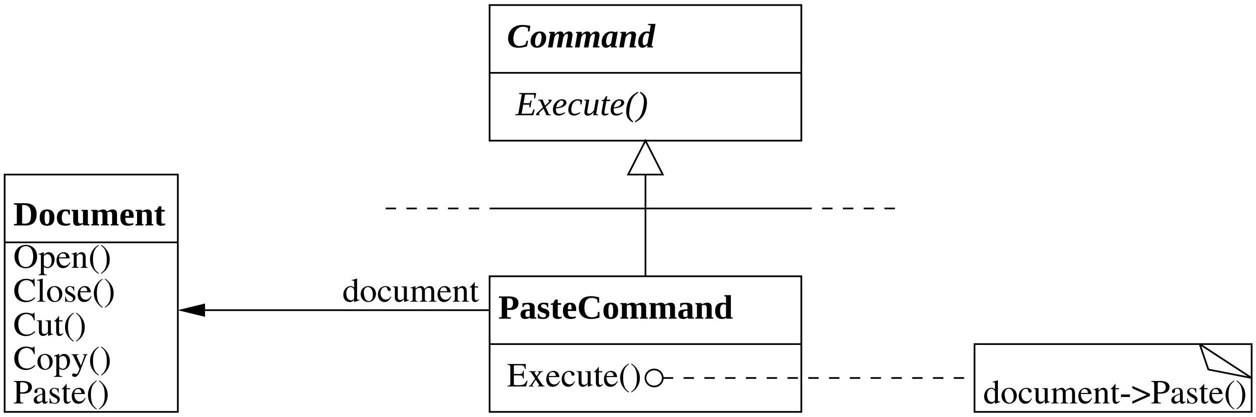
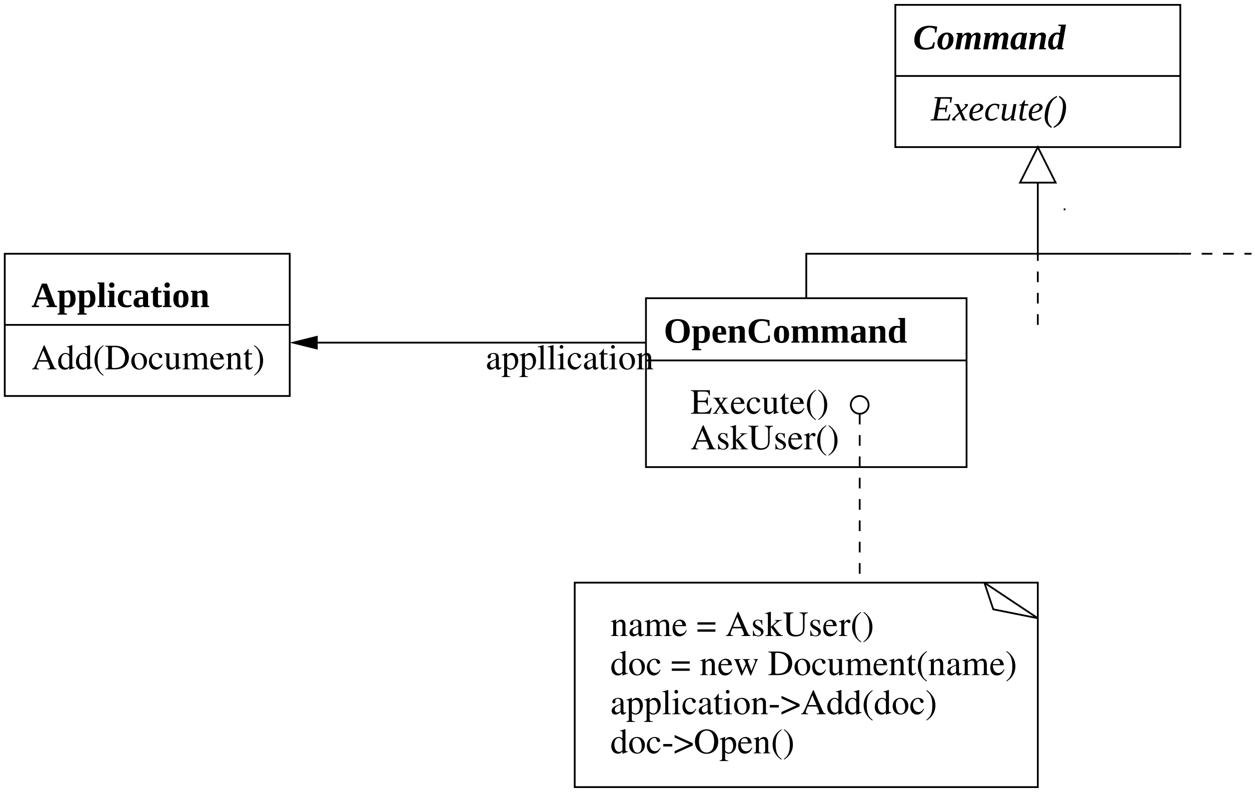


Illustration of Command Pattern



Usage of Command Pattern



Command

Execute()

Application

Add(Document)

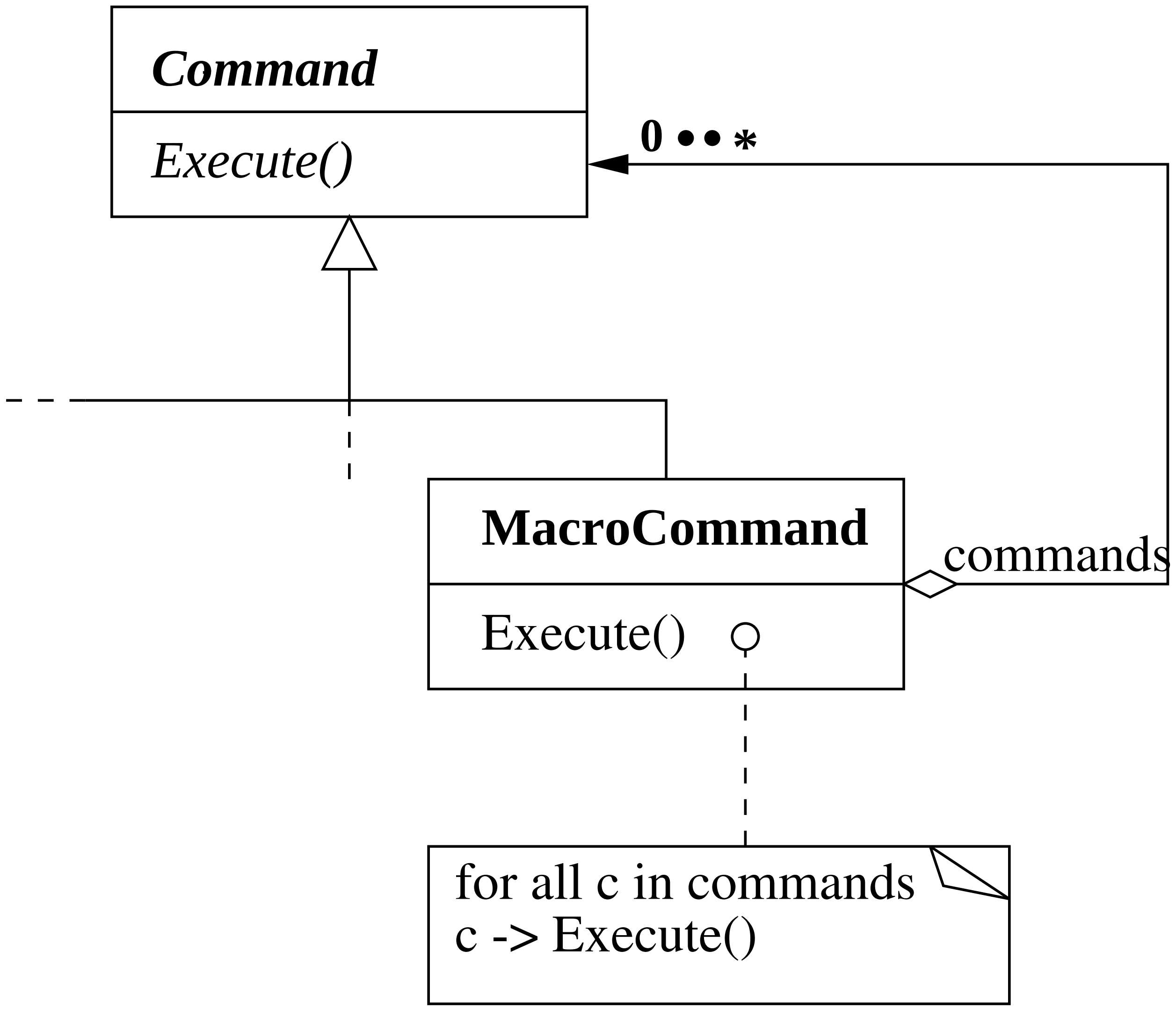
appllication

OpenCommand

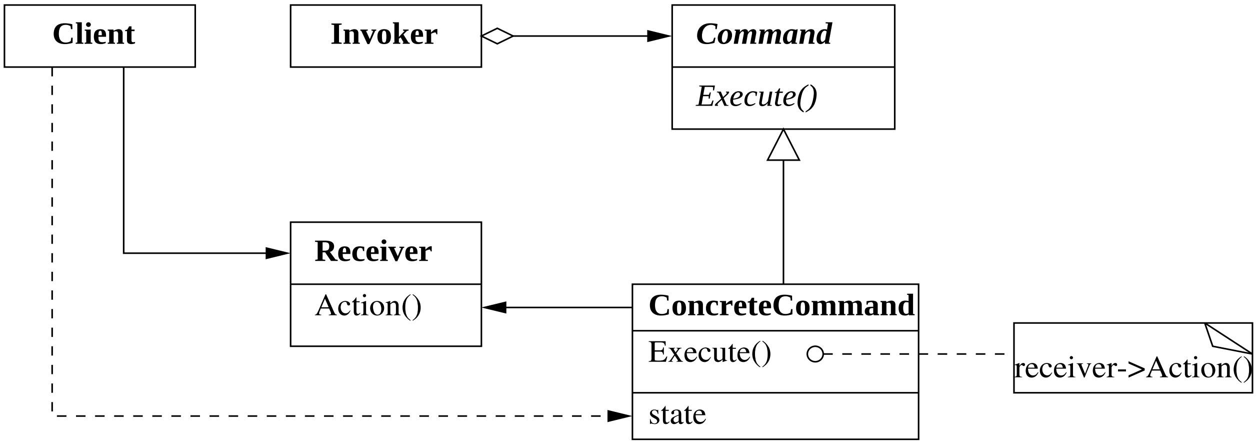
Execute()
AskUser()

```
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()
```

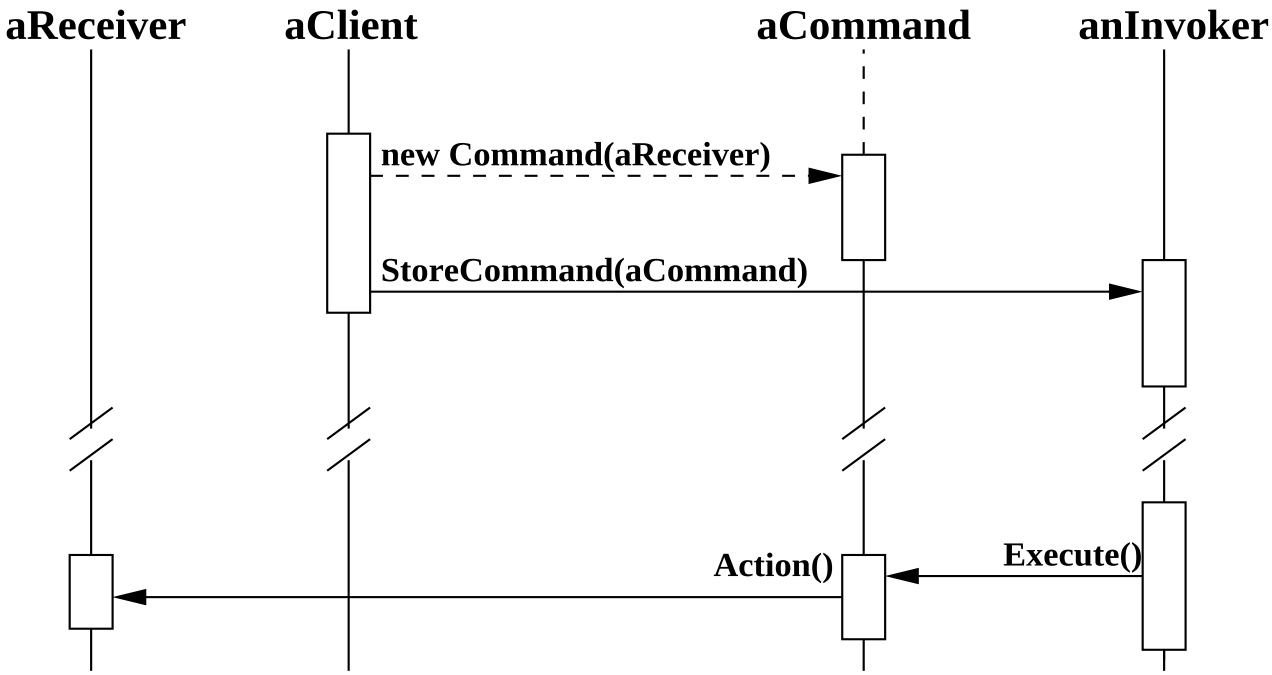
Command and Macrocommand



Structure of Command Pattern



A Sequence Diagram for Command



Iterator (Cursor) Pattern

- **Intent**: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- **Motivation**:
  - It is often required to access the individual elements of an aggregate object such as a list without exposing its internal structure
  - There is often a need to traverse these elements in different ways at different times
- **Key Idea**: Create a separate abstraction for access and traversal and pull out these responsibilities from the aggregate object

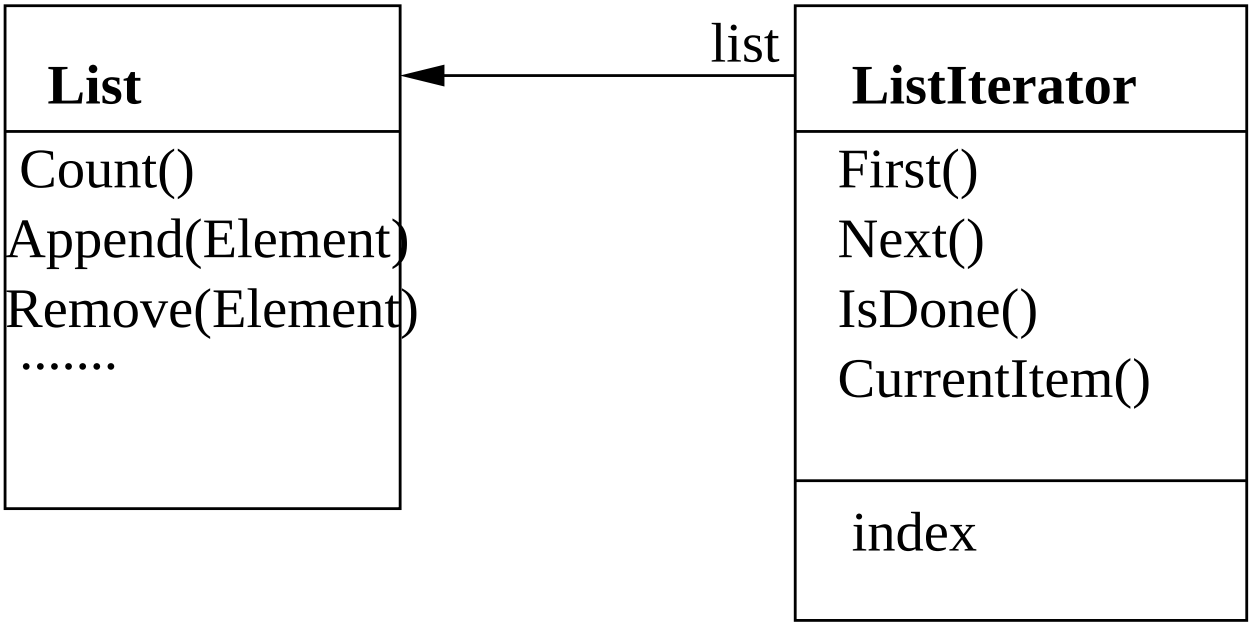
- **Applicability**: Use this pattern
  - to access an aggregate object's contents without exposing its internal representation
  - to support multiple traversals of aggregate objects
  - to provide a uniform interface for traversing different aggregate structures (and thus support polymorphic iteration)
- **Participants**:
  1. **Iterator**: defines an interface for accessing and traversing elements
  2. **ConcreteIterator**: implements the Iterator interface and keeps track of the current position in the traversal of the aggregate
  3. **Aggregate**: defines an interface for creating an Iterator object
  4. **ConcreteAggregate**: implements the Iterator creation to return an instance of the proper ConcreteIterator

- **Collaborations**:
  1. A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- **Consequences**:
  + Iterators make it easy to change traversal algorithms
  + Iterator's interface simplifies the aggregate's interface
  + Since an iterator keeps track of its own traversal state, one can have more than one traversal in progress at a given time
- **Implementation**:
  1. The iteration can be controlled by the client (external iterator) or by the iterator (internal iterator).
  2. The traversal algorithm can be defined by the aggregate or the iterator

3. Careful handling is required if the aggregate is required to be modified during the traversal
4. Recursive aggregate structures are best traversed using internal iterators

- **Related Patterns**:
  - **Composite**: Iterators are often applied to composite structures
  - **Memento**: An iterator can use a memento to capture the state of an iteration. The Iterator stores the memento internally.
  - **Factory Method**: Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass

Motivational Example for Iterator



List

Count()

Append(Element)

Remove(Element)

....

list

ListIterator

First()

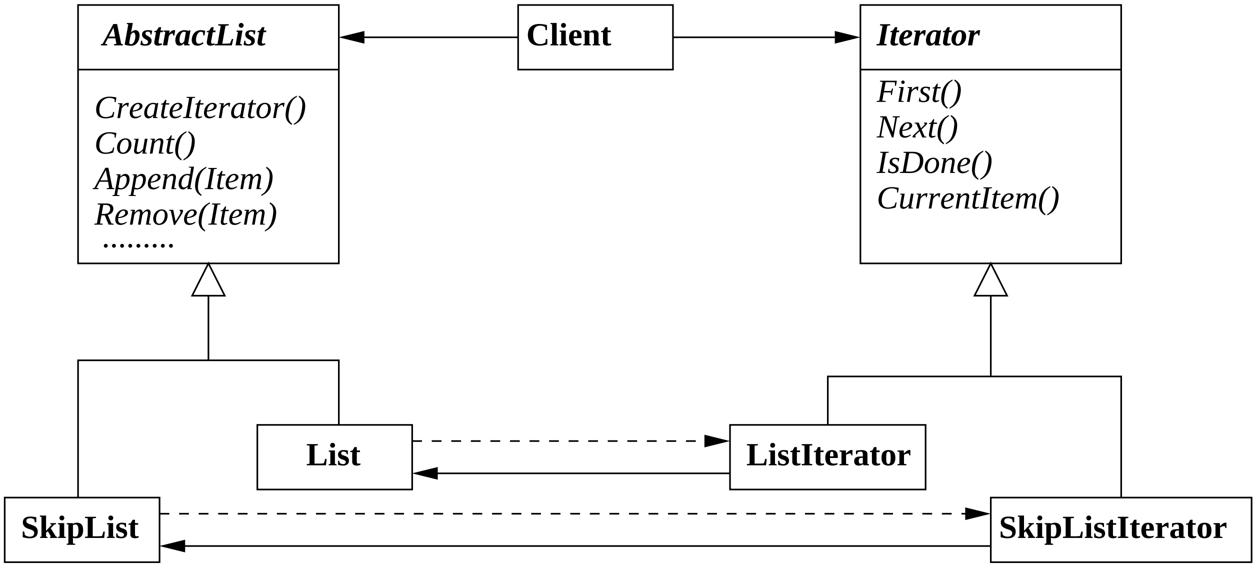
Next()

IsDone()

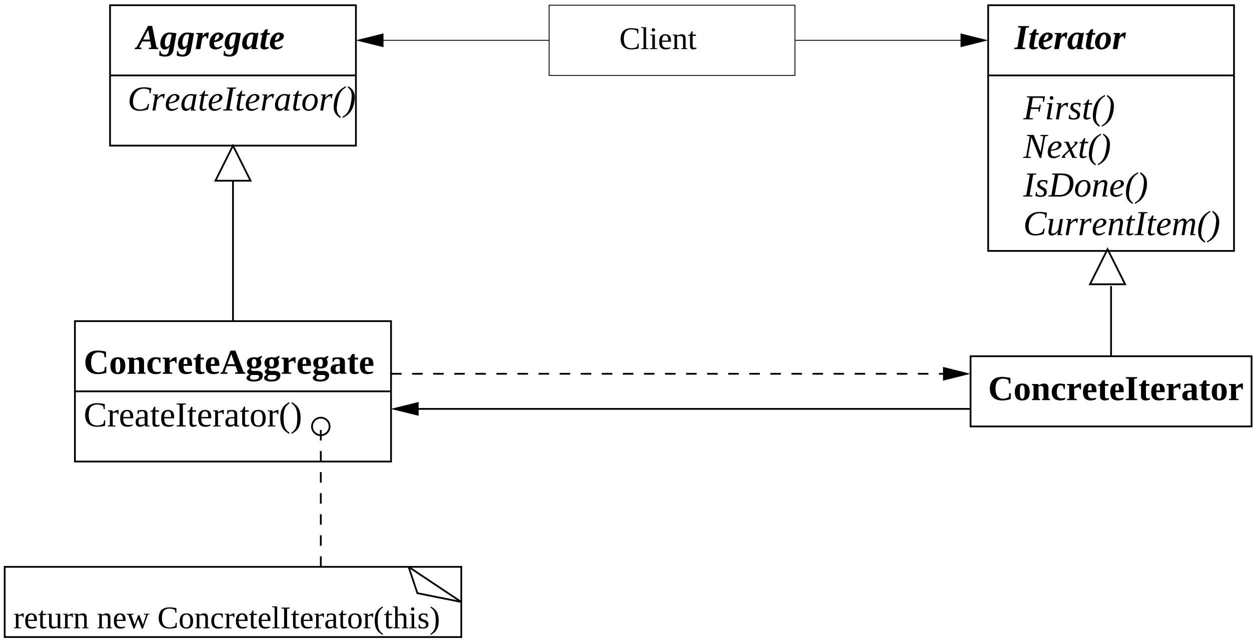
CurrentItem()

index

Relationships in Iterator



Structure of Iterator



Visitor Pattern

- **Intent**: Represent an operation to be performed on the elements of an object structure. Enable to define a new operation without changing the classes of the elements on which it operates
- **Motivation**:
  - In compilers, different related operations are carried out on syntax trees. Visitor avoids the need to include related operations as part of every subclass definition
- **Key Idea**: Keep related operations together by defining them in one class

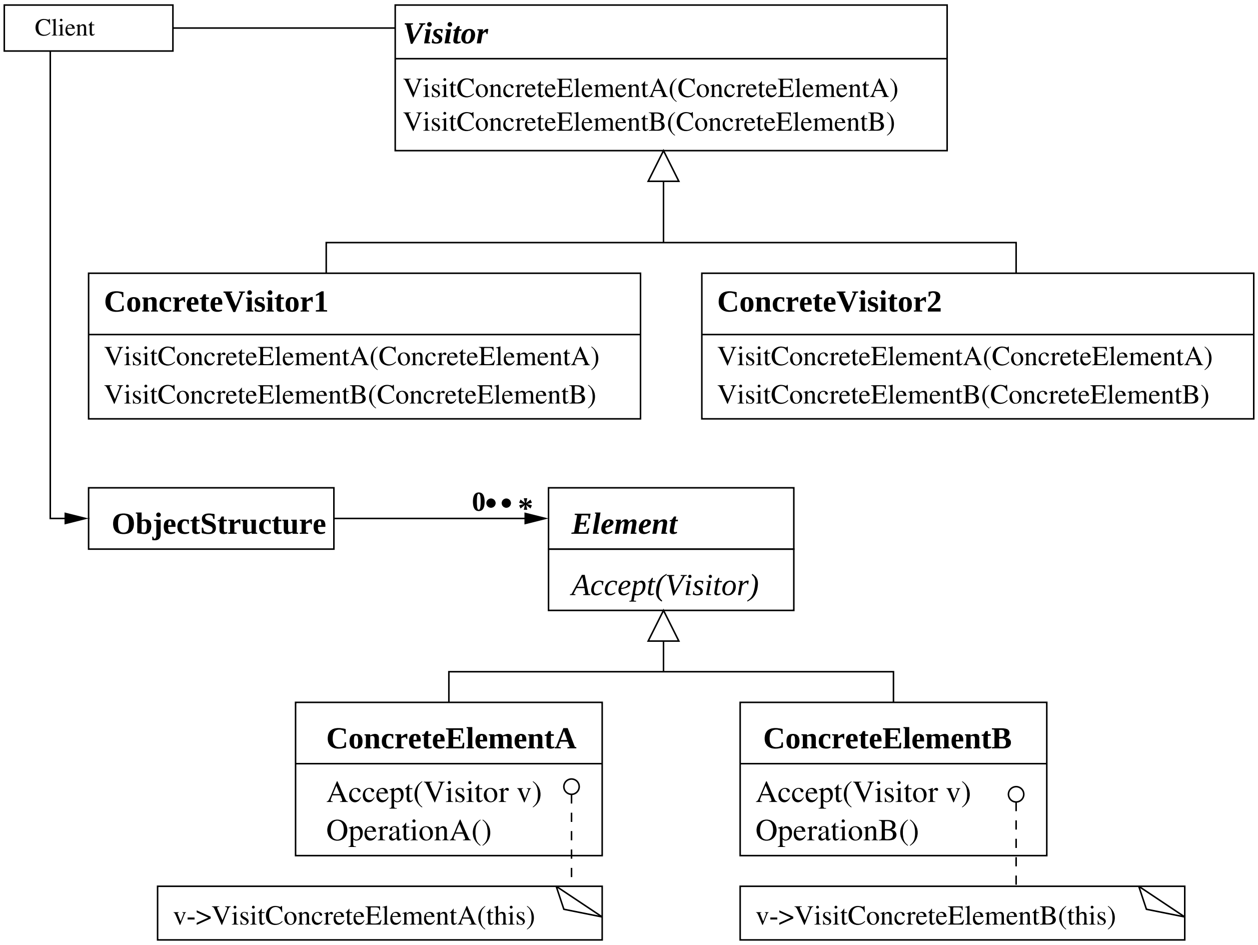
- **Applicability**: Use this pattern when
  - an object structure contains many classes of objects with differing interfaces and it is required to perform operations on these objects that depend on their concrete classes
  - many distinct and unrelated operations need to be performed on objects in an object structure and we would like to avoid polluting their classes with these operations
  - the classes defining the object structure rarely change, but often it is required to define new operations over the structure
- **Participants**:
  1. **Visitor**: declares a Visit operation for each class of ConcreteElement in the object structure
  2. **ConcreteVisitor**: implements each operation declared by Visitor

3. **Element**: defines an Accept operation that takes a visitor as an argument
4. **ConcreteElement**: implements an Accept operation that takes a visitor as an argument
5. **ObjectStructure**: can enumerate its elements and may provide a high-level interface to allow the visitor to visit its elements

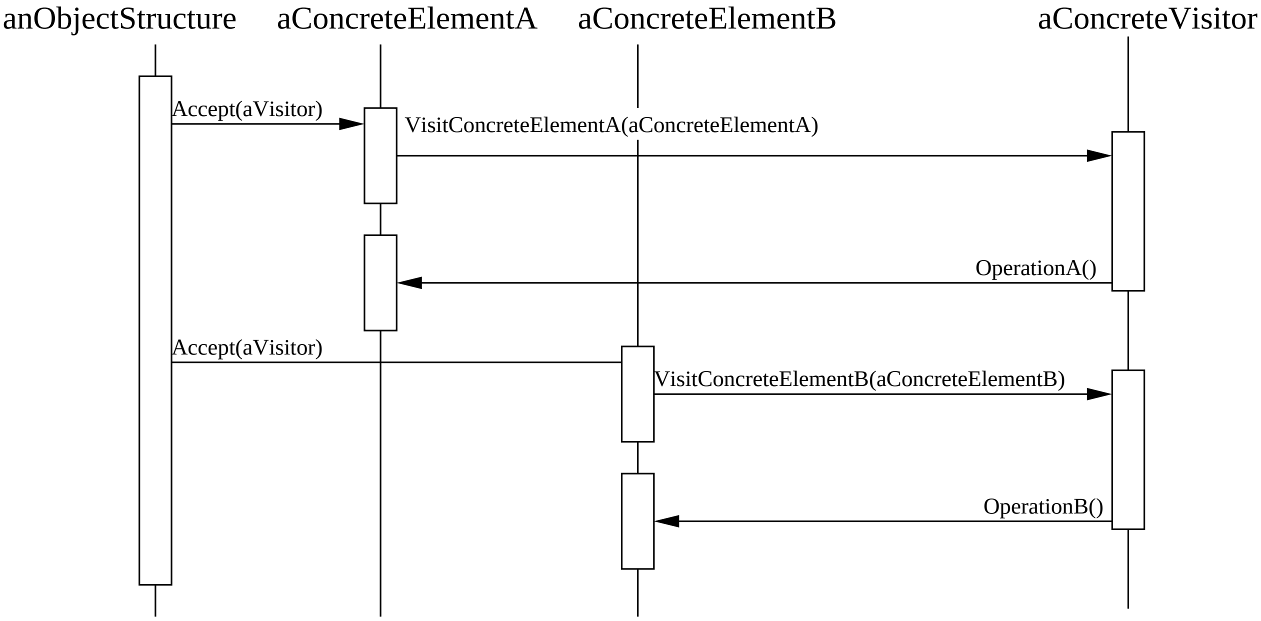
- **Collaborations**:
  1. A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure
  2. When an element is visited, the appropriate Visitor operation is called

- **Consequences**:
  - + Visitor makes adding new operations on an object structure easy
  - + A visitor gathers related operations and separates unrelated ones
  - + A visitor can visit objects that don't have a common parent class thus enabling visiting across object hierarchies
  - – Adding new ConcreteElement classes is hard
  - – To let visitors do their job, there may be a need to access an element's internal state, compromising its encapsulation
- **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 interpretation

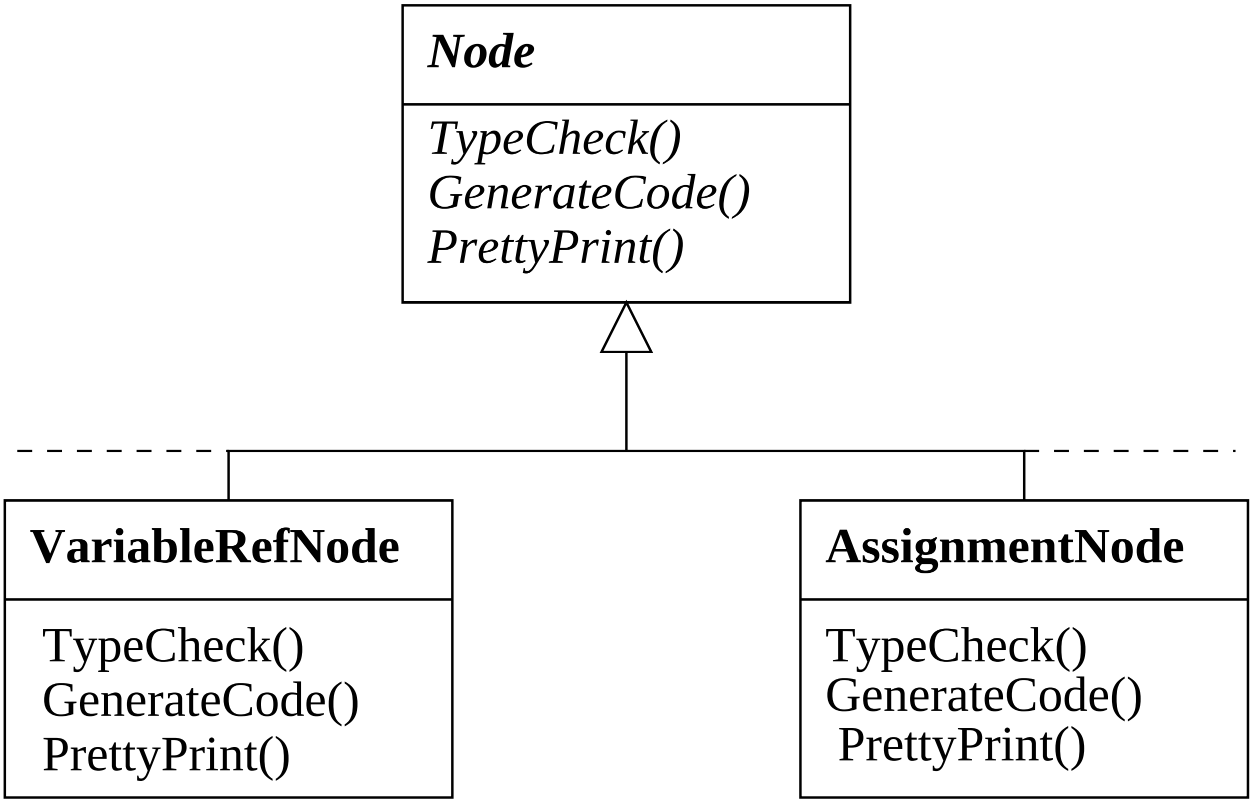
Structure of Visitor


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
0••*
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Interactions in Visitor Pattern



Motivational Example for Visitor



Node

TypeCheck()
GenerateCode()
PrettyPrint()

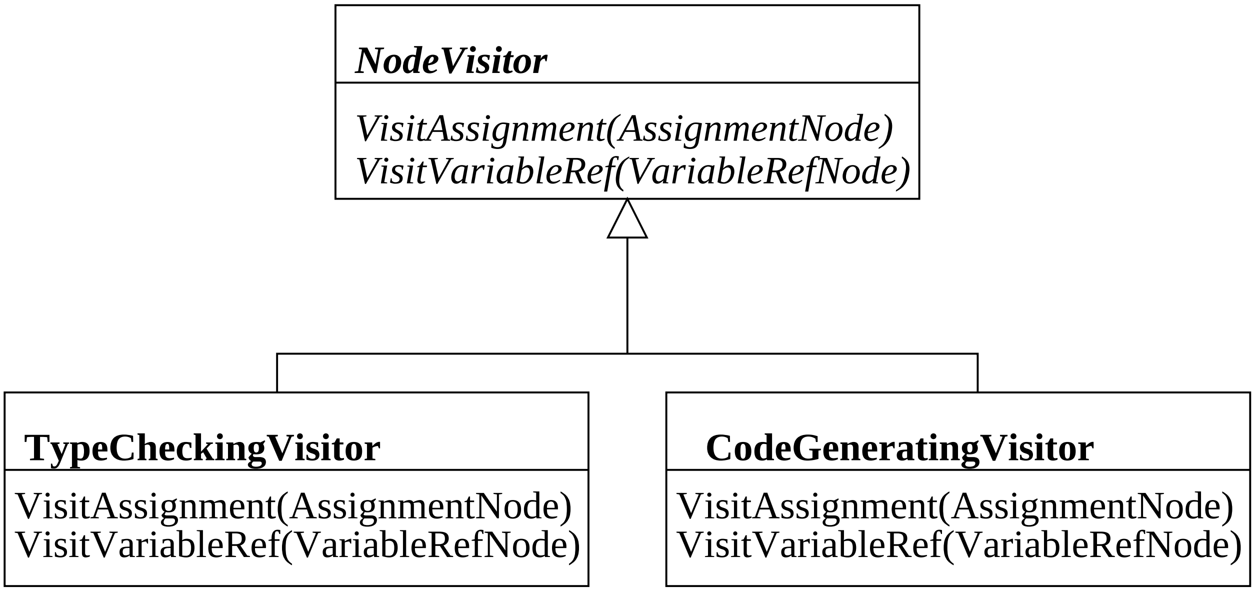
VariableRefNode

TypeCheck()
GenerateCode()
PrettyPrint()

AssignmentNode

TypeCheck()
GenerateCode()
PrettyPrint()

Class Hierarchy in Visitor



NodeVisitor

VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

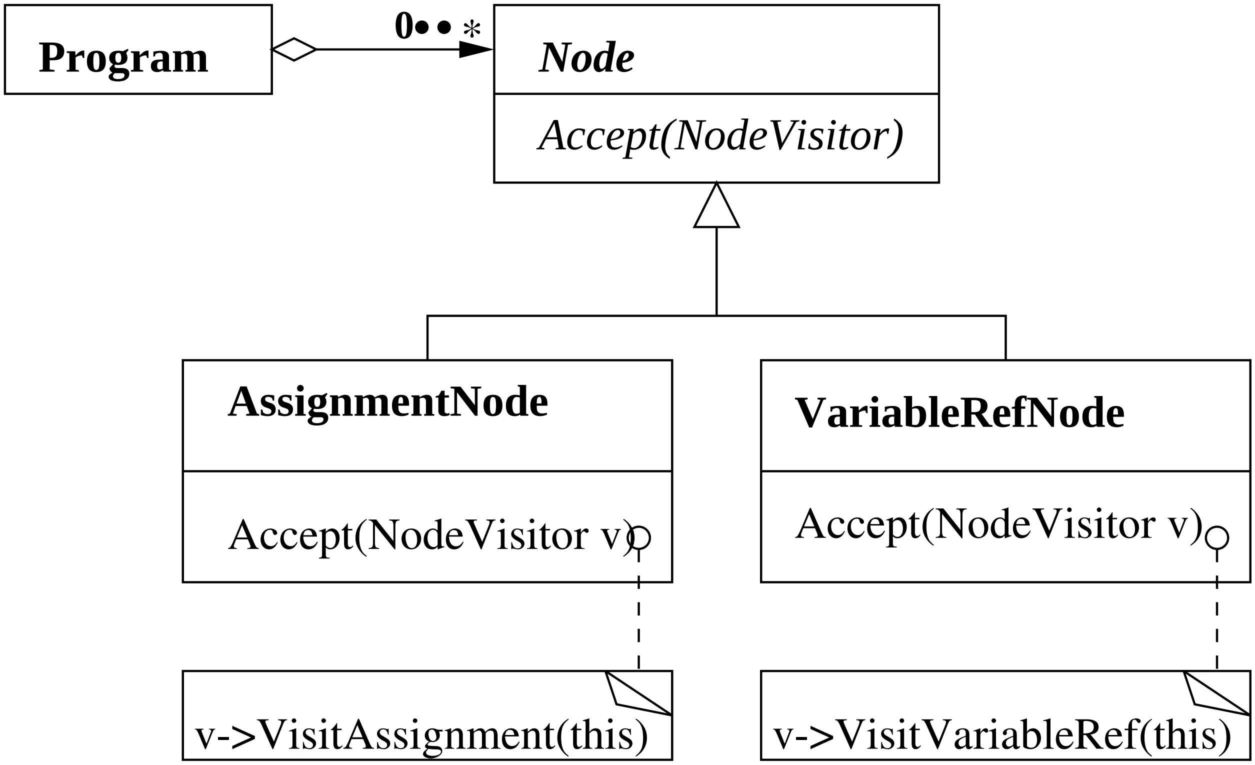
TypeCheckingVisitor

VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

CodeGeneratingVisitor

VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

Structure of Visitor



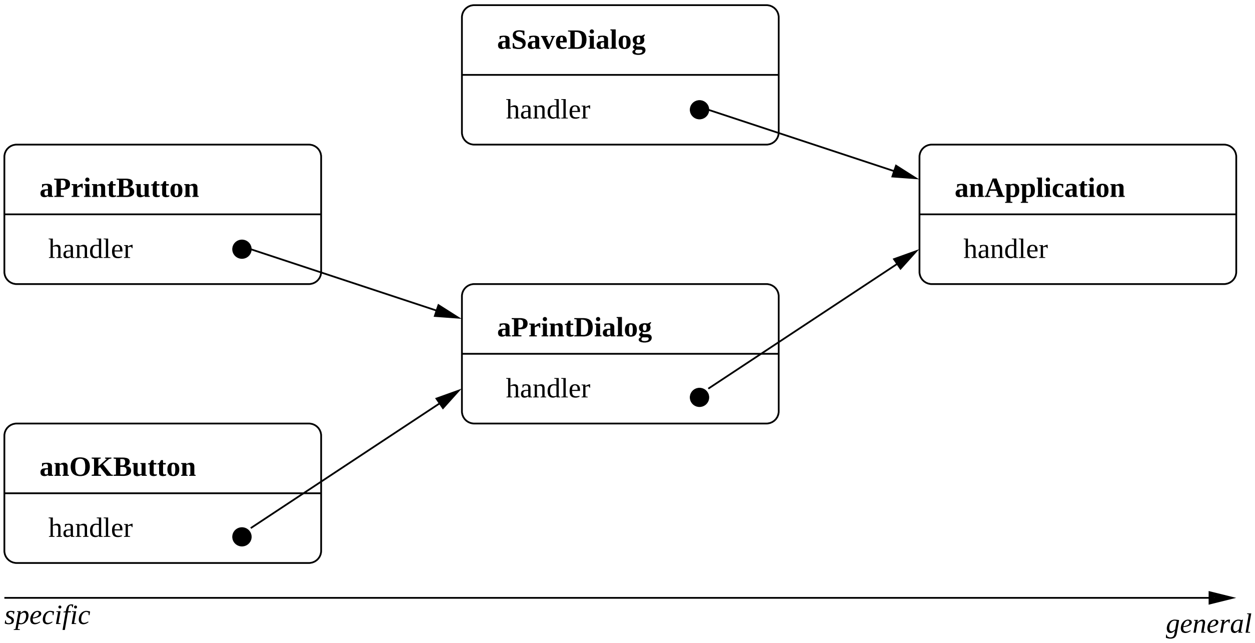
Chain of Responsibility

- **Intent**:
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
  - Chain the receiving objects and pass the request along the chain until an object handles it
- **Motivation**:
  - Context sensitive help facility for GUIs; a help request is handled by one of several user interface objects; the object that provides the help is not known explicitly to the object that initiates the help request.
- **Key Idea**: Decouple senders and receivers by giving multiple objects a chance to handle a request; the request gets passed along a chain of objects until one of them handles it

- **Applicability**: Use this pattern when:
  - more than one object may handle a request, and the handler is not known apriori. The handler should be ascertained automatically
  - you wish 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
- **Participants**:
  - **Handler**: defines an interface for handling requests and possibly implements the successor link
  - **Client**: initiates the request to a ConcreteHandler object on the chain
  - **ConcreteHandler**:
    - 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 its successor

- **Collaborations**:
  - When a client issues a request, the request propagates along the chain until a ConcreteHandler object handles it
- **Consequences**:
  - leads to reduced coupling and simplifies object interconnections.
    - An object in the chain only has to keep a single reference to its successor
    - Receiver and Sender have no explicit knowledge of each other
  - increased flexibility in assigning responsibilities to objects
  - receipt is not guaranteed

Motivational Example for COR



aSaveDialog

handler

aPrintButton

handler

anApplication

handler

aPrintDialog

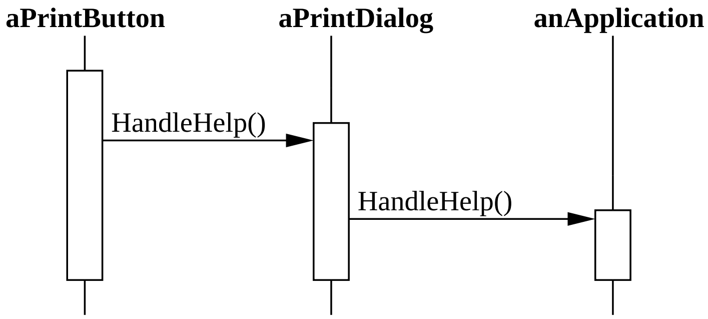
handler

anOKButton

handler

specific

general



aPrintButton

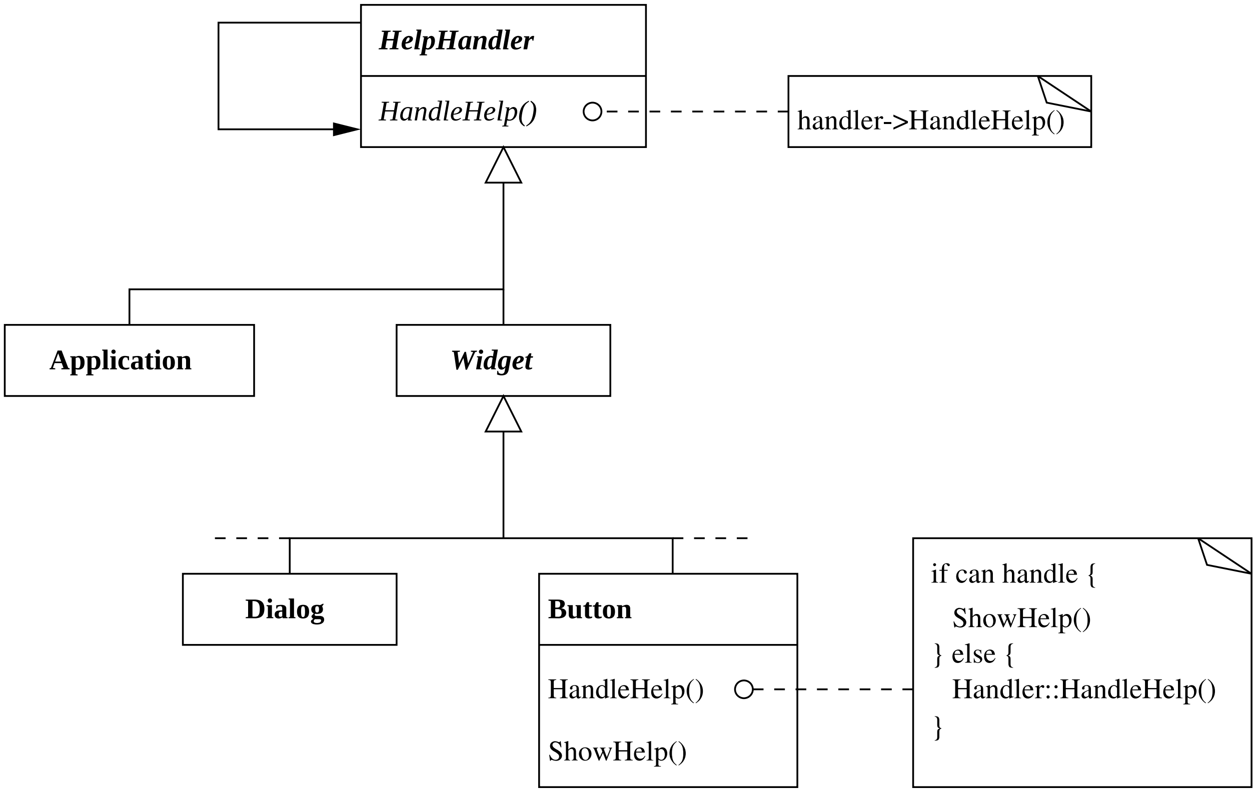
aPrintDialog

anApplication

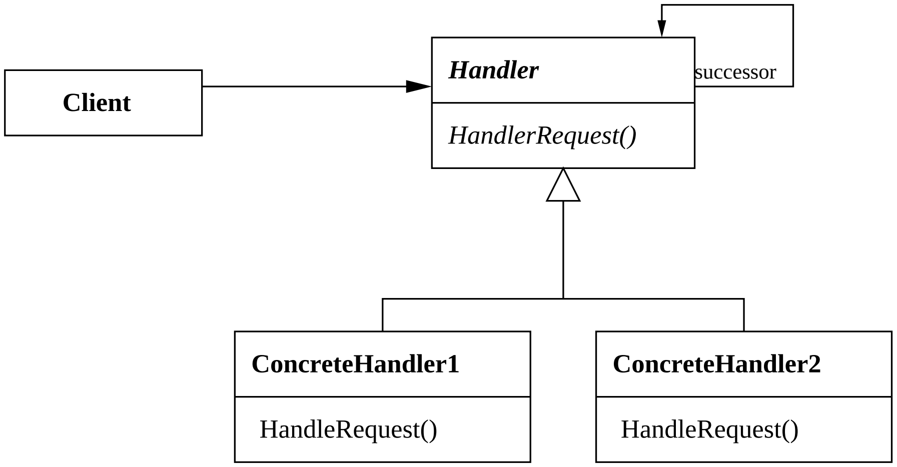
HandleHelp()

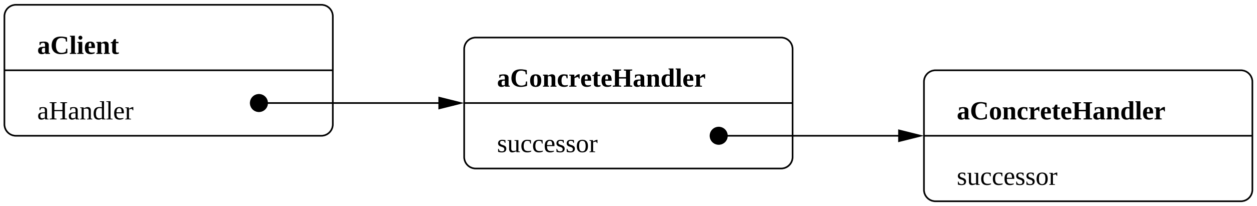
HandleHelp()

Chain of Responsibility



Chain of Responsibility





Interpreter Pattern

- **Intent**: Given the language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
- **Motivation**:
  - If a problem occurs often enough, it is worthwhile to express instances of the problem as sentences in a simple language.
  - An example: searching for strings that match a pattern (regular expressions)
- **Applicability**: Use this pattern when
  - there is a language to interpret and you can represent statements in the language as abstract syntax trees
  - The pattern works best when:
    - the grammar is simple (class hierarchy becomes unmanageable otherwise)
    - efficiency is not a critical concern

- **Participants**:
  1. **Abstract Expression**:
     - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree
  2. **Terminal Expression**:
     - implements an Interpret operation associated with terminal symbols in the grammar
     - an instance is required for every terminal symbol in a sentence
  3. **NonterminalExpression**
     - one such class is required for every rule:

       $$R ::= R_1 R_2 \ldots R_n$$

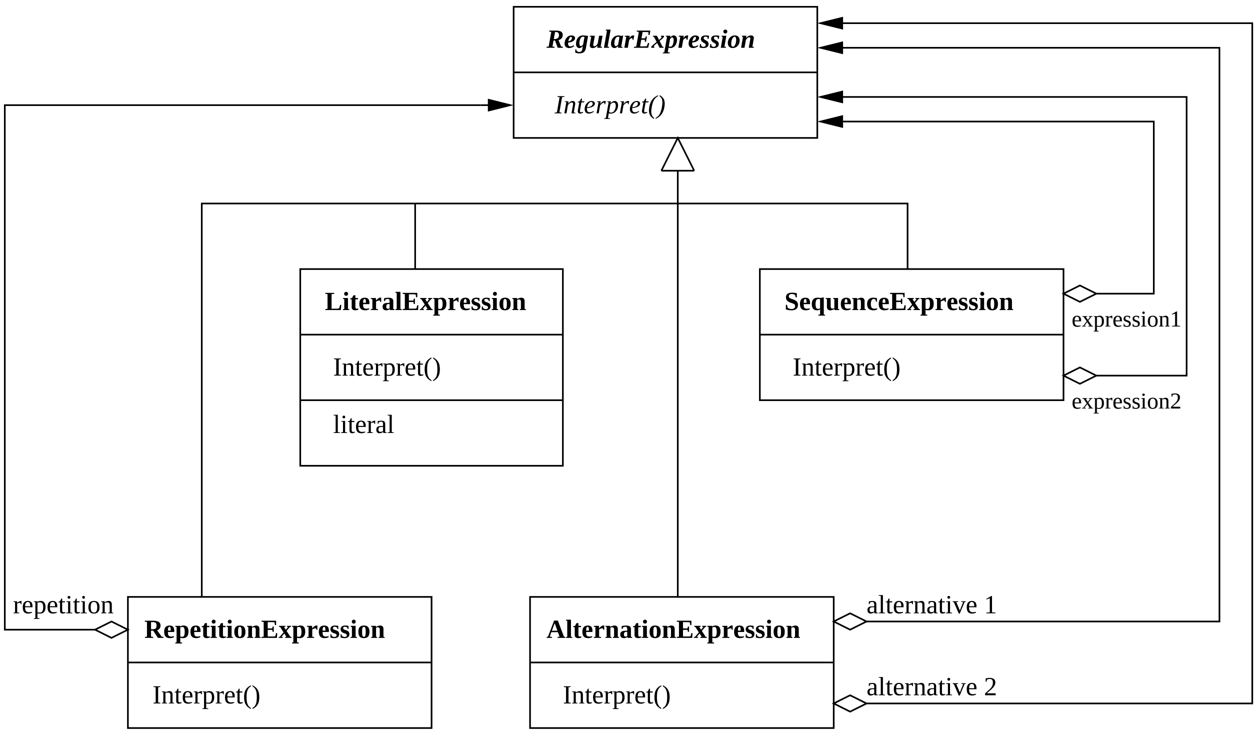
       in the grammar
     - maintains instance variables of type AbstractExpresion for each of the symbols R_1 through R_n
     - implements an Interpret operation for nonterminal symbols in the grammar

4. **Context** contains information that is global to the interpreter
5. **Client**
    - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines
    - invokes interpret operation

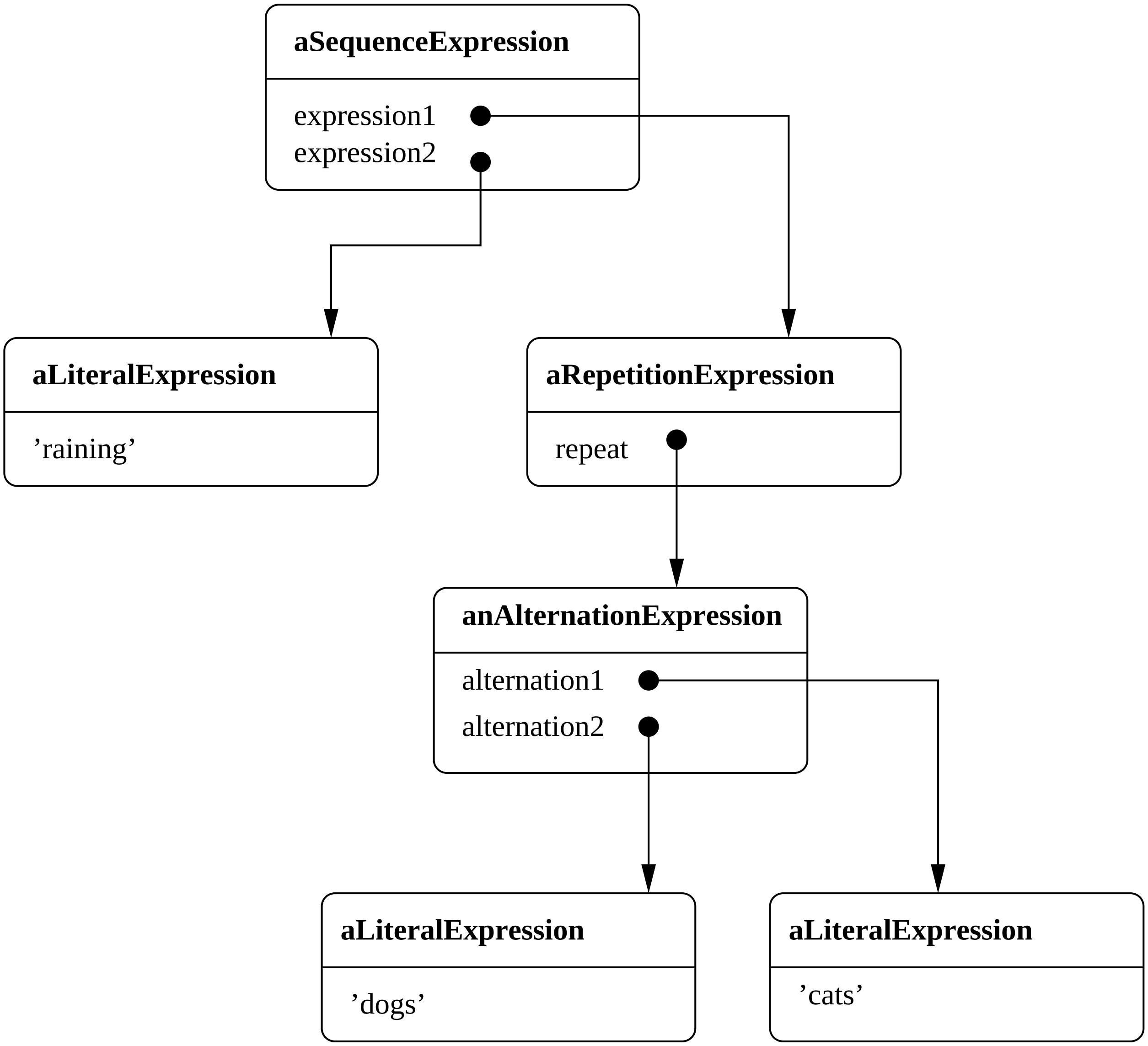
- **Collaborations**:
    - The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances; the client initializes the context and invokes the Interpret operation
    - Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression
    - The Interpret operation of each TerminalExpression defines the base case in the recursion

- **Consequences**:
  - + it is easy to change and extend the grammar (using inheritance)
  - + makes it easy to implement the grammar
  - + makes it easier to evaluate an expression in a new or different way
  - – complex grammars are hard to manage and maintain
- **Related Patterns**
  - **Composite**: The abstract syntax tree is an instance of the Composite pattern
  - **Flyweight**: shows how to share terminal symbols within the abstract syntax tree
  - **Visitor**:can be used to maintain the behavior in each node in the abstract syntax tree in one class
  - **Iterator**: can be used to traverse the structure

Motivational Example for Interpreter



Interpreter Pattern



aSequenceExpression

expression1

expression2

aLiteralExpression

’raining’

aRepetitionExpression

repeat

anAlternationExpression

alternation1

alternation2

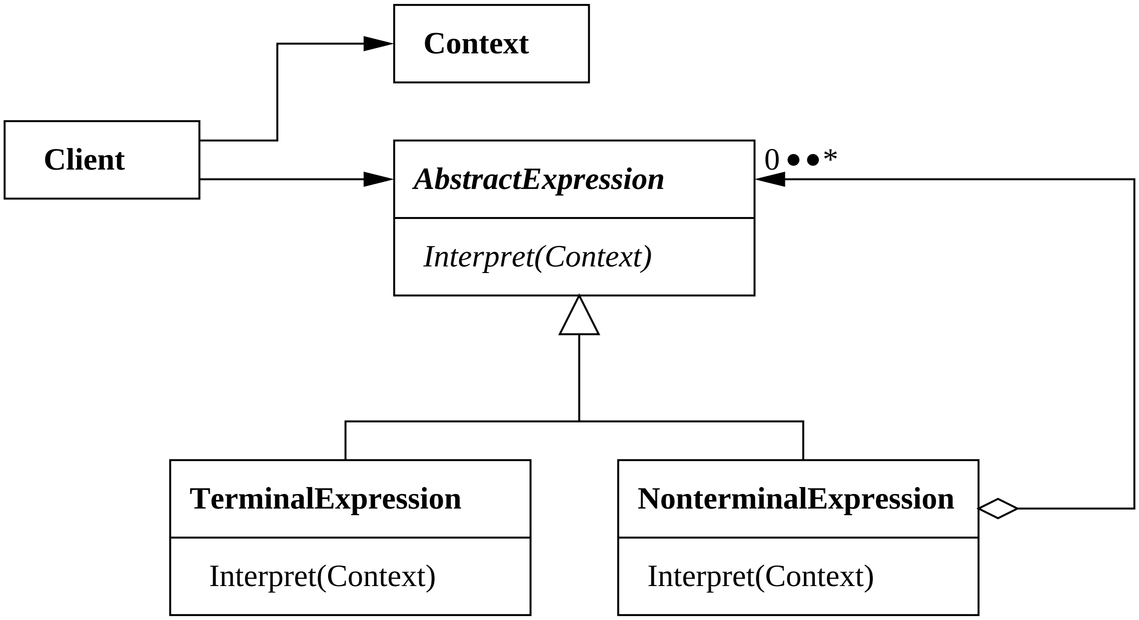
aLiteralExpression

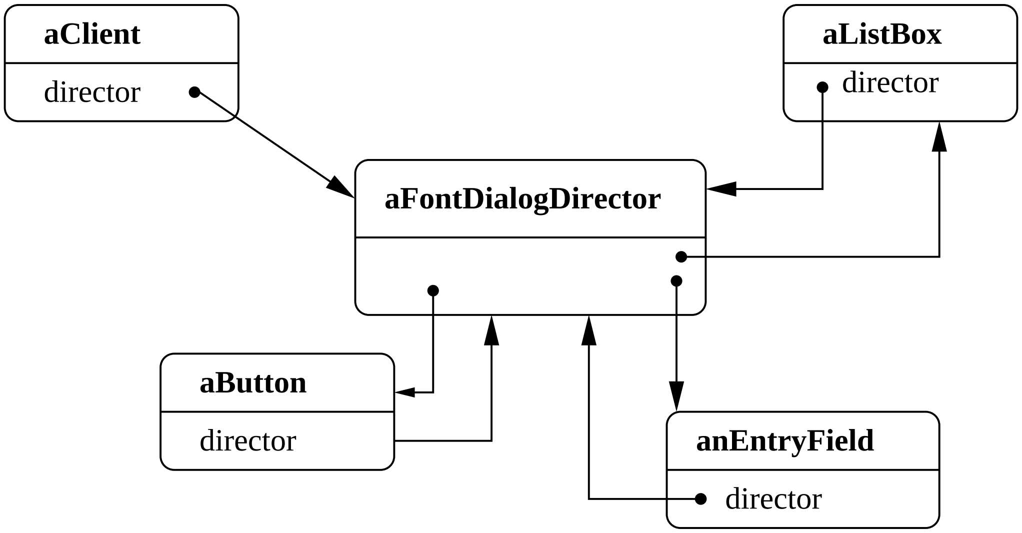
’dogs’

aLiteralExpression

’cats’

Interpreter Pattern





Mediator Pattern

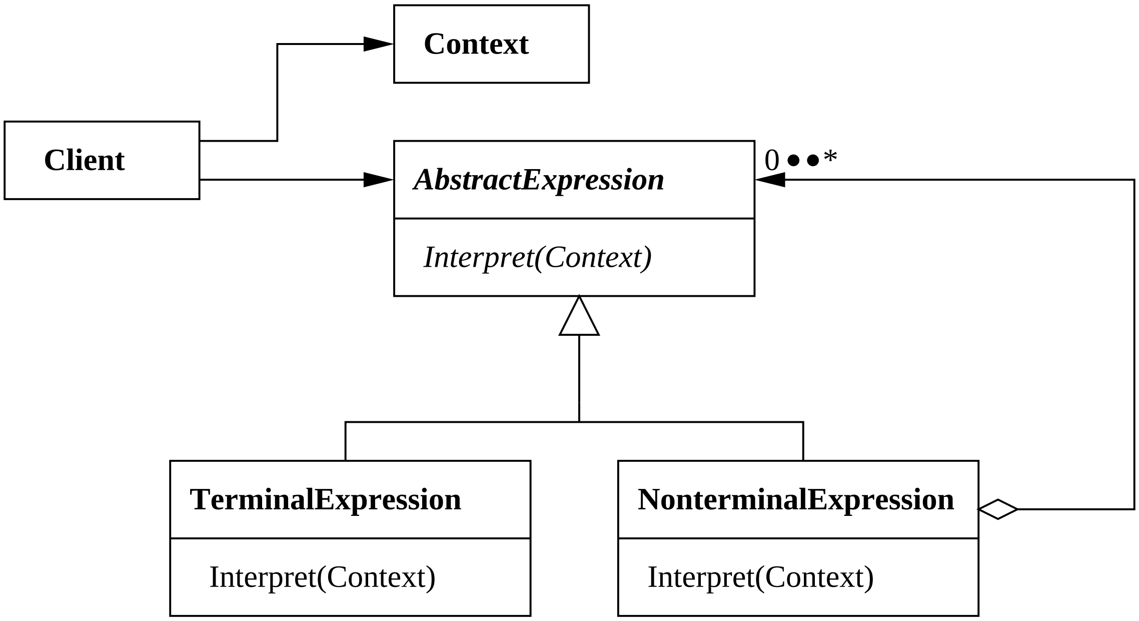
- **Intent**:
  - defines an object that encapsulates how a set of objects interact.
  - promotes loose coupling by keeping objects from referring to each other explicitly
  - consequently enables to vary interaction of objects independently
  - **Motivation**:
    * Object oriented design works best when there is distribution of responsibility among objects, and fair but not too much interaction among these objects
  - **Applicability**: Use this pattern when
    * a set of objects communicate in well defined but complex ways
    * reusing an object is difficult because it refers to and communicates with many other objects

    * a behavior that is distributed between several classes should be customizable without a lot of subclassing

- **Participants**:
  1. **Mediator**: defines an interface for communicating with Colleague objects
  2. **Concrete Mediator**:
     - implements cooperative behavior by coordinating Colleague objects
     - knows and maintains its colleagues
  3. **Colleague classes**
     - 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
  - Mediator implements the cooperative behavior by routing requests between the appropriate colleague(s)
- **Consequences**:
  - + It limits subclassing by localizing behavior
  - + It decouples colleagues
  - + It simplifies object protocols
  - + It abstracts how objects cooperate
  - – It centralizes control and could become a monolith that is hard to maintain
- **Related Patterns**
  - **Facade**: abstracts a subsystem of objects to provide a more convenient interface
  - **Observer**: colleagues can communicate with the mediator using the observer pattern

Motivational Example for Mediator



Context

Client

AbstractExpression

Interpret(Context)

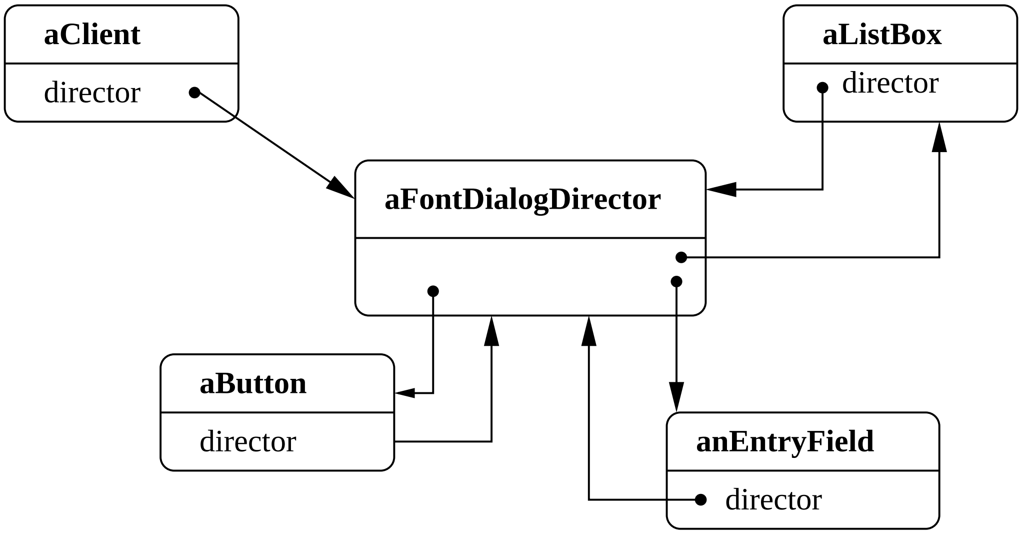
0 ●●*

TerminalExpression

Interpret(Context)

NonterminalExpression

Interpret(Context)



aClient

director

aListBox

director

aFontDialogDirector

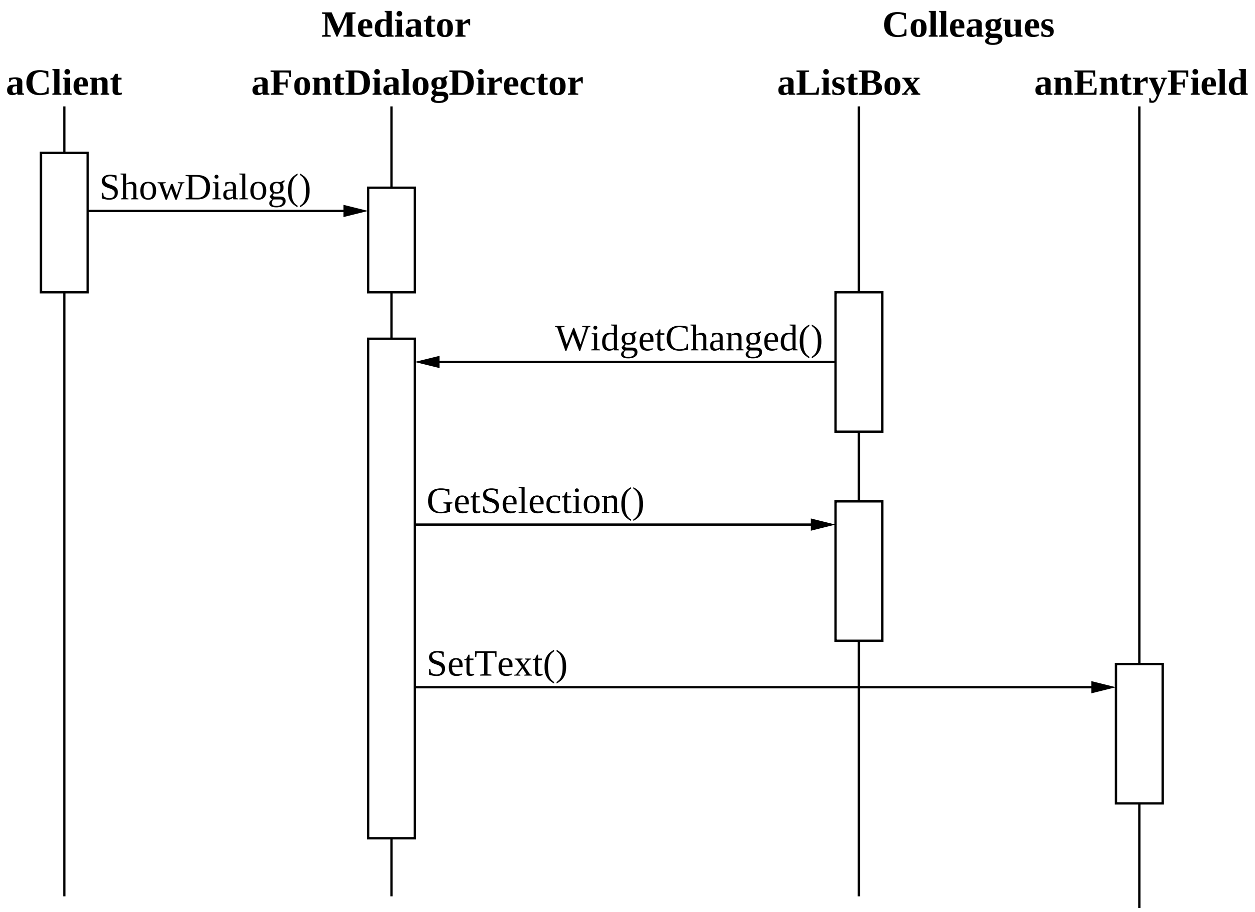
aButton

director

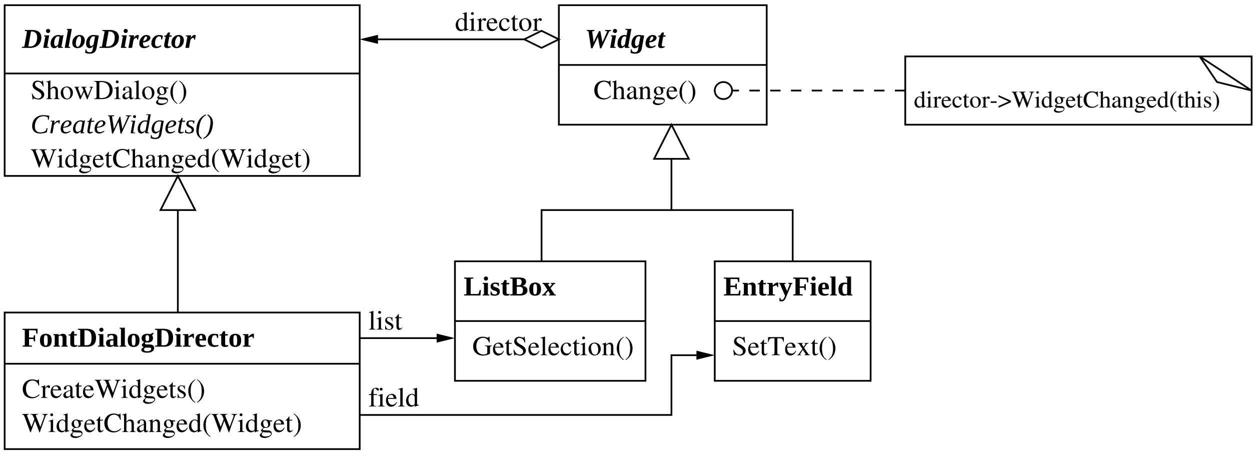
anEntryField

director

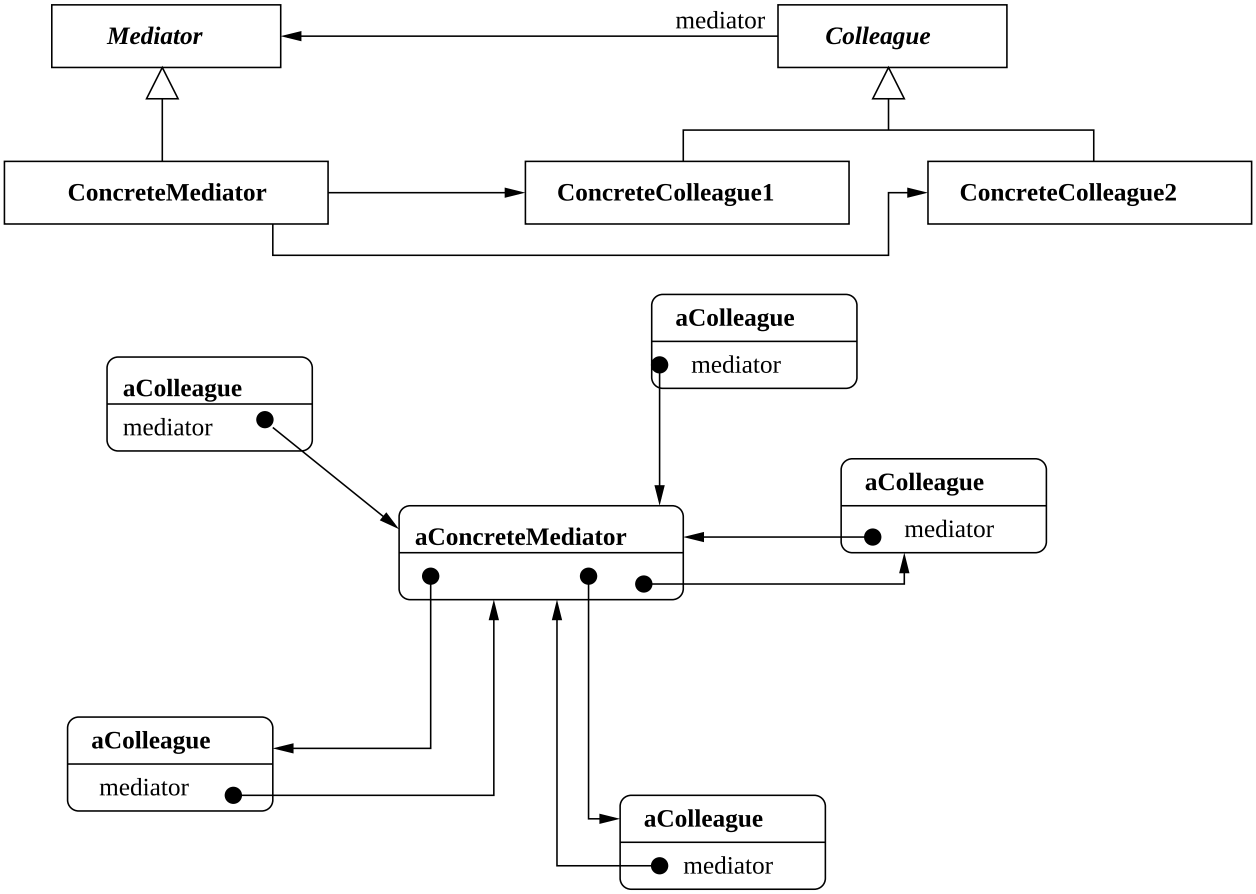
Mediator Pattern



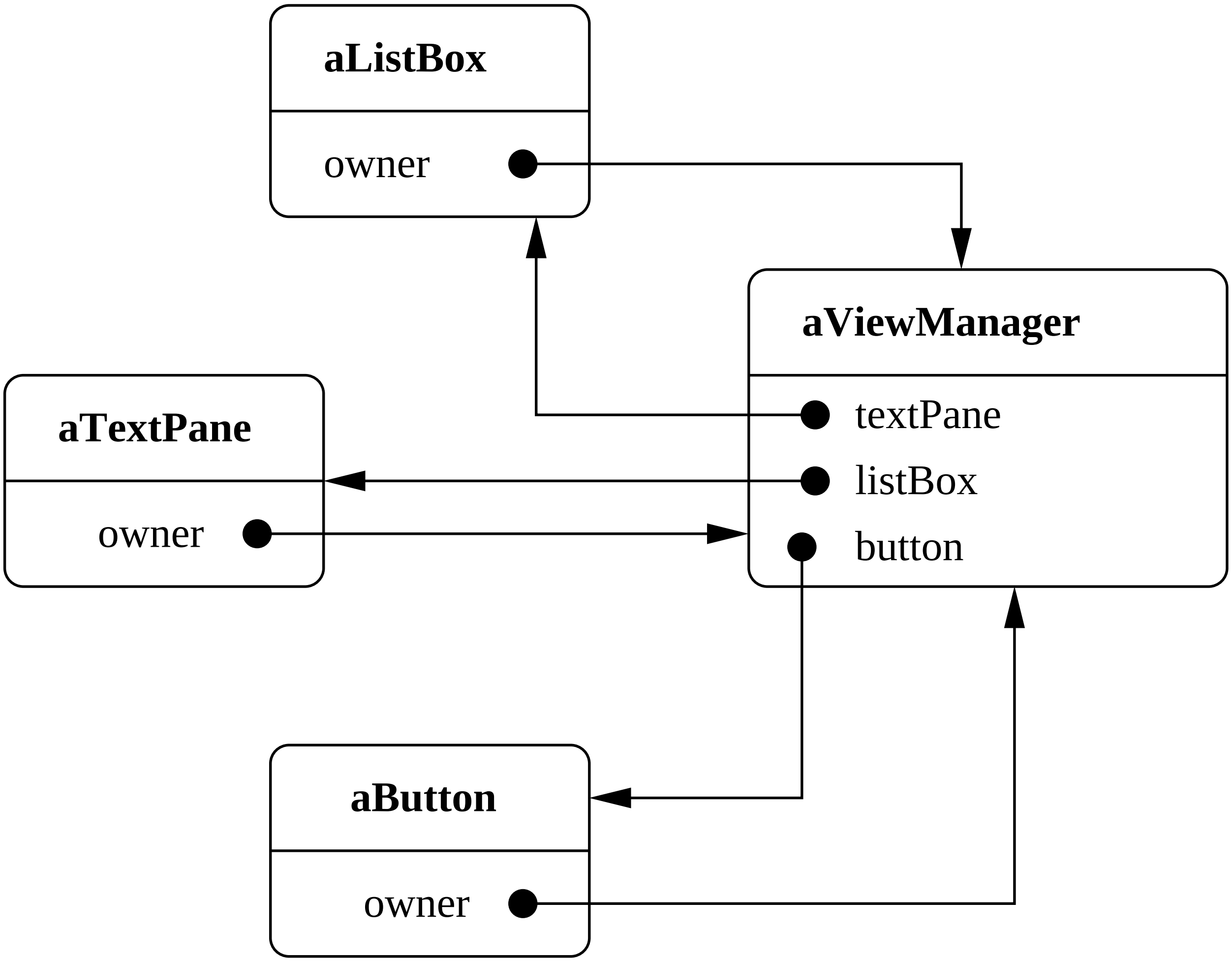
Mediator Pattern



Mediator Pattern



Mediator Pattern



Memento Pattern

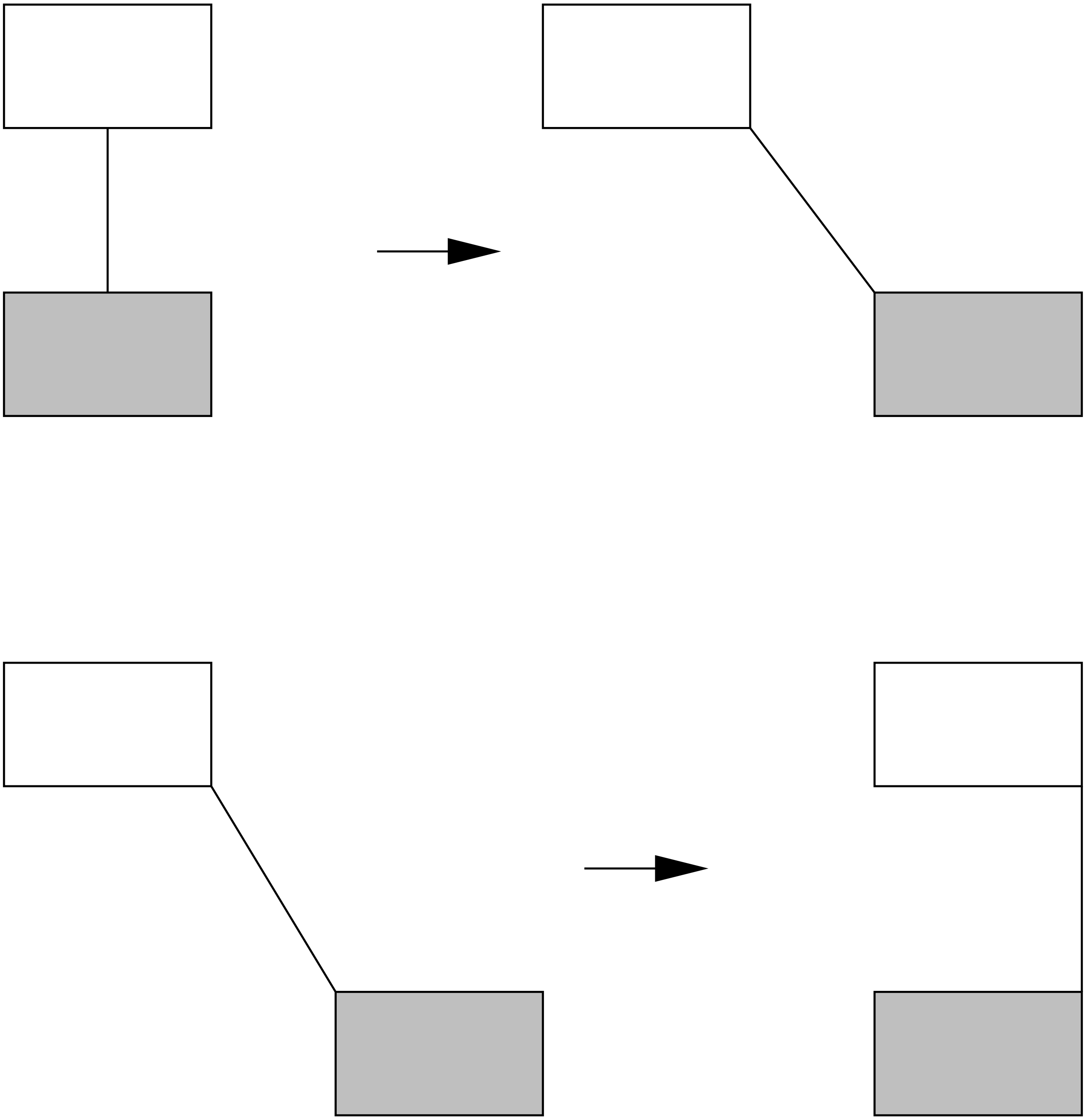
- **Intent**: Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.
- **Motivation**:
  - Often, it is necessary to record the internal state of an object
    - ∗ implementing checkpoints
    - ∗ implementing undo mechanisms
    - ∗ implement error recovery
- **Applicability**: Use this pattern when
  - 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

- **Participants**:
  1. **Memento**:
     - stores internal state of the Originator object; what the memento stores is at its originator's discretion
     - protects against access by objects rather than the originator;
       * Caretaker sees a narrow interface to the Memento; it can only pass the memento to other objects
       * Originator sees a wide interface to the Memento
  2. **Originator**:
     - creates a memento containing a snapshot of its current internal state
     - uses the memento to restore its internal state
  3. **Caretaker**
     - is responsible for the memento's safekeeping
     - never operates on or examines the contents of a memento

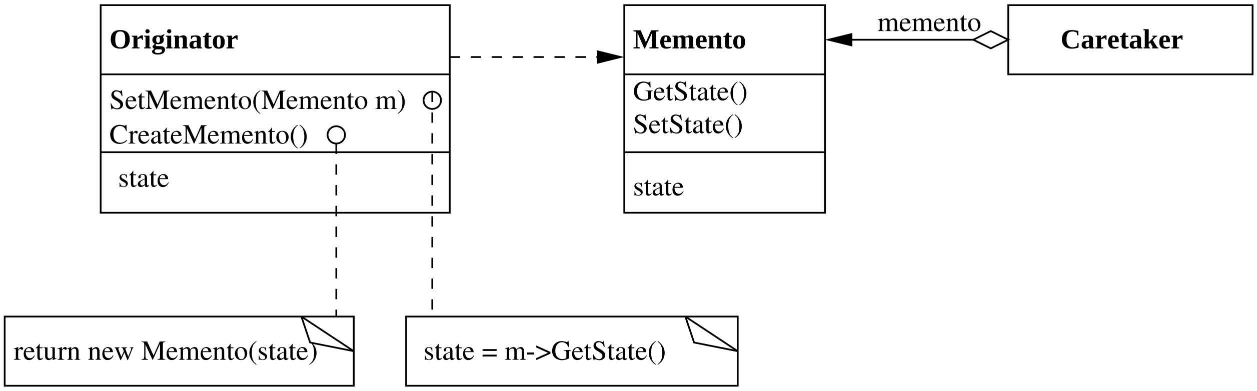
- **Collaborations**:
  - A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.
  - Sometimes the caretaker won't pass the memento back to the originator since the originator might never need to revert to an earlier state
  - Mementos are passive; only the originator that created a memento will assign or retrieve its state
- **Consequences**:
  - + Preserves encapsulation boundaries
  - + Simplifies Originator
  - – Using mementos might be expensive
  - – It may be difficult to ensure that only the Originator can access the Memento's state

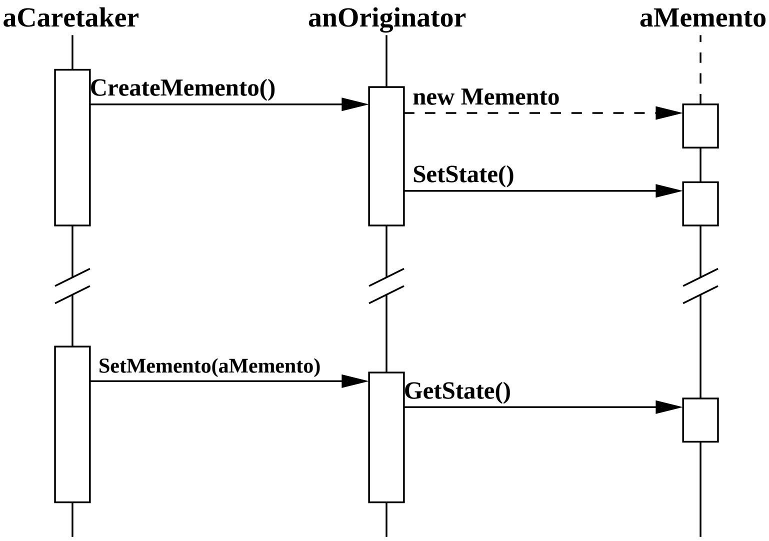
- **Related Patterns**
  - **Command**: Commands can use mementos to maintain state for undoable operations
  - **Iterator**: Mementos can be used for iteration.

Motivational Example for Memento



Memento Pattern





Observer (Publish-Subscribe) Pattern

- **Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Motivation**:
  - Often, when a system is partitioned into a collection of cooperating classes, there is need to maintain consistency between related objects without making them tightly coupled (which reduces their reusability)
    - ∗ implementing checkpoints
    - ∗ implementing undo mechanisms
    - ∗ implement error recovery

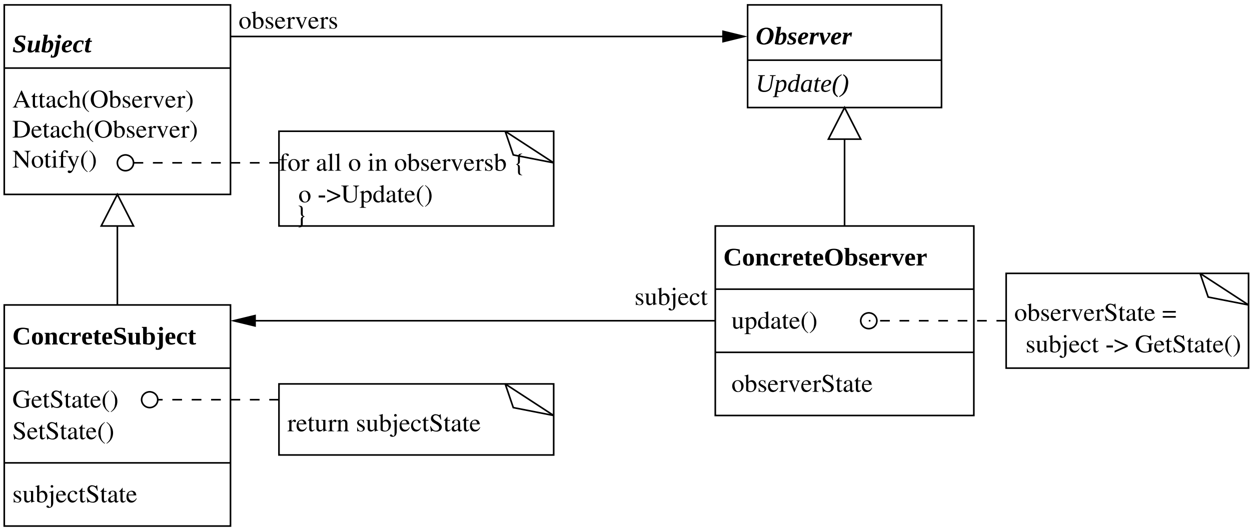
- **Applicability**: Use this pattern in any of the following situations:
  - when an abstraction has two aspects, one dependent on the other (encapsulating these lets you vary and reuse them independently)
  - when a change to one object requires changing others, and you do not know how many objects need to be changed
  - when an object should be able to notify other objects without making assumptions about who these objects are (i.e. you do not want these objects tightly coupled)

- **Participants**:
  1. **Subject**:
     - knows its observers. Any number of Observer objects may observe a subject.
     - provides an interface for attaching and detaching observer objects
  2. **Observer**:
     - defines an updating interface for objects that should be notified of changes in a subject
  3. **ConcreteSubject**
     - stores state of interest to ConcreteObserver objects
     - sends a notification to its observers when its state changes
  4. **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 the above information to reconcile its state with that of the subject
- **Consequences**:
  - + Lets you vary subjects and observers independently
  - + All a subject needs to know is it has observers; it does not need to know the concrete class of any observer
  - – Unexpected and cascades of updates can occur because observers have no knowledge of each other's presence

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

Motivational Example for Observer



Subject

Attach(Observer)
Detach(Observer)
Notify()

observers

Observer

Update()

for all o in observersb {
o ->Update()
}

ConcreteObserver

update()

observerState

observerState =
subject -> GetState()

subject

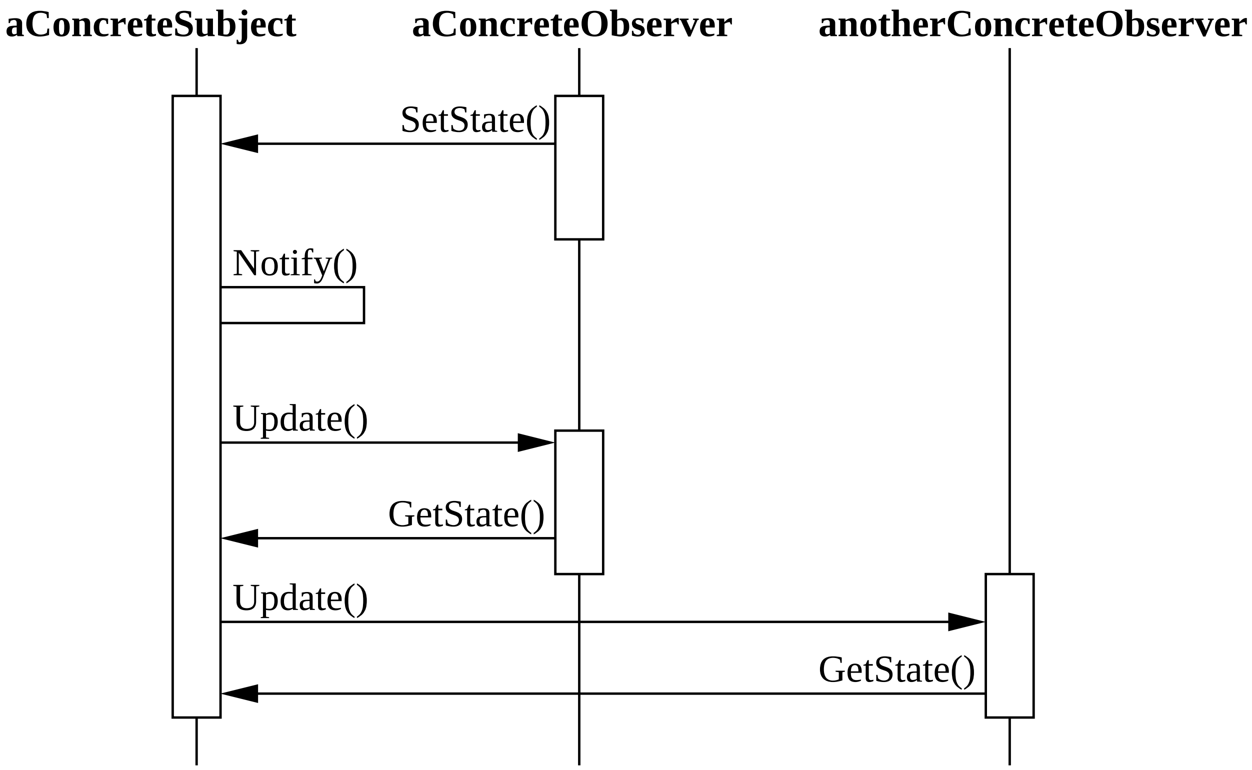
ConcreteSubject

GetState()
SetState()

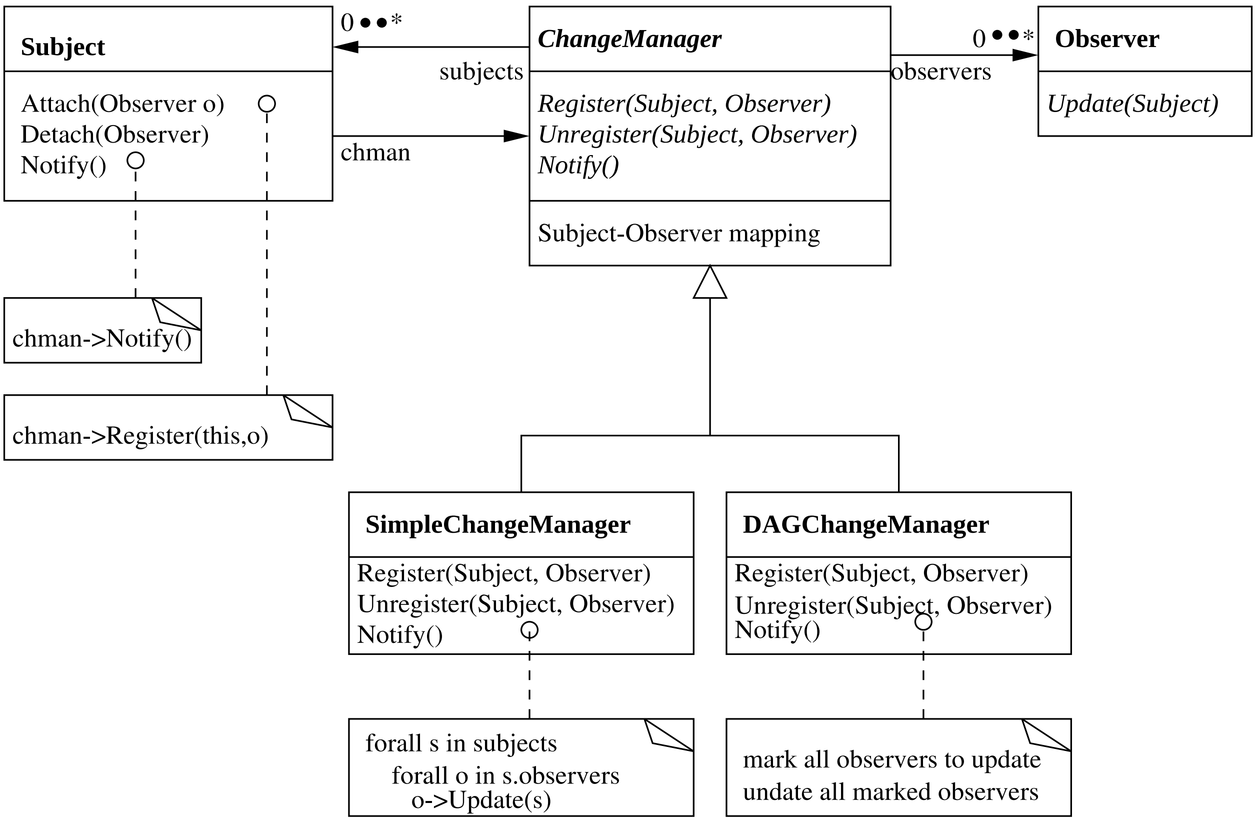
subjectState

return subjectState

Observer Pattern



Observer Pattern



Subject

Attach(Observer o)
Detach(Observer)
Notify()

chman->Notify()

chman->Register(this,o)

0 ••*
subjects

chman

ChangeManager

Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()

Subject-Observer mapping

0 ••*
observers

Observer

Update(Subject)

SimpleChangeManager

Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()

```
forall s in subjects
   forall o in s.observers
     o->Update(s)
```

DAGChangeManager

Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()

mark all observers to update
undate all marked observers

State Pattern

- **Intent**: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class
- **Motivation**:
  - Often, an object is expected to behave differently depending on which state it is currently in
  - An Example: A TCP connection object can be in three different states: Established, Listening, and Closed. Its response to object requests (such as an *Open* request will depend on its current state

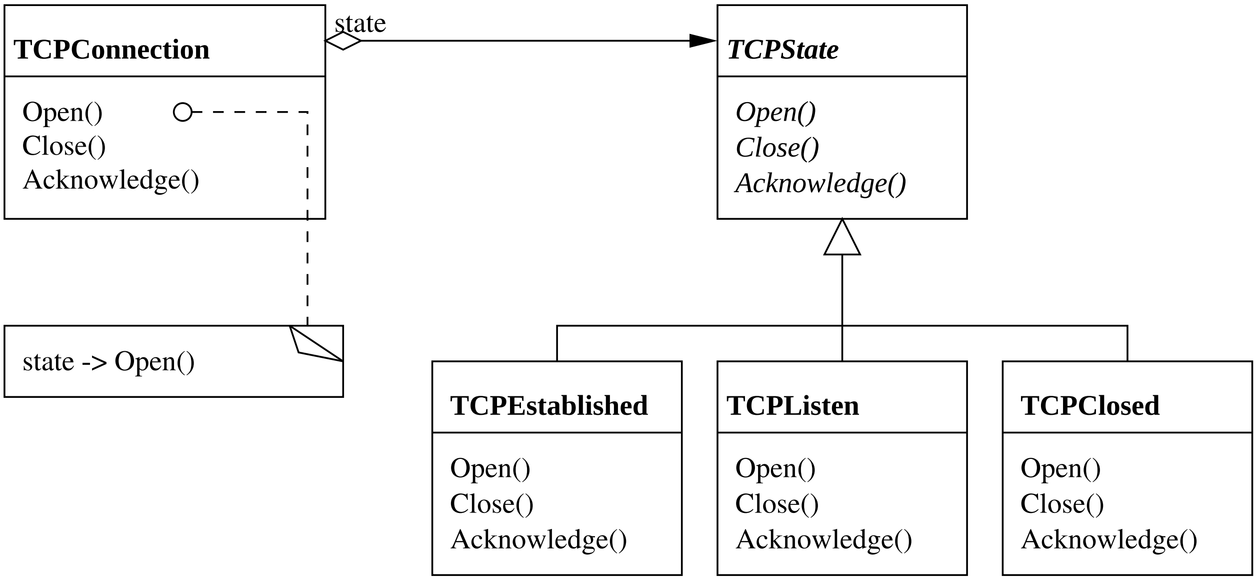
- **Applicability**: Use this pattern in either of the following cases:
  - an object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - operations have large, multipart conditional statements that depend on the object's state;
    - the State pattern will put 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

- **Participants**:
  1. **Context**:
     - defines the interfaces of interest to clients
     - maintains an instance of a ConcreteState subclass that defines the current state
  2. **State**:
     - defines an interface for encapsulating the behavior associated with a particular state of the Context
  3. **ConcreteState subclasses**
     - each subclass implements a behavior associated with a state of the Context
- **Collaborations**:
  - Context delegates state-specific requests to the ConcreteState object
  - A context may pass itself as an argument to the State object handling the request, letting the State object access the Context if necessary

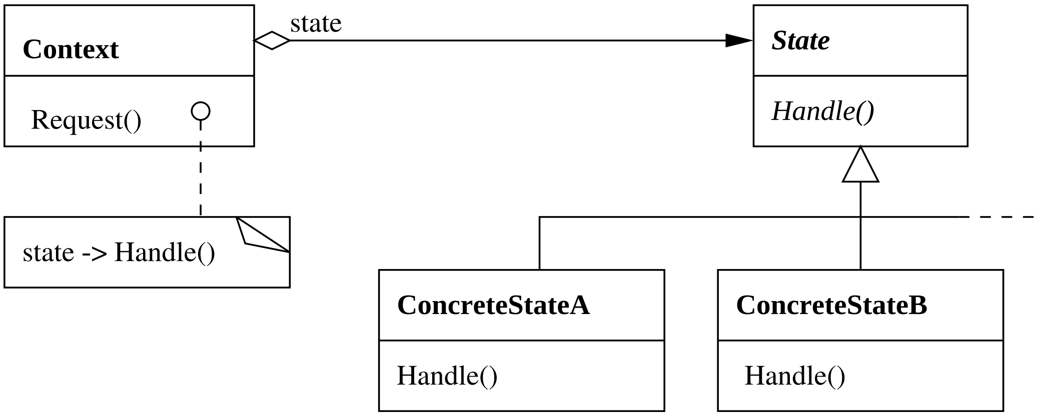
- Context is the primary interface for the clients and clients can configure a context with State objects; once a context is configured, its clients don't have to deal with the State objects directly
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances

- **Consequences**:
  - + Localizes state-specific behavior and partitions behavior for different states
  - + It makes state transitions explicit (inconsistent internal states can be avoided)
  - + State objects can be shared (flyweights)
- **Related Patterns**
  - **Flyweight**: The Flyweight pattern explains when and how State objects can be shared
  - **Singleton**: State objects are often singletons

Motivational Example for State Pattern



State Pattern



Context

Request()

state

State

Handle()

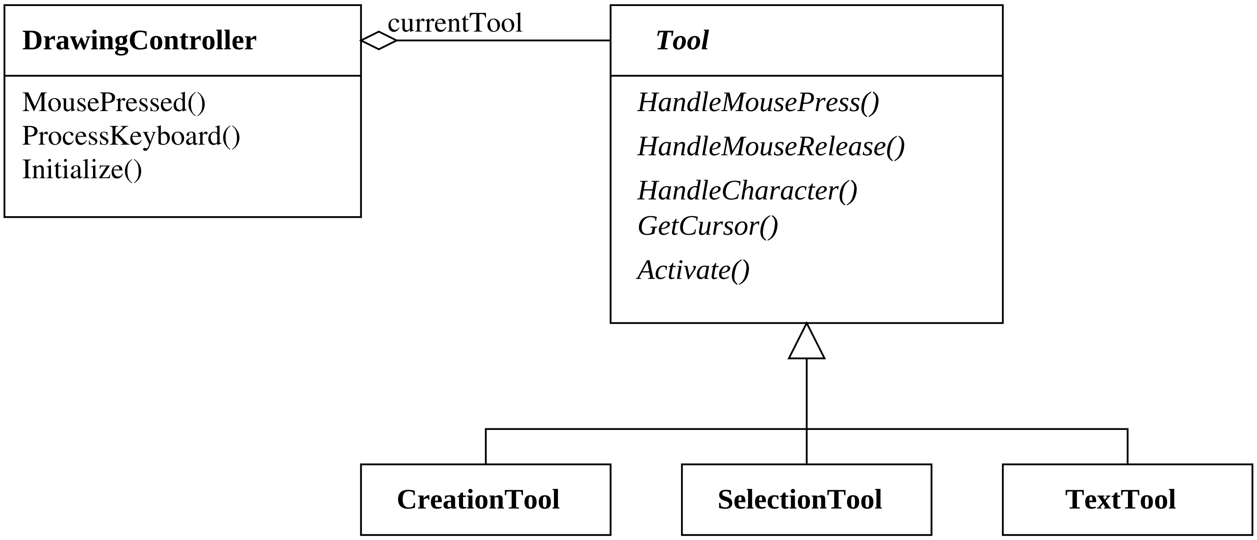
state -> Handle()

ConcreteStateA

Handle()

ConcreteStateB

Handle()



DrawingController

MousePressed()
ProcessKeyboard()
Initialize()

currentTool

Tool

HandleMousePress()
HandleMouseRelease()
HandleCharacter()
GetCursor()
Activate()

CreationTool

SelectionTool

TextTool

Template Pattern

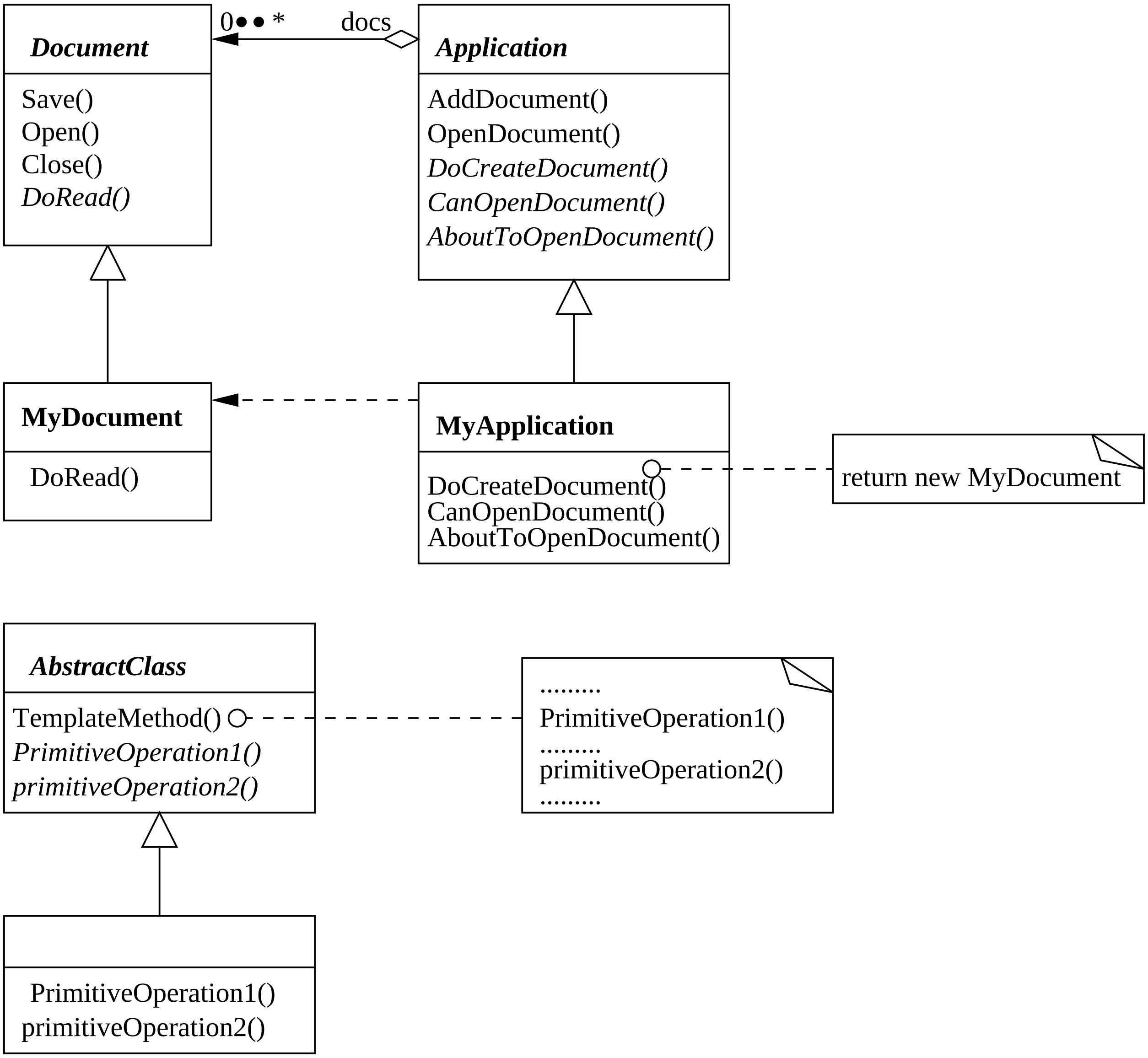
- **Intent**:
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- **Motivation**:
  - Often, there is a need to define an algorithm in terms of abstract operations that subclasses override to provide behavior
- **Applicability**: Use this pattern
  - to implement the invariant parts of an algorithm 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 (a case of *refactoring to generalize*)
- to control subclasses extensions; once can define a template method that calls **hook** operations at specific points, thereby permitting extensions only at those points

- **Participants**:
  1. **AbstractClass**:
     - defines the abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
     - implements a template method defining the skeleton of an algorithm.
  2. **ConcreteClass**:
     - implements the primitive operations to carry out subclass-specific steps of the algorithm

- **Collaborations**:
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm
- **Consequences**:
  - + Template methods are a fundamental technique for code reuse, particularly important in class libraries (because they help factor out common behavior in library classes).
  - + Leads to an *inverted control* structure whereby a parent class calls the operations of a subclass and not the other way round.
- **Related Patterns**
  - **Factory Method**: Template methods often call factory methods
  - **Strategy**: Template methods use inheritance to vary part of an algorithm while Strategies use delegation to vary the entire algorithm

Template Pattern



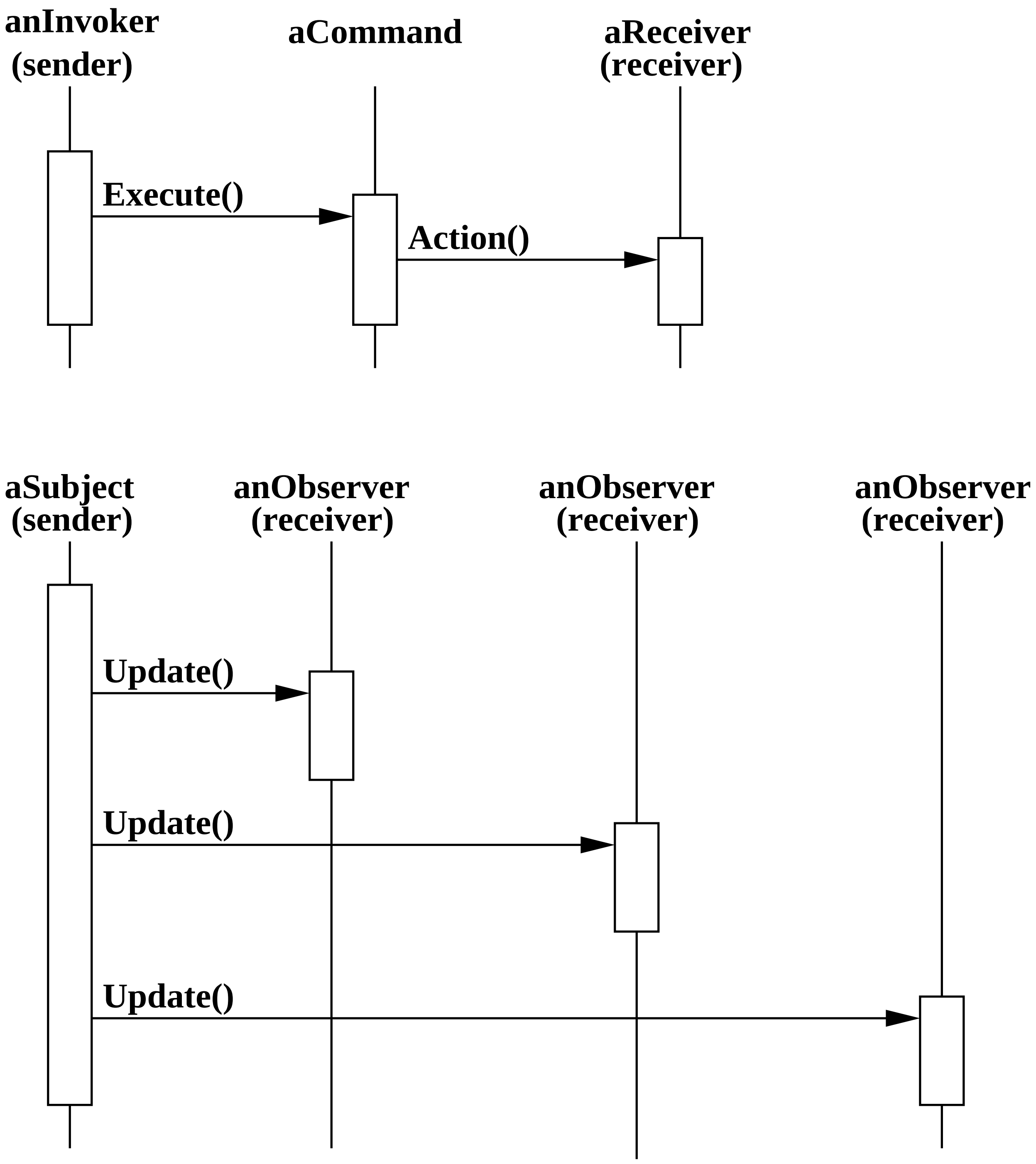
Common Themes in Behavioral Patterns

- **Encapsulating Variation**:
  - A Strategy object encapsulates an algorithm
  - A State object encapsulates a state-dependent behavior
  - A Mediator object encapsulates the protocol between objects
  - An Iterator object encapsulates the way to access and traverse the components of an aggregate object
  - This theme is common to creational and structural patterns also: Abstract Factory, Builder, Prototype, Decorator, Bridge, etc.

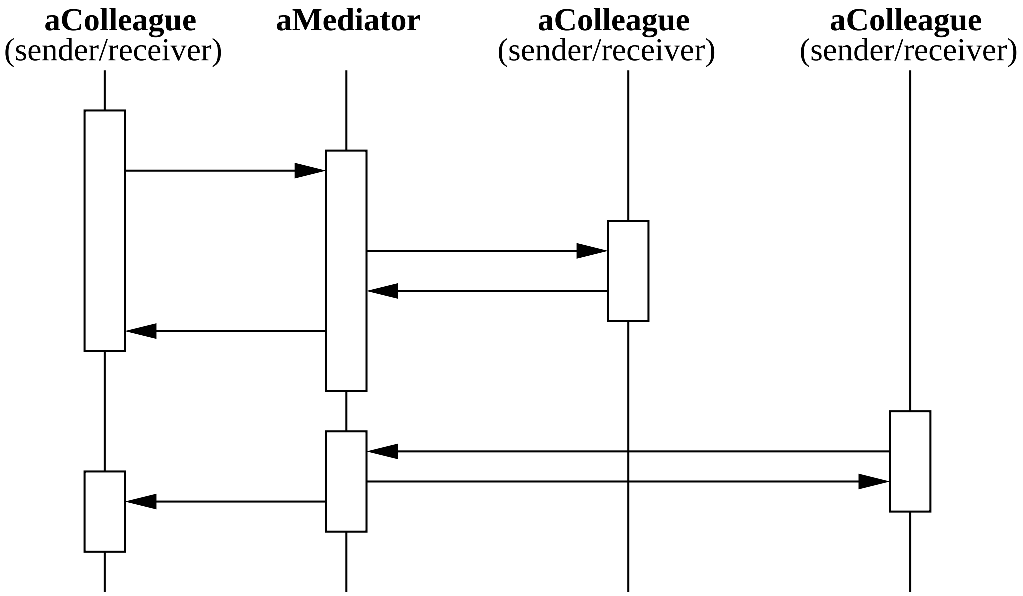
- **Objects as Arguments**:
  - A Visitor object is the argument to a polymorphic Accept operation on the objects it visits
  - Command and Memento define objects that act as magic tokens passed around and invoked at a later time (the token represents a request in Command and internal state in Memento)
  - Typically, clients are not aware of these tokens
- **Handling of Communication**:
  - Observer distributes communication by introducing Observer and Subject objects
  - Mediator object encapsulates the communication between objects
  - Typically, clients are not aware of these tokens

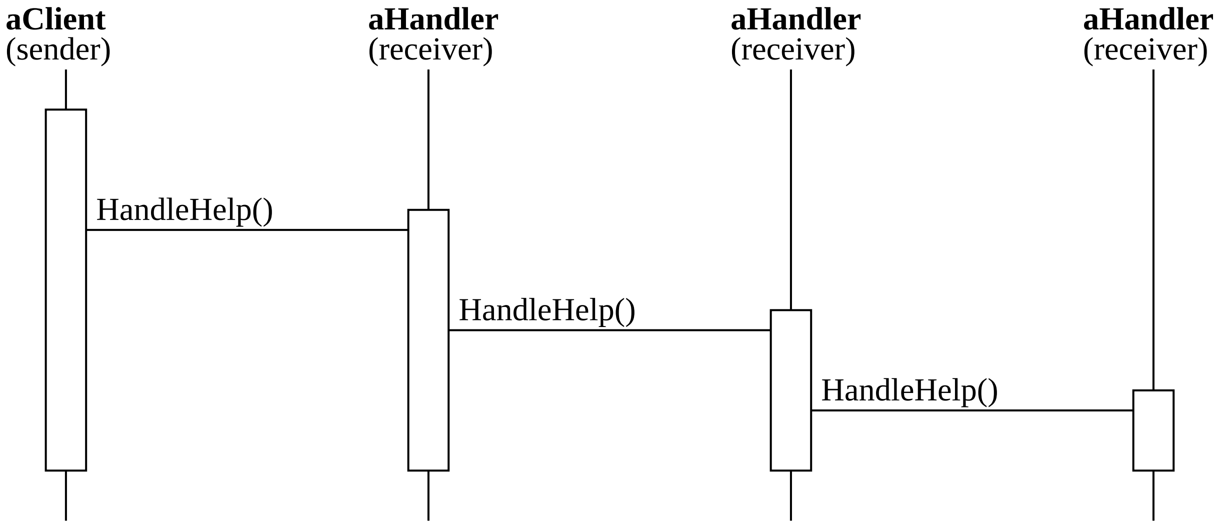
- **Decoupling Senders and Receivers**:
  - When collaborating objects refer to each other directly, they become dependent on each other, which will have an adverse impact on layering and reusability
  - The following patterns decouple senders and receivers: Command, Observer, Mediator, and Chain of Responsibility, all in different ways
  - Command supports decoupling by using a Command object to define the binding between a sender and receiver
  - Observer decouples subjects from observers by defining an interface for signaling changes in subjects
  - Mediator achieves decoupling by having them refer to each other indirectly through a Mediator object
  - Chain of Responsibility decouples the sender from the receiver by passing the request along a chain of potential receivers

Command and Observer



Mediator and Chain of Responsibility





Impact of Design Patterns

- Enable best design practices to be used commonly by less experienced software engineers
- Provide a common vocabulary for designers to communicate, explore, and discuss design alternatives
- Provide a valuable documentation and learning aid
- Design patterns are invaluable in turning an analysis model into an effective design/implementation model, providing guidance and justification for ”why” of designs
- DPs help determine how to reorganize or refactor a design in an effective way

Case Study of an

Auction House

Y. NARAHARI

Computer Science and Automation
Indian Institute of Science
Bangalore - 560 012

WHAT: Web Based House of Auctions

Requirements Specification

- WHAT is a web server with auction logic and a backend database, and can host a multitude of parallel auctions. It supports:
  - Single item open cry auction
  - Multiple item open cry auction
  - Dutch auctions (usually for multiple items)
- WHAT has facilities for:
  1. Registration of buyers and sellers: This will include some way of authenticating the participants and profiling
  2. Setting up the auction event: this involves describing the item(s) on auction and setting up the auction rules:

   - type of auction
   - negotiable parameters in auction
   - start time of auction
   - auction closing rules
   - penalties for defaulting

3. Conducting the bidding process by:
   - collecting bids from buyers
   - implementing bid control rules
4. Evaluation of bids and determining winner(s) of auction
5. Electronic payment
6. Initiating trade settlement

- WHAT should have support for:
  - defining of ”new” auctioning mechanisms and algorithms
  - deployment of agents
  - security of data and transactions
  - maintaining anonymity of buyers and sellers
  - notification mechanisms to indicate the status and progress of auctions
  - participation in multiple auctions
  - searching through the ongoing auctions for desired information (search engine)
  - an elegant and functional (but need not be fancy) user interface
  - a functional, scalable database

Different Steps in the Process

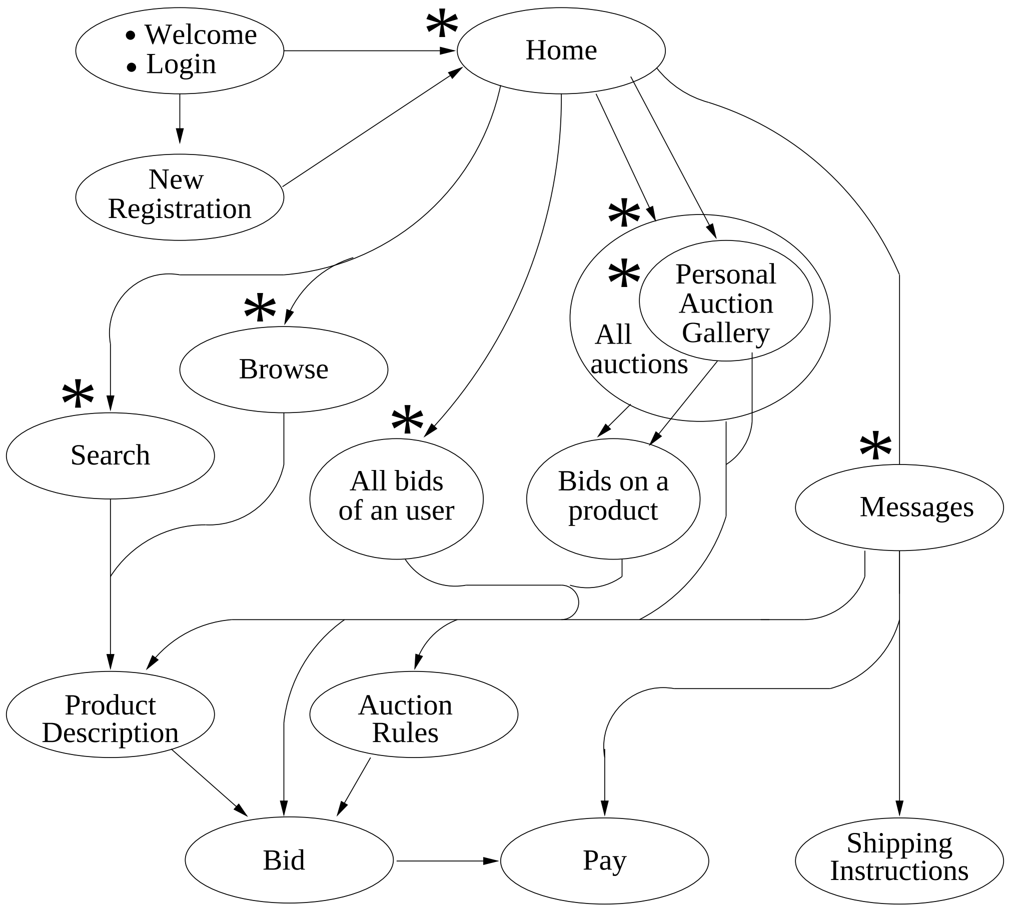
- **Requirements analysis** to come up with all important use cases, their descriptions, and and use-case diagrams.
- **Domain analysis** to discover all important abstractions for the problem domain (classes, interfaces, collaborations, and relationships)
  - Structural: Class and object diagrams
  - Behavioral: sequence, collaboration, activity, state charts
- **Architecture Design**: Identify the major subsystems and the connections and interactions

- **Detailed Design**: This will involve specifying a working solution that can be transformed into programming code Additional issues to be addressed here are: GUI design (a functional and elegant GUI); database design (a functional, scalable database organization supporting persistency and querying). Design patterns are useful here.
- **Implementation and Testing**: Some issues to look for here are - web enabling, concurrency of transactions, etc.

Requirements Analysis

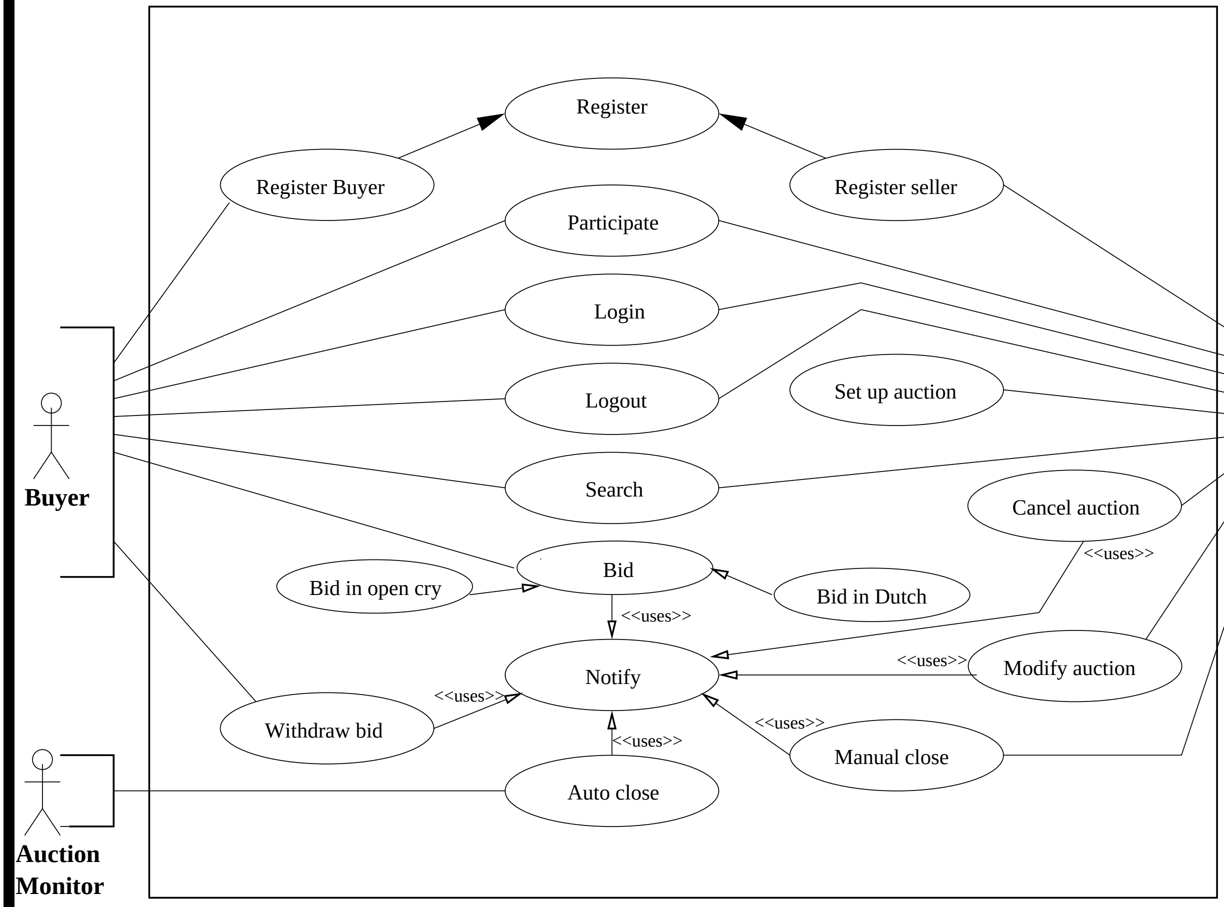
- Actors: Buyers, Sellers, Auctioneers, System Administrators
- Use cases:
  - Search Auction
  - Register Buyer
  - Register Seller
  - Select Auction
  - Place Bid
  - Cancel Bid
  - Withdraw from Auction
  - Add Auction
  - Delete Auction
  - Validate User
  - Cancel Registration
  - Add New Auction Protocol
  - Notify users

Navigation of Website



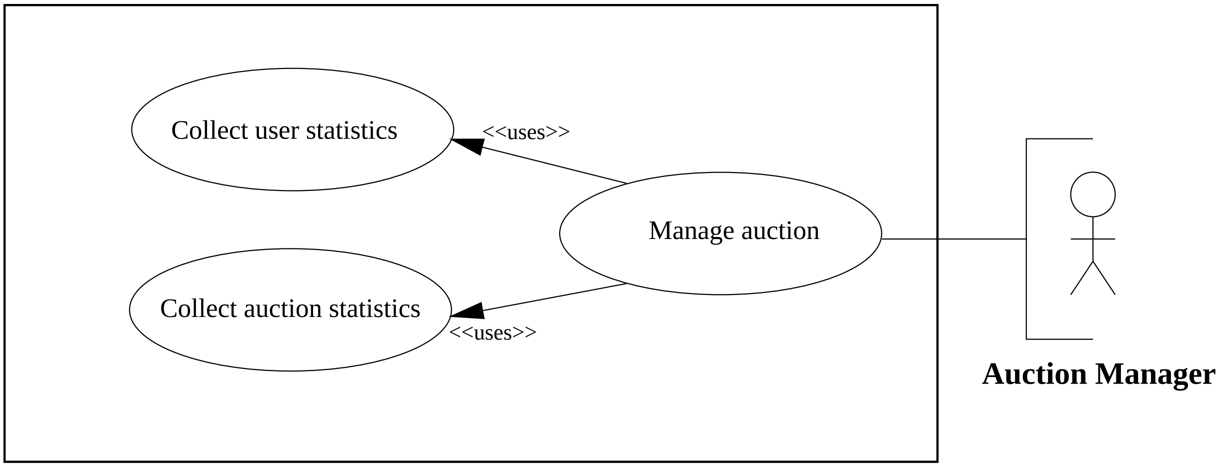
Use Case Diagram

USE CASE DIAGRAM FOR AUCTION



Use case Diagram

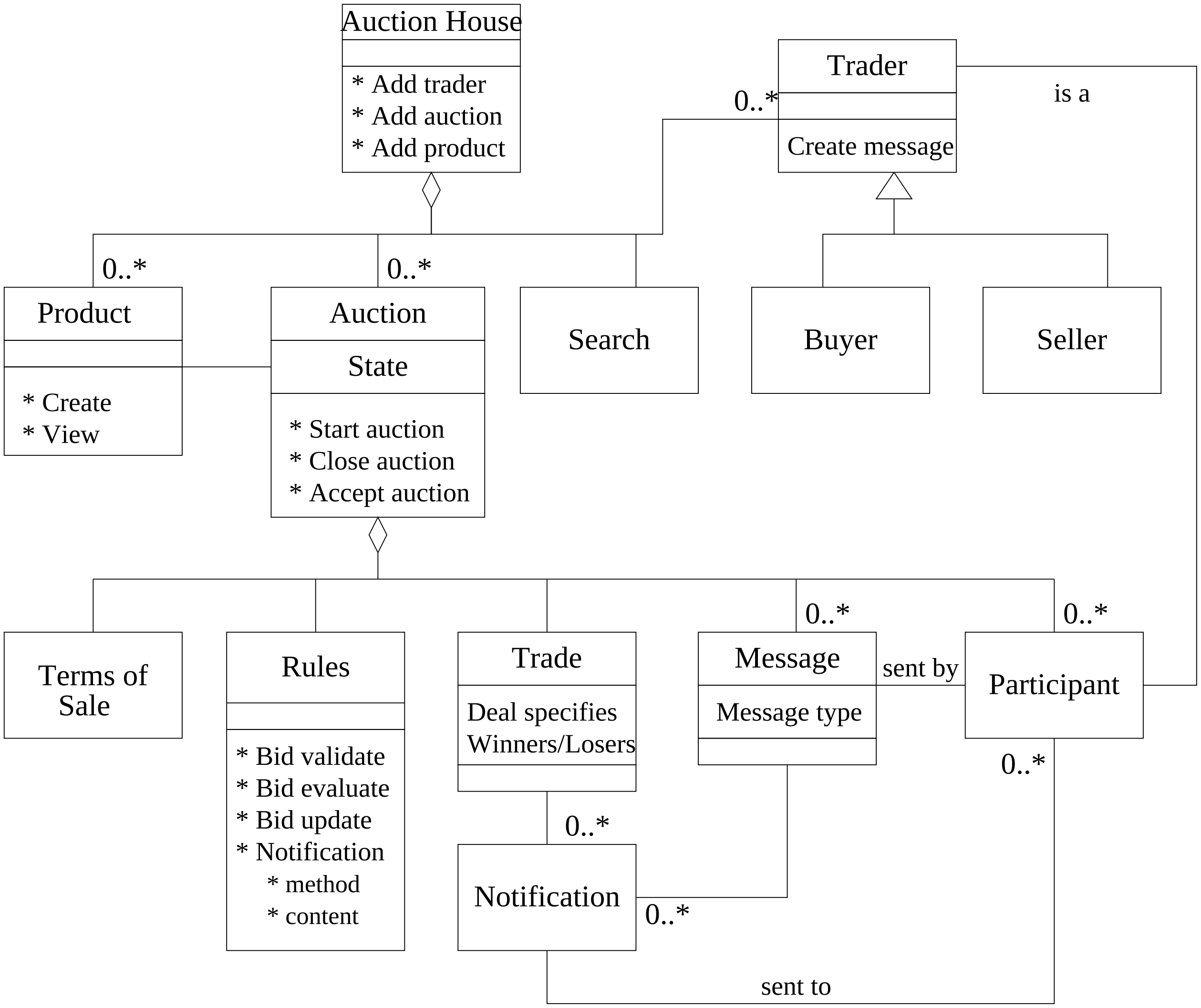
USE CASE DIAGRAM FOR MANAGE AUCTION HOUSE



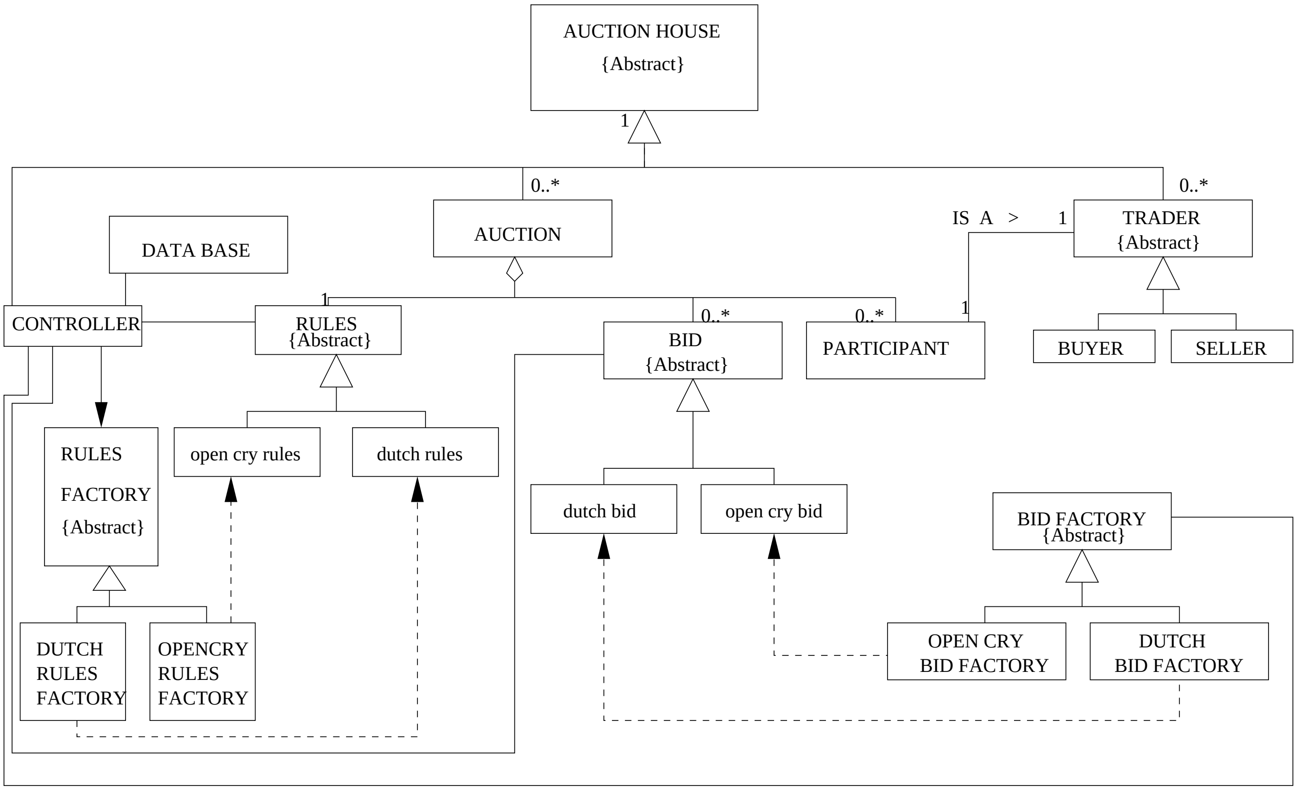
Domain Analysis

- Identify the classes from the specs and the use cases
- Domain classes:
  - Auction House
  - Auction
  - Product
  - Search
  - Buyer
  - Seller
  - Termsofsale
  - Rules
  - Trade
  - Notification,
  - Participant
  - Price

A Class Diagram



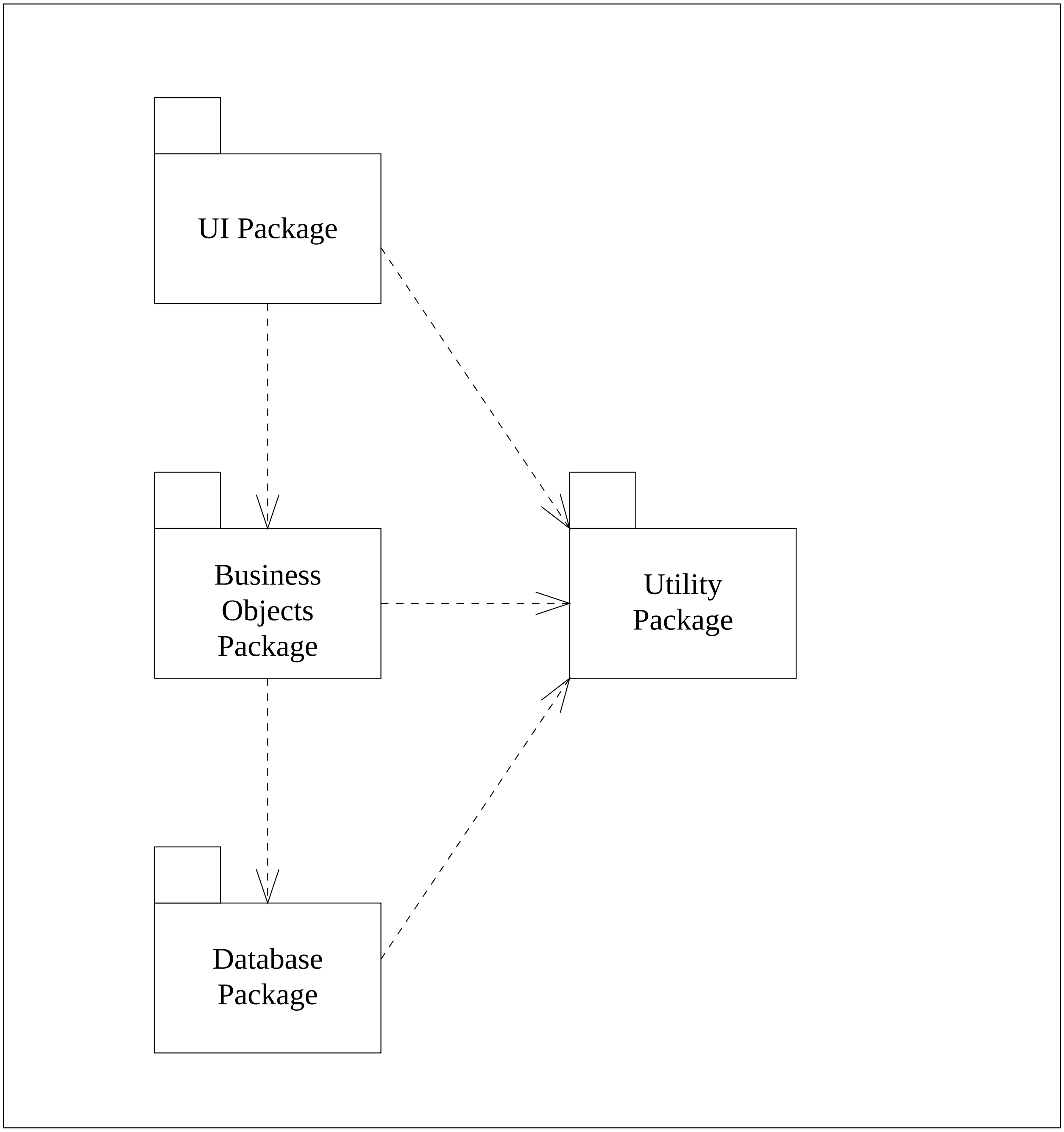
A Detailed Class Diagram



Architecture Design

- High level design where subsystems (packages) are defined, including dependencies and communication mechanisms between between the packages
- A well-designed architecture is the foundation for an extensible and changeable system
- Packages in the Auction System:
  - User Interface Package: based on the Java AWT package
  - Business Objects Package: includes domain classes
  - Database Package: provides persistence to business objects
  - Utility Package: services that are used in other packages

Architecture of Subsystems



Detailed Design

- Describe the new technical classes; expand and detail the descriptions of business object classes; (through more detailed UML diagrams)
- **Database Package**:
  - *Persistent* is a class that all classes that need persistent objects must inherit
- **Utility Package**:
  - *ObjId* is a class whose objects are used to refer to any persistent object in the system; used by all packages in the system
- **Business Objects Package**:
  - Class descriptions are detailed
  - Diagrams are refined further taking into account design-level details
- **User Interface Package**

Design Patterns

- **Abstract Factory**: for generating objects for different auction protocols
- **Singleton**: Auction House is a singleton
- **Builder**: Separate construction of web pages from representation (html pages, forms, other ways of rendering, etc.)
- **Bridge**:
  - Avoid permanent binding between auction type and effective price
  - Determining winner in combinatorial auctions
  - Notification mechanisms
- **Composite**: to represent bids with complex structure
- **Decorator**: decorate auction object with multiple individual value added services

- **Facade**: Auction house provides a unified interface to the whole system
- **Iterator**: To access database information
- **Proxy**: To provide detailed description of item
- **Observer**: automatic generation of services, notifications, etc.
- **Strategy**: for different types of auction protocols
- **Chain of Responsibility**: consotium can forward requests to a chain of auction houses

The Auction Process

1. Registration of buyers and sellers
   - authentication of servers and clients (SSL)
   - authentication of trading parties
   - exchange of cryptography keys
   - may involve revealing credit card information
   - information transmitted is substantial and most of it is to be kept confidential
   - trader profiling
2. Setting up an Auction Event
   - describe the item: certification may be required for the item being described
   - set up auction rules: type of auction, reserve price, negotiable parameters, start time of auction, penalties on withdrawal, auction closing rules

- Some items are not revealed (for example, reserve price, number of items available)
- Auction may not be publicly accessible, so access control may need to be enforced
- Digital signing of contracts under third party supervision may be required for non-repudiation

3. Scheduling and Advertising
    - unauthorized postings are to be prevented
    - unauthorized alterations are to be prevented
4. Bidding process
    - collect bids from buyers
    - implement bid control rules
    - Unusual behavior needs to be monitored: abnormal speculation, frequent withdrawals, etc.
    - only a subset of bidding history is to be made available to bidders

- verfiable connection from every bid to its bidder
- access control is to be enforced
- Denial of service attacks to be countered
- prevent: tampering with a bid, spurious bids, disclosing of bids to other bidders, fraudulent activity by auctioneer, bidder collusion

5. Evaluation of bids and determination of winners:
   - pricing issues: discriminative, non-discriminative
   - revenue maximization: combinatorial auctions
6. Electronic payment: payment protocols, standards enter the fray here (for example, SET)
7. Trade settlement: Backend synchronization is the key here

Technologies Required

- Object Technology: OOAD and UML, Design Patterns and Analysis Patterns
- Distributed Objects: RMI, CORBA, DCOM
- Java Technology (Java, JavaBeans, JavaScript, EJB)
- Internet Technologies: XML, PERL, CGI scripting, cookies
- Agent Technology: Mobile Objects
- Pervasive computing
- Security services
- Electronic payment mechanisms