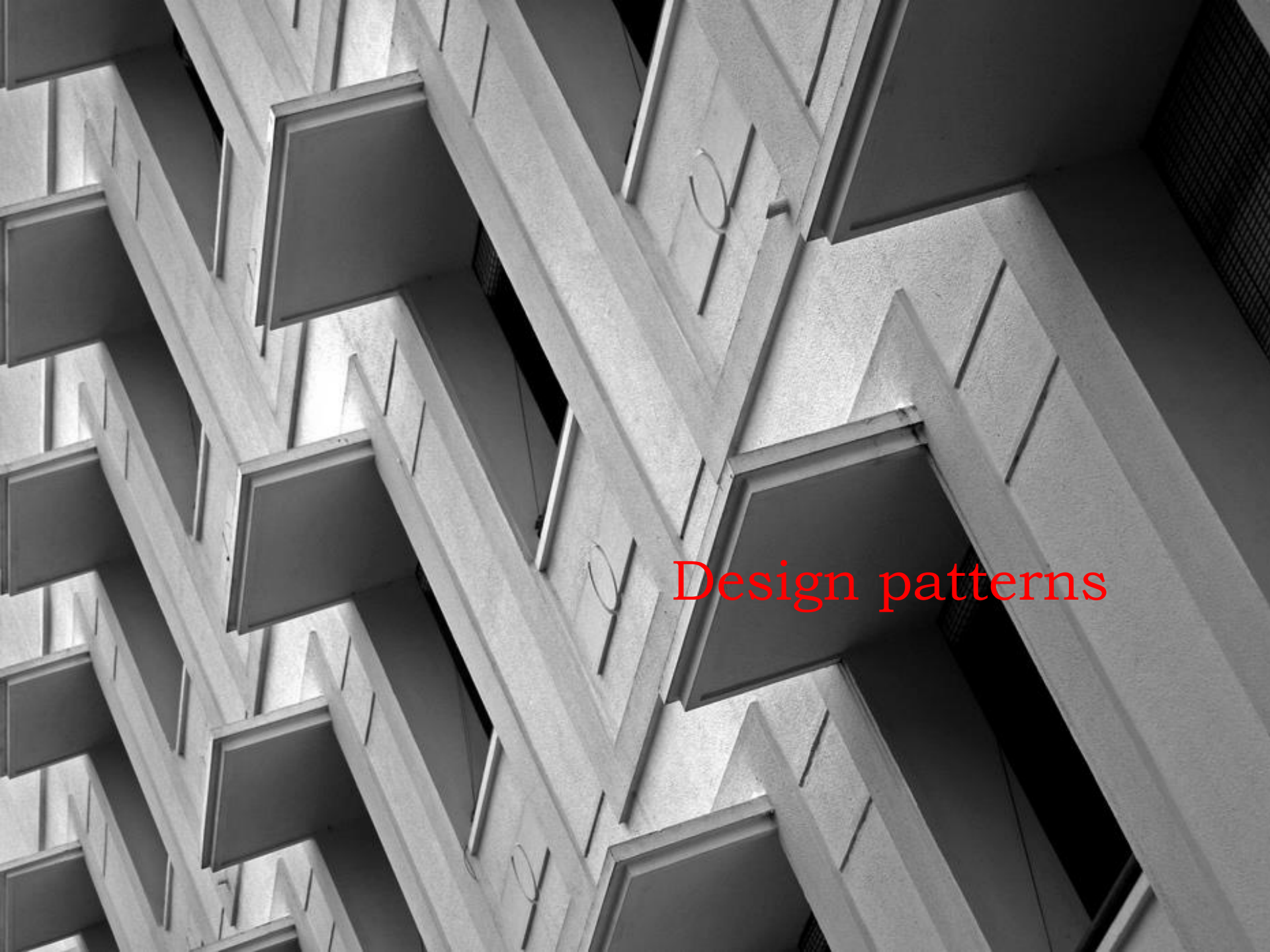
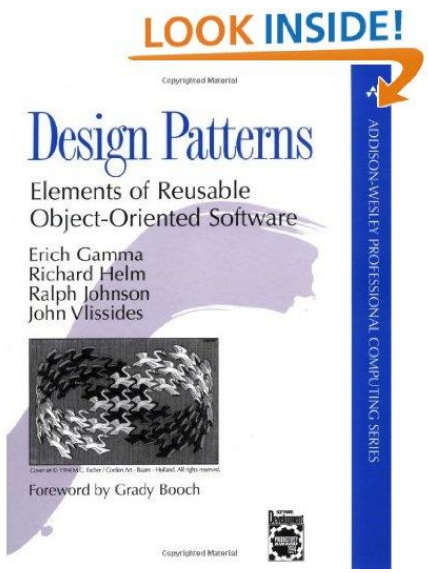
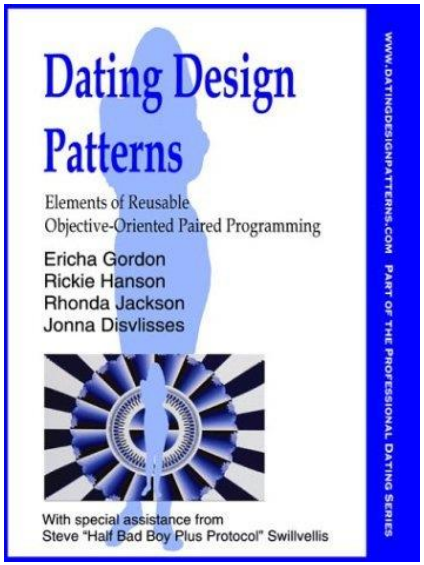

Design patterns

Sources

- Cours de Pascal Molli « A System of Pattern » Bushmann et All
- « Design Patterns » Gamma et All (GoF)
- « Applying UML and Patterns » Larman
- "Design Patterns Java™ Workbook" Steven John Metsker





Design for change

- You should avoid
  - Creating an object by specifying a class explicitly.
  - Dependence on specific operations.
  - Dependence on object representations or implementations.
  - Algorithmic dependencies.
  - Tight coupling.
  - Extending functionality by subclassing.
  - Inability to alter classes conveniently

Patterns…

- « Patterns help you build on the collective experience of skilled software engineers. »
- « They capture existing, well-proven experience in software development and help to promote good design practice »
- « Every pattern deals with a specific, recurring problem in the design or implementation of a software system »
- « Patterns can be used to construct software architectures with specific properties… »

Becoming a Chess Master

- First learn rules and physical requirements
- Then learn principles
- However, to become a master of chess, one must study the games of other masters
- There are hundreds of these patterns

Becoming a Software Designer Master

- First learn the rules
  - e.g., the algorithms, data structures and languages of software
- Then learn the principles
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns

What is a pattern ?

- A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.
- Patterns document existing, well-proven design experience.
- Patterns identify and specify abstractions that are above the level of single classes and instances, or of components
- Patterns provide a common vocabulary and understanding for design principles

From POSA

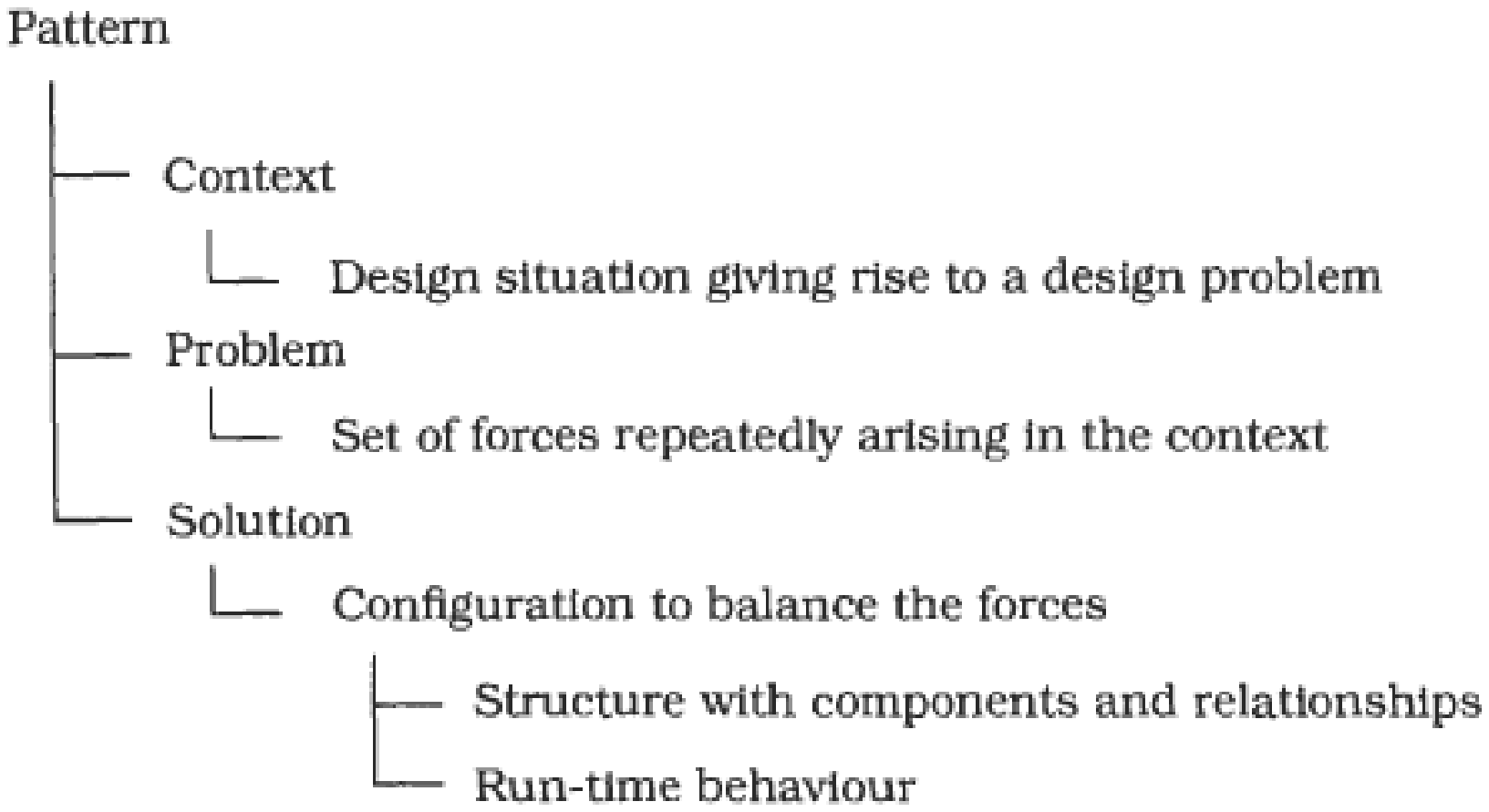
What is a pattern

- Patterns are a means of documenting soffware architectures.
- Patterns support the construction of software with defined properties.
- Patterns help you build complex and heterogeneous software architectures.
- Patterns help you to manage somare complexity.

What is a pattern (continued)

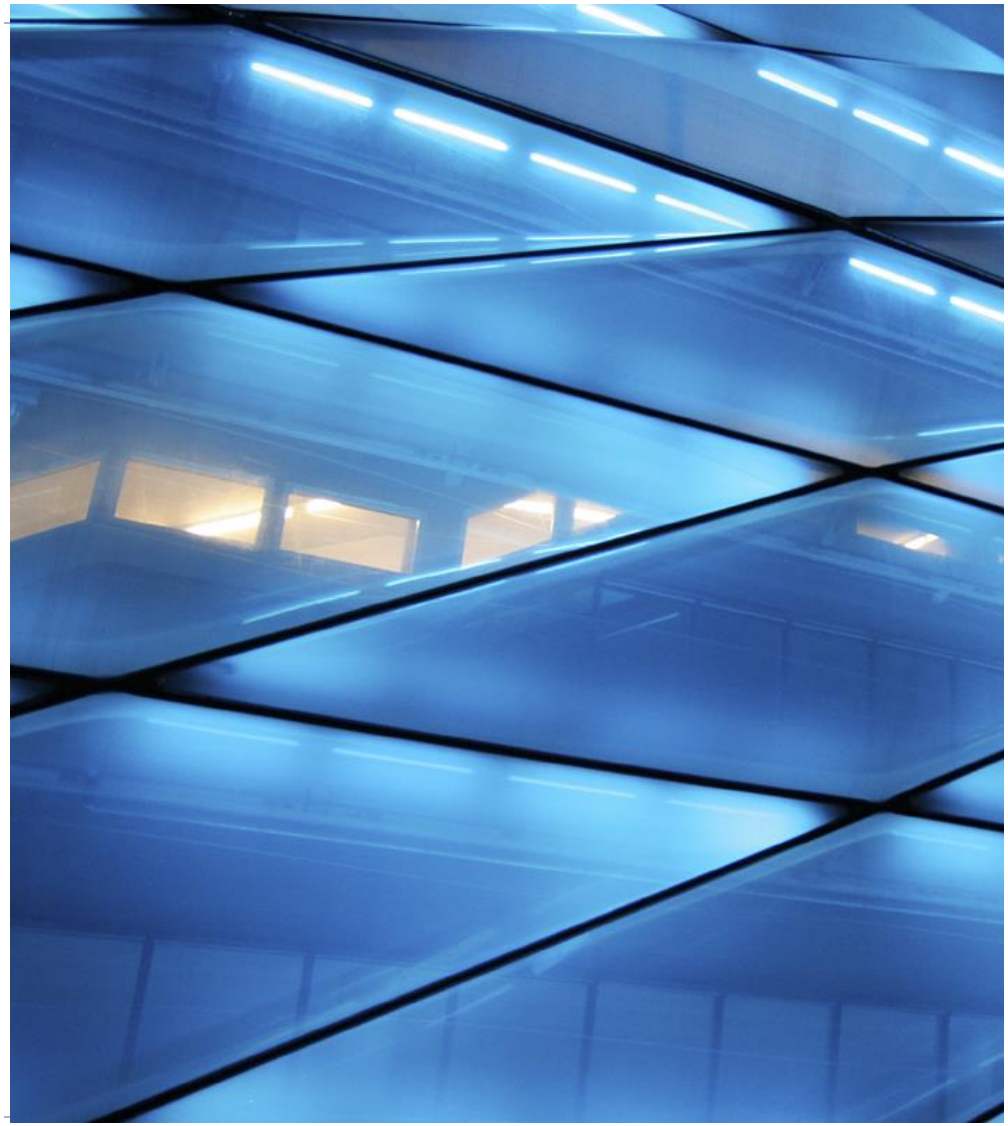
- A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.
- The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

Pattern constitution



Patterns categories

- Architectural patterns
- Design Patterns
- Idioms

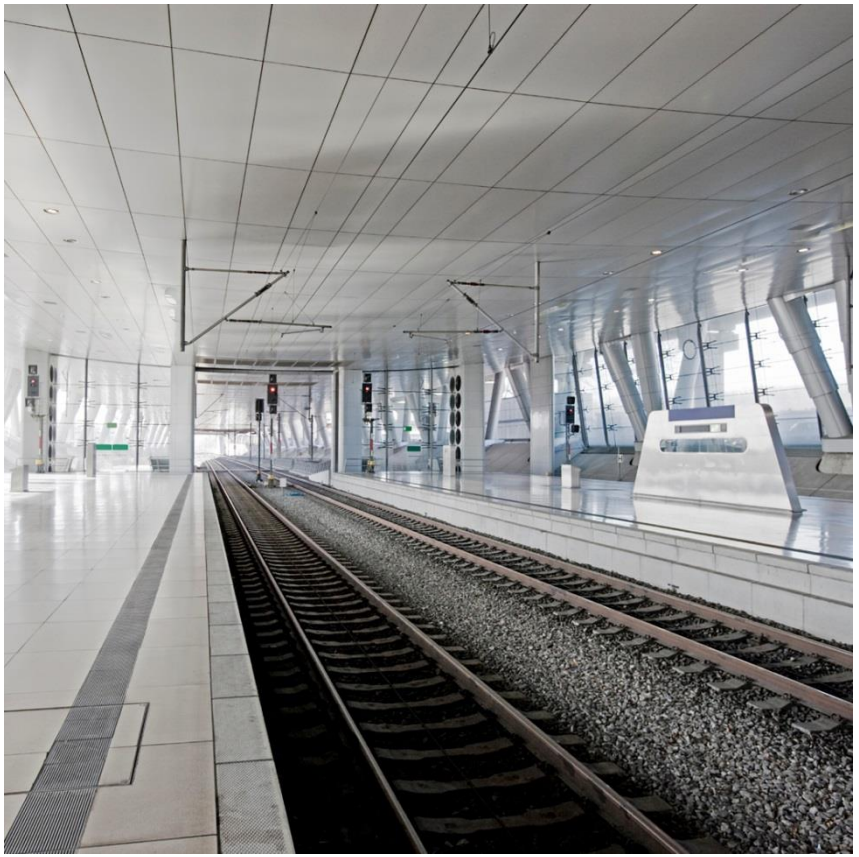


Architectural Patterns

- An architectural Pattern express a fundamental structural organization schema for software systems.
- It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Design patterns

- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relation ships between them.
- It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.



Idioms

- An Idiom is a low-level pattern specific to a programming language.
- An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Pattern Description (1)

- **Name** The name and a short summary of the pattern.
- **Also Known As** Other names for the pattern, if any are known.
- **Example** A real-world example demonstrating the existence of the problem and the need for the pattern. Throughout the description we refer to the example to illustrate solution and implementation aspects, where this is necessary or useful. Text that is specifically about the example is marked by the r symbol at its beginning and by the D symbol at its end.
- **Context** The situations in which the pattern may apply
- **Problem** The problem the pattern addresses, including a discussion of its associated forces.
- **Solution** The fundamental solution principle underlying the pattern.
- **Structure** A detailed specification of the structural aspects of the pattern, including CRC-cards

Pattern description (2)

- **Dynamics** Typical scenarios describing the run-time behavior of the pattern.
- **Implementation** Guidelines for implementing the pattern.
- **Example resolved** Discussion of any important aspects for resolving the example that are not yet covered in the Solution, Structure, Dynamics and Implementation sections.
- **Variants** A brief description of variants or specializations of a pattern.
- **Known Uses** Examples of the use of the pattern, taken from existing systems.
- **Consequences** The benefits the pattern provides, and any potential liabilities.
- **See Also** References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.

Pattern description (Gof)

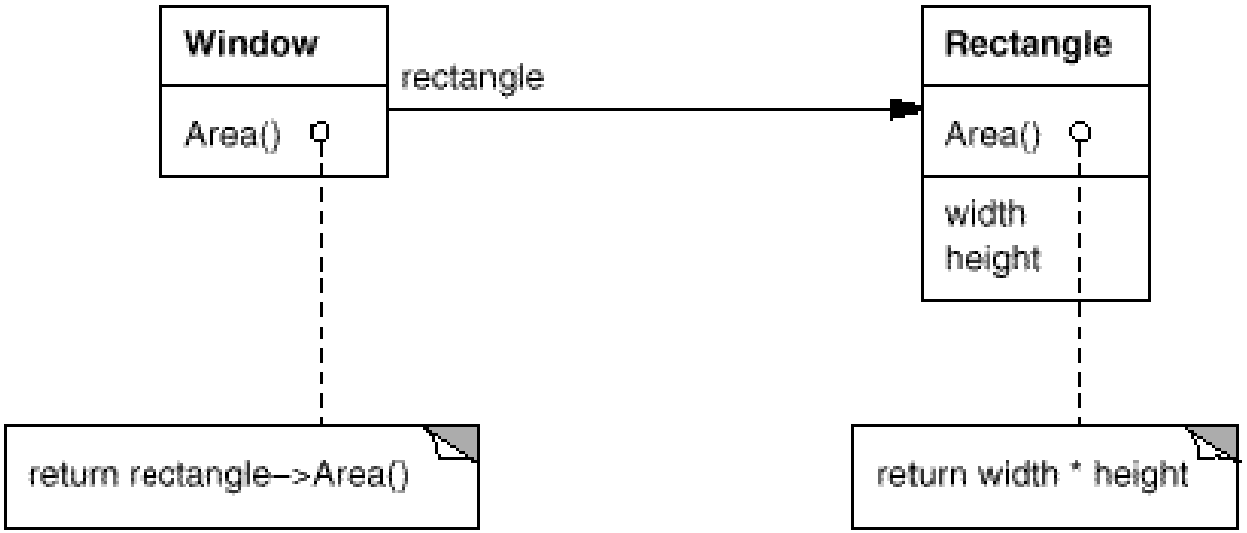
- Name – Aliases
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences

How patterns solve problems

- Finding Appropriate Objects
- Determining Object Granularity
- Specifying Object Interfaces
- Specifying Object Implementations
  - Class versus Interface Inheritance
  - Programming to an Interface, not an Implementation
- Putting Reuse Mechanisms to Work
  - Inheritance versus Composition
  - Favor object composition over class inheritance.
  - Delegation

How patterns solve problems (2)

- Example of delegation
  - The window delegates to the rectangle its behavior



Relating Run-Time and Compile-Time Structures

- acquaintance relationship
  - A class that refers to another class has an acquaintance with that class.
- aggregation relationship
  - The relationship of an aggregate object to its parts. A class defines this relationship for its instances (e.g., aggregate objects).
- No difference in programming languages

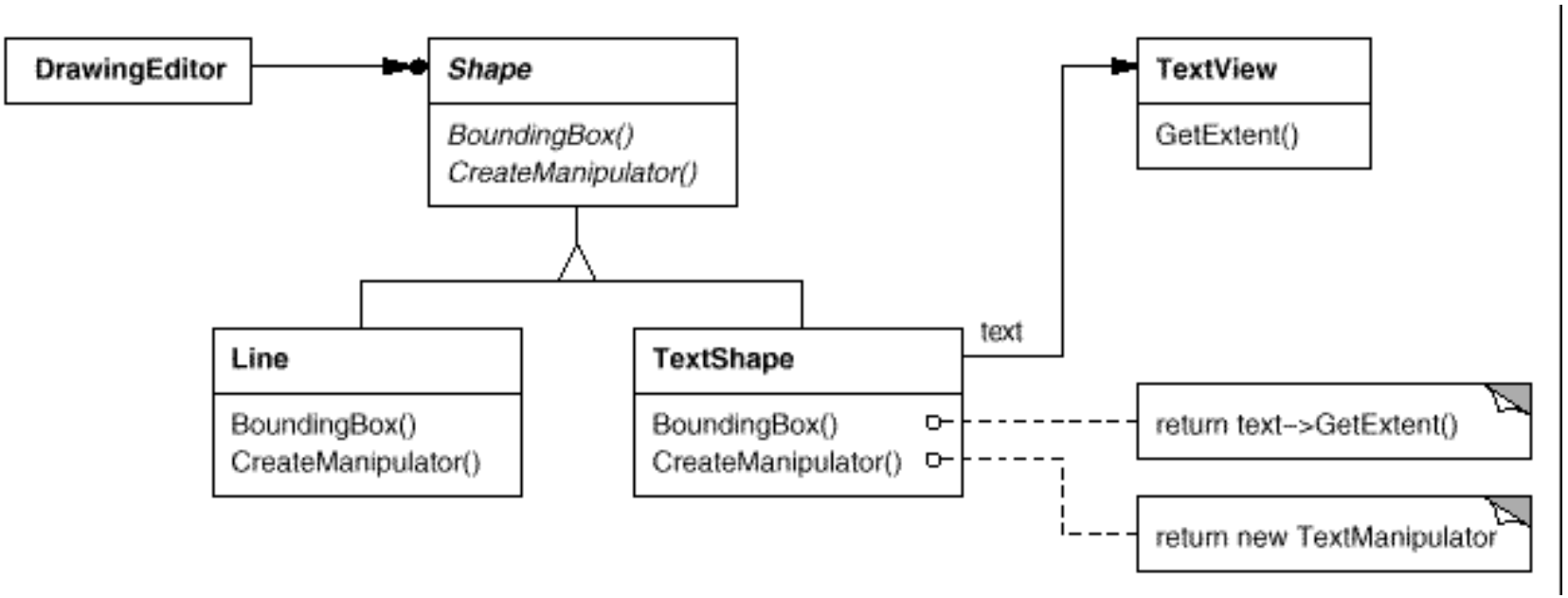
Common Patterns

- Abstract Factory
- Adapter
- Composite
- Decorator
- Factory Method
- Observer
- Strategy
- Template Method

Adapter

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

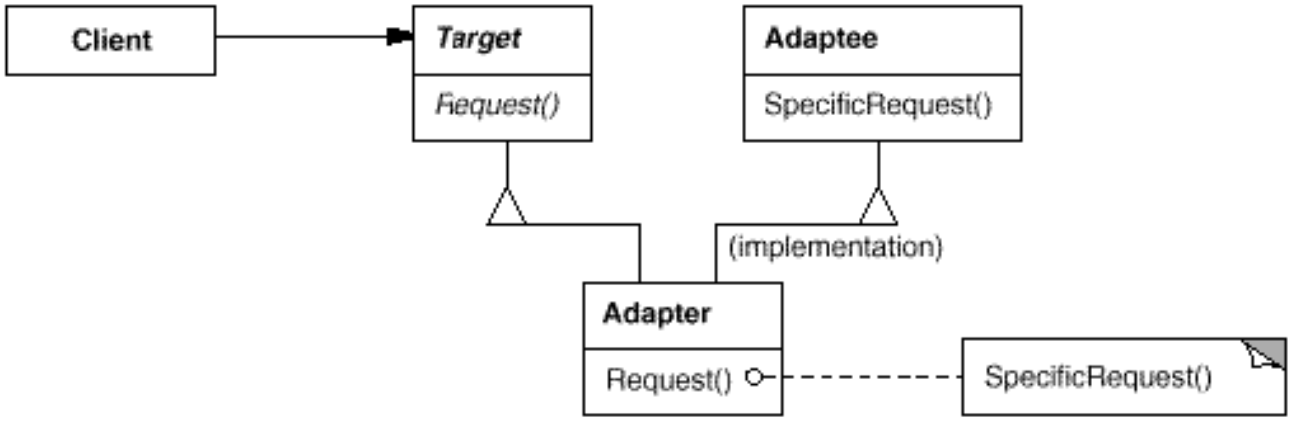
Motivation

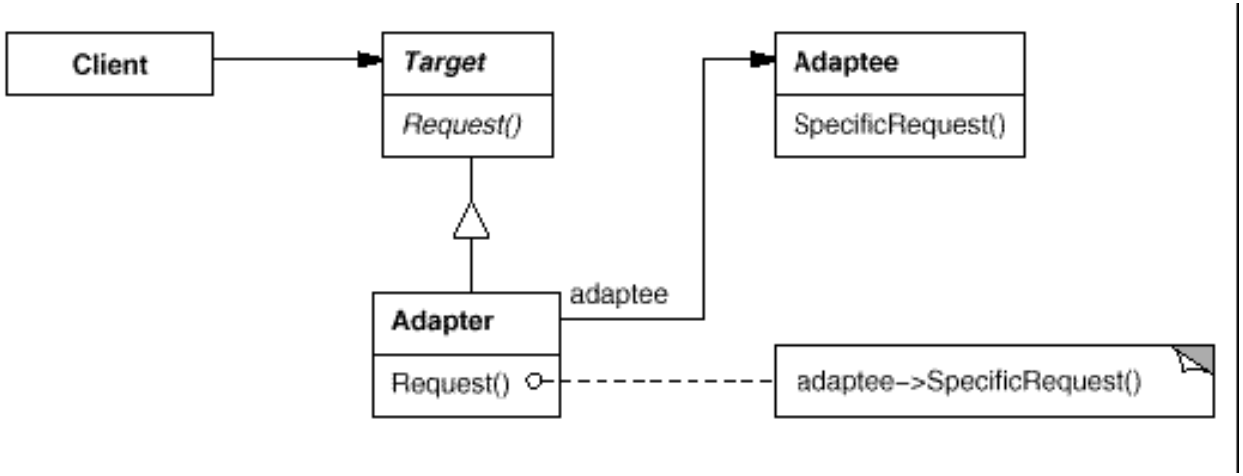

DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator

Applicability

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

Structure


Client
Target
Request()
Adaptee
SpecificRequest()
(implementation)
Adapter
Request()
SpecificRequest()


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

Participants

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

Collaborations

- Clients call operations on an Adapter instance.
- In turn, the adapter calls Adaptee operations that carry out the request.

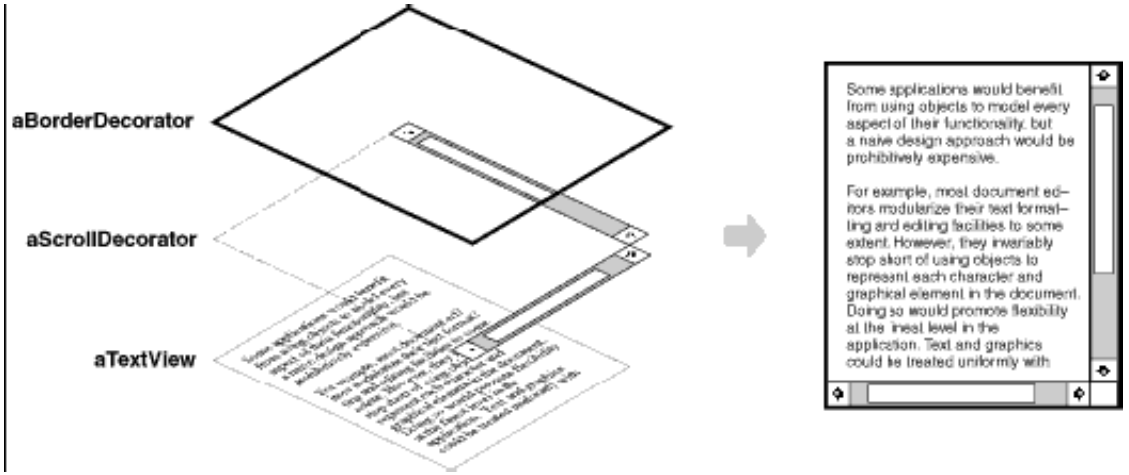
Consequences

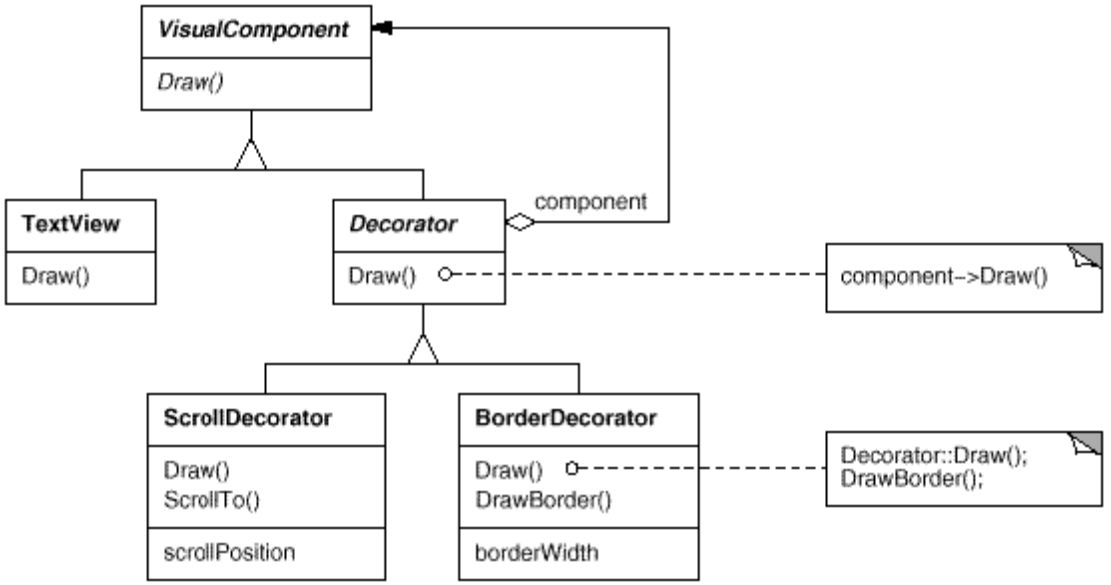
- A class adapter
  - adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- An object adapter
  - lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
  - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Decorator

- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

Motivation

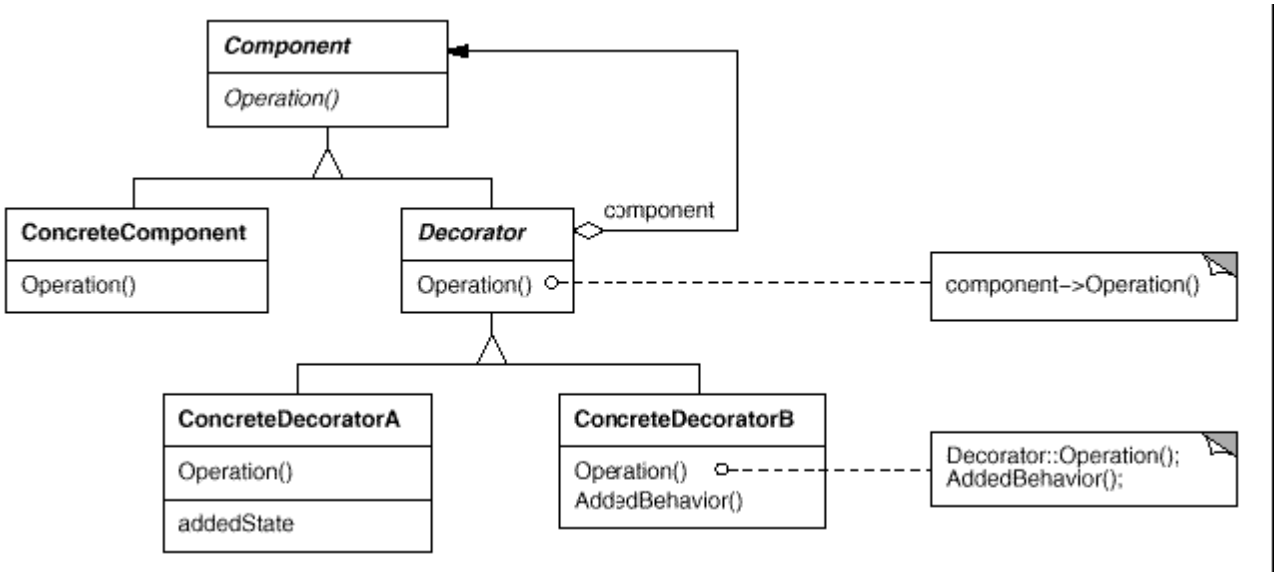

aBorderDecorator
aScrollDecorator
aTextView
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with


VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Applicability

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

Structure


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

Participants

- Component (VisualComponent)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- ConcreteComponent (TextView)
  - defines an object to which additional responsibilities can be attached.
- Decorator
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- ConcreteDecorator (BorderDecorator, ScrollDecorator)
  - adds responsibilities to the component.

Collaborations

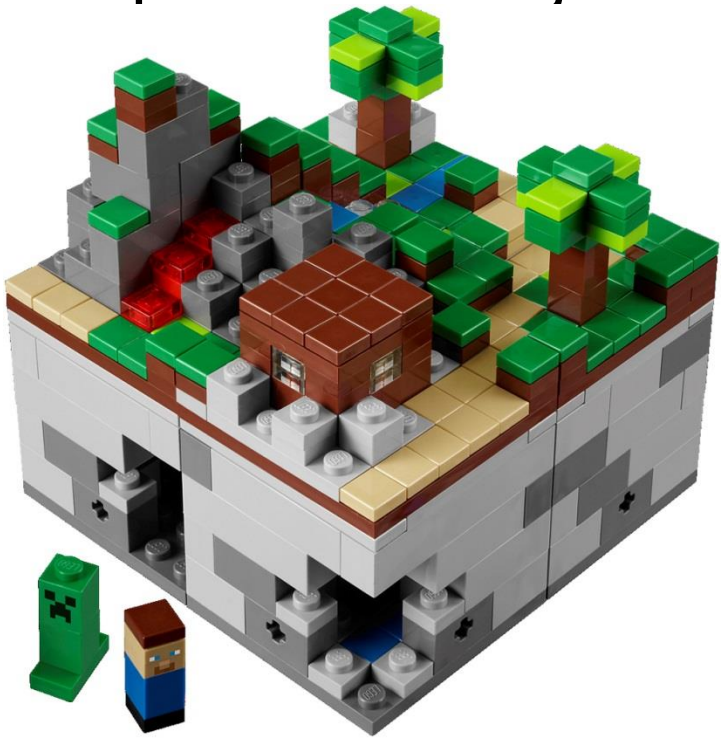
- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

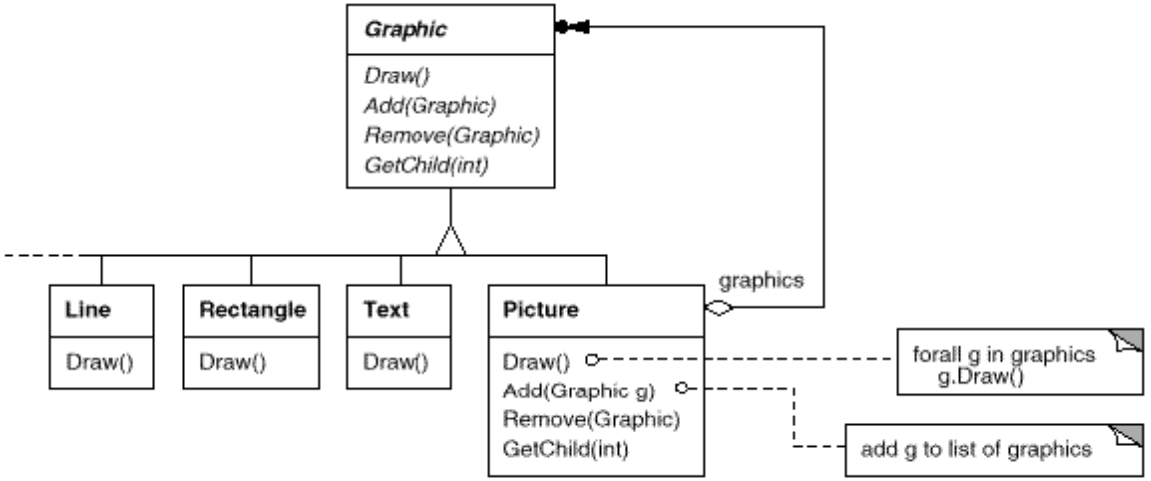
- Decorator Pattern
  - More flexibility than static inheritance.
  - Avoids feature-laden classes high up in the hierarchy.
  - A decorator and its component aren't identical.
  - Lots of little objects.

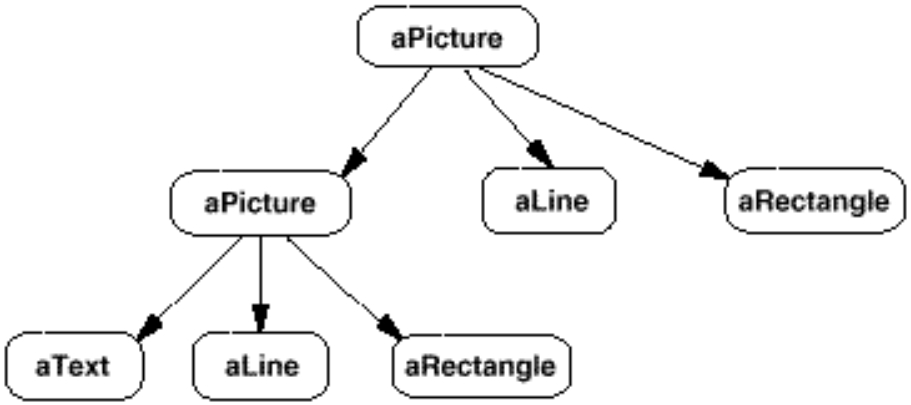
Composite

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Motivation

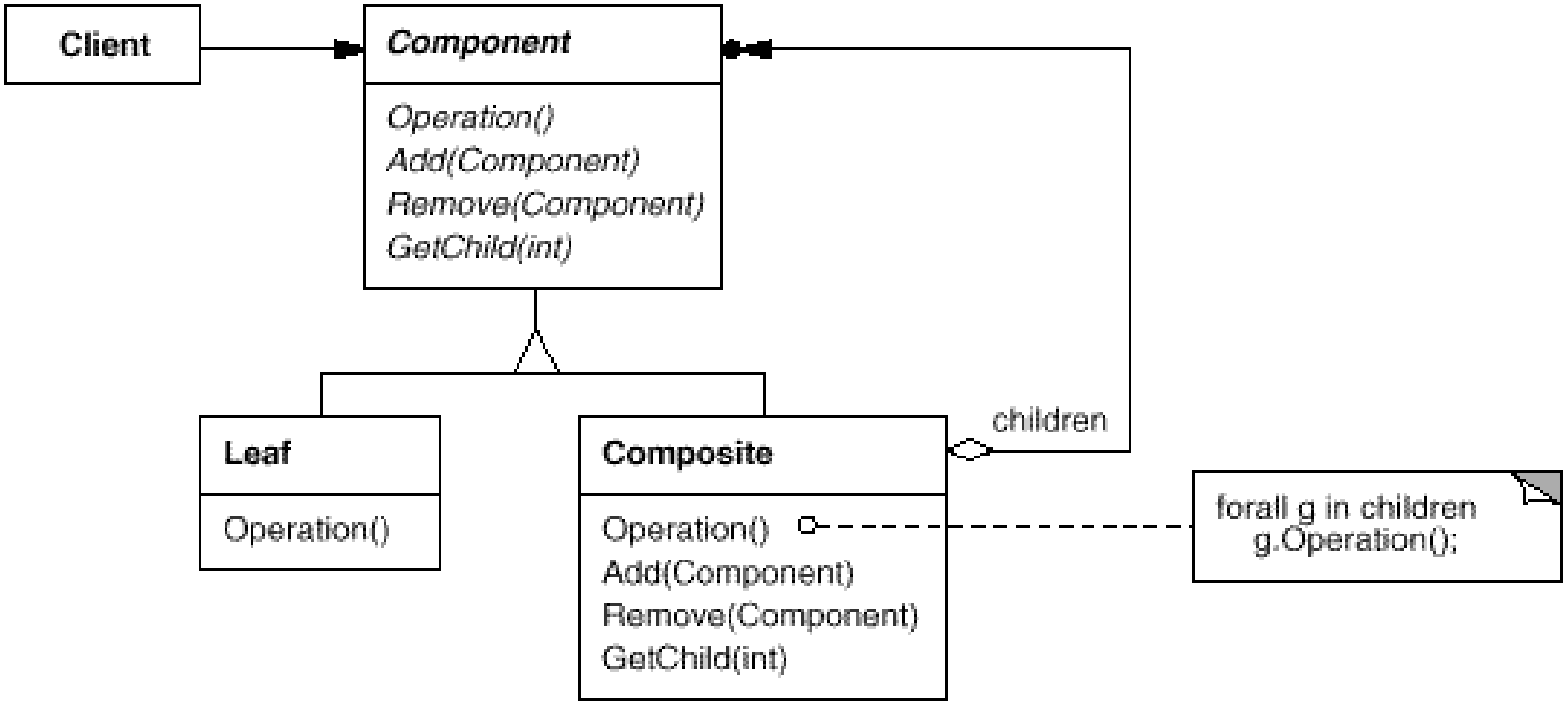

Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
graphics
forall g in graphics
g.Draw()
add g to list of graphics


aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

Applicability

- Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects.
  - Clients will treat all objects in the composite structure uniformly.

Structure


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

Participants

- Component (Graphic)
  - 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.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- Leaf (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- Composite (Picture)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client
  - manipulates objects in the composition through the Component interface.

Collaborations

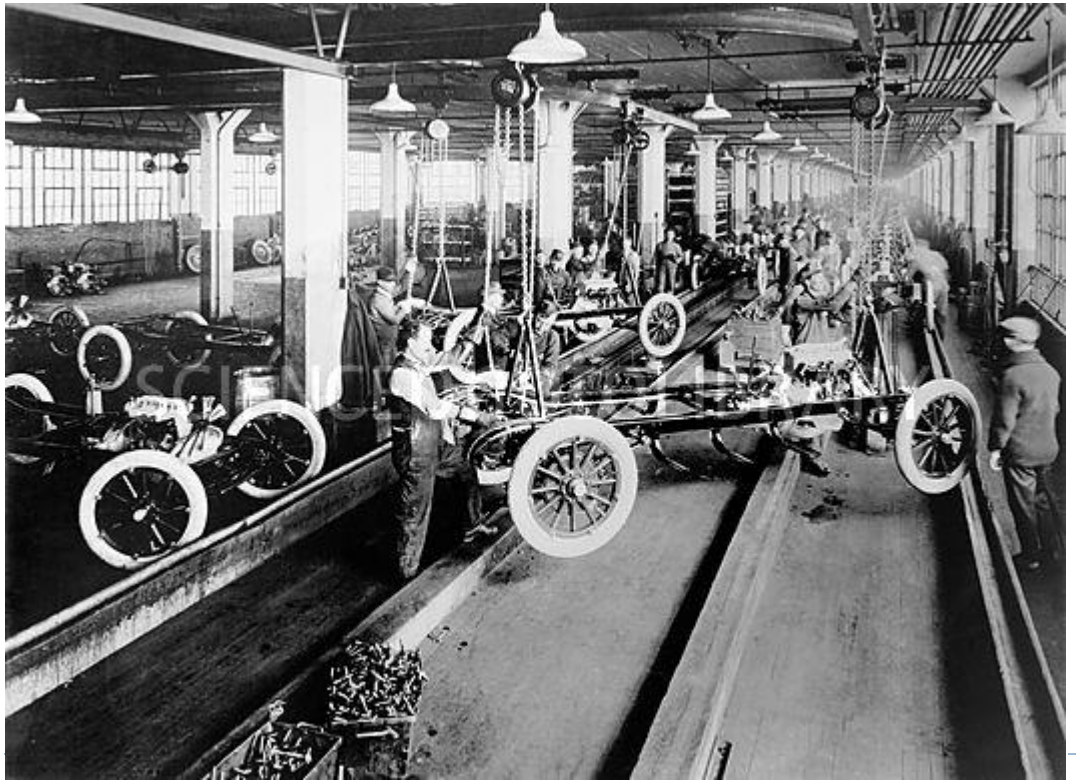
- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

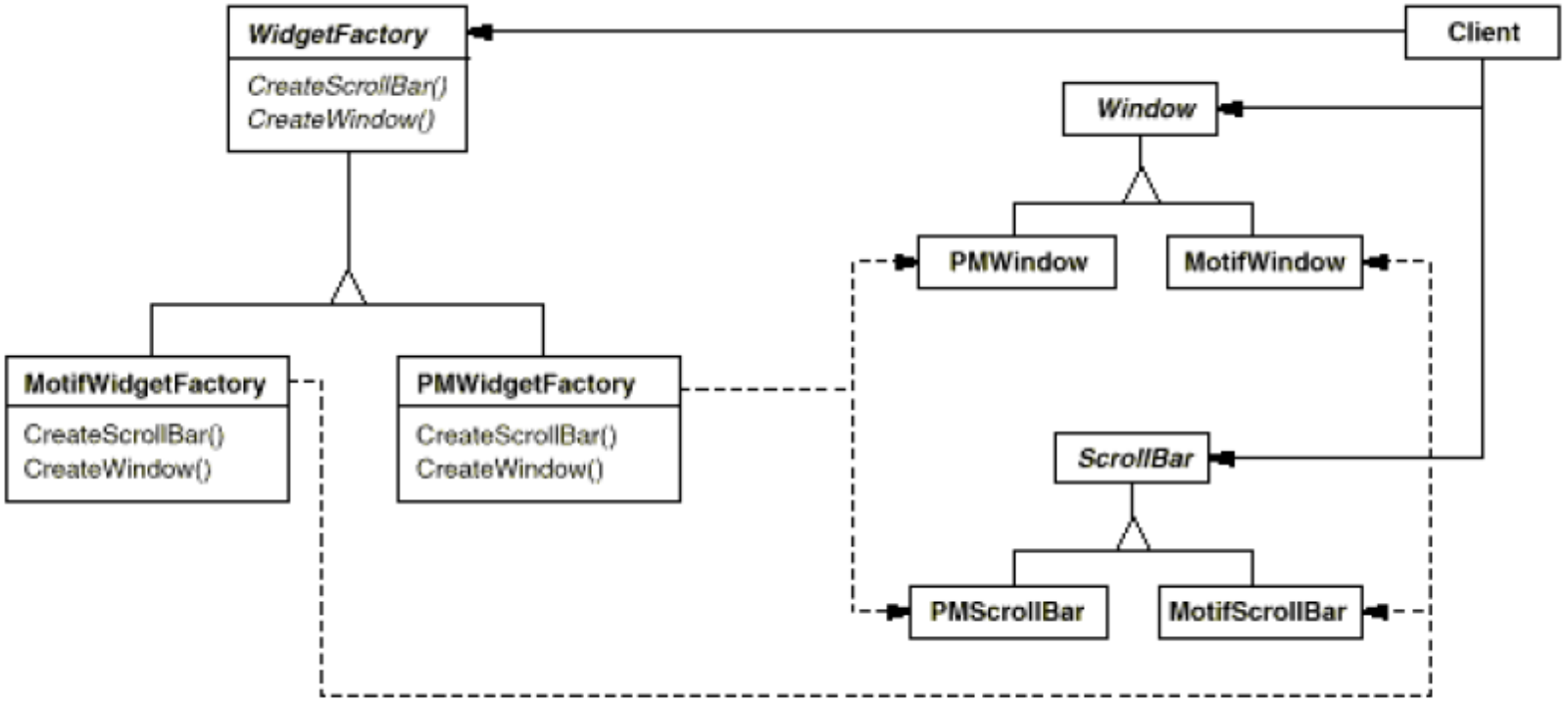
- The Composite pattern
  - defines class hierarchies consisting of primitive objects and composite objects.
  - makes the client simple. Clients can treat composite structures and individual objects uniformly.
  - makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
  - can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite.

Abstract Factory (Kit)

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



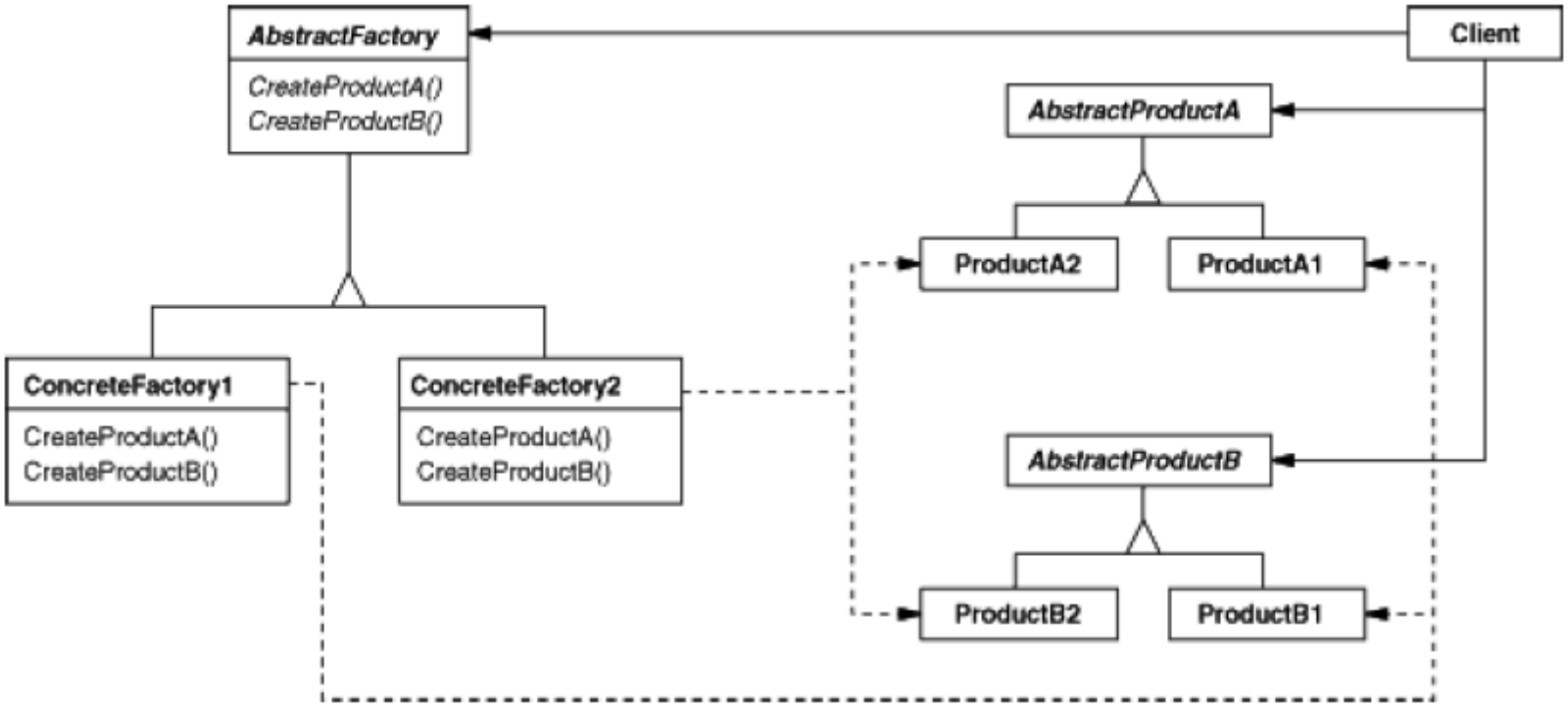
Motivation


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar

Applicability

- Use the Abstract Factory 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 you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

Participants

- AbstractFactory (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- AbstractProduct (Window, ScrollBar)
  - declares an interface for a type of product object.
- ConcreteProduct (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- Client
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

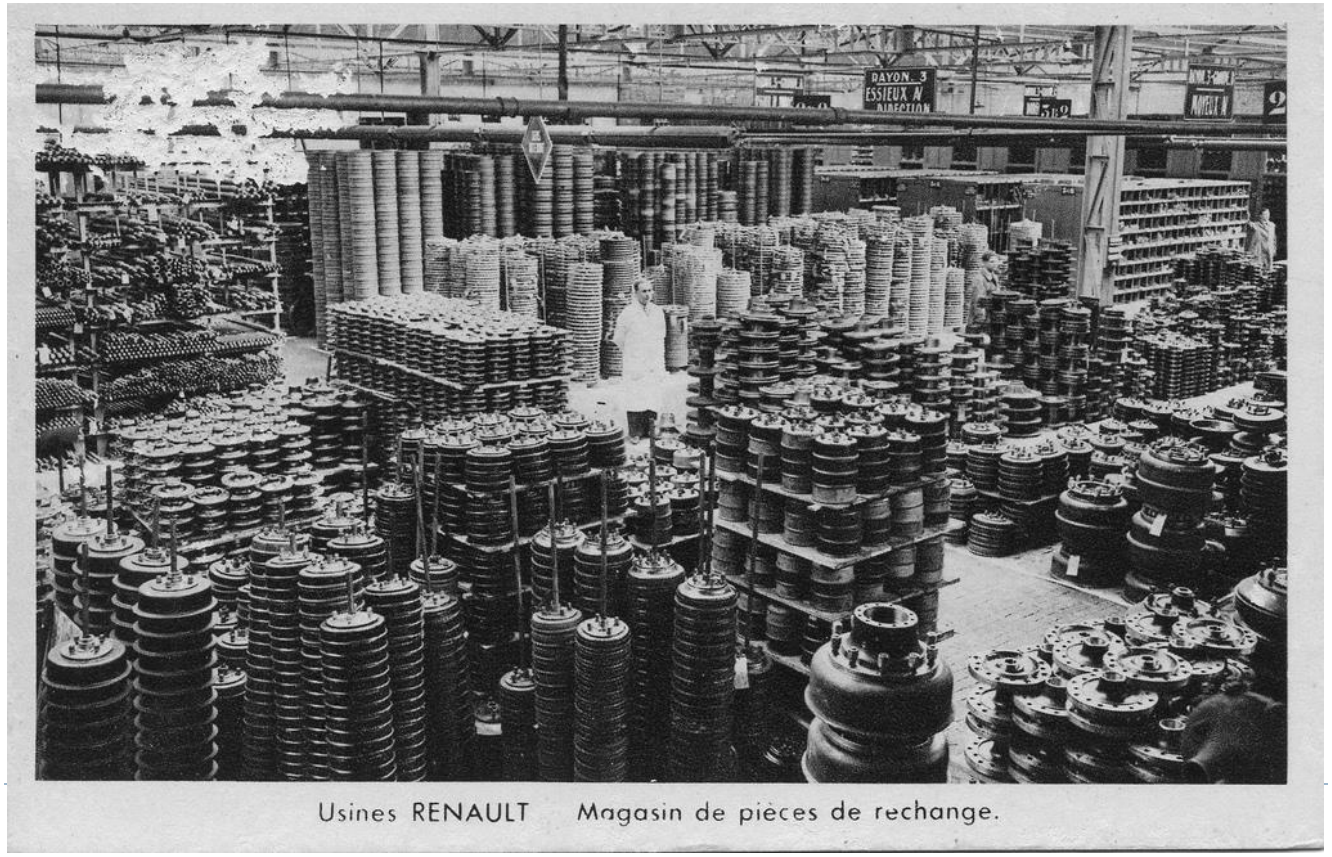
- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

- It isolates concrete classes.
- It makes exchanging product families easy.
- It promotes consistency among products.
- Supporting new kinds of products is difficult.

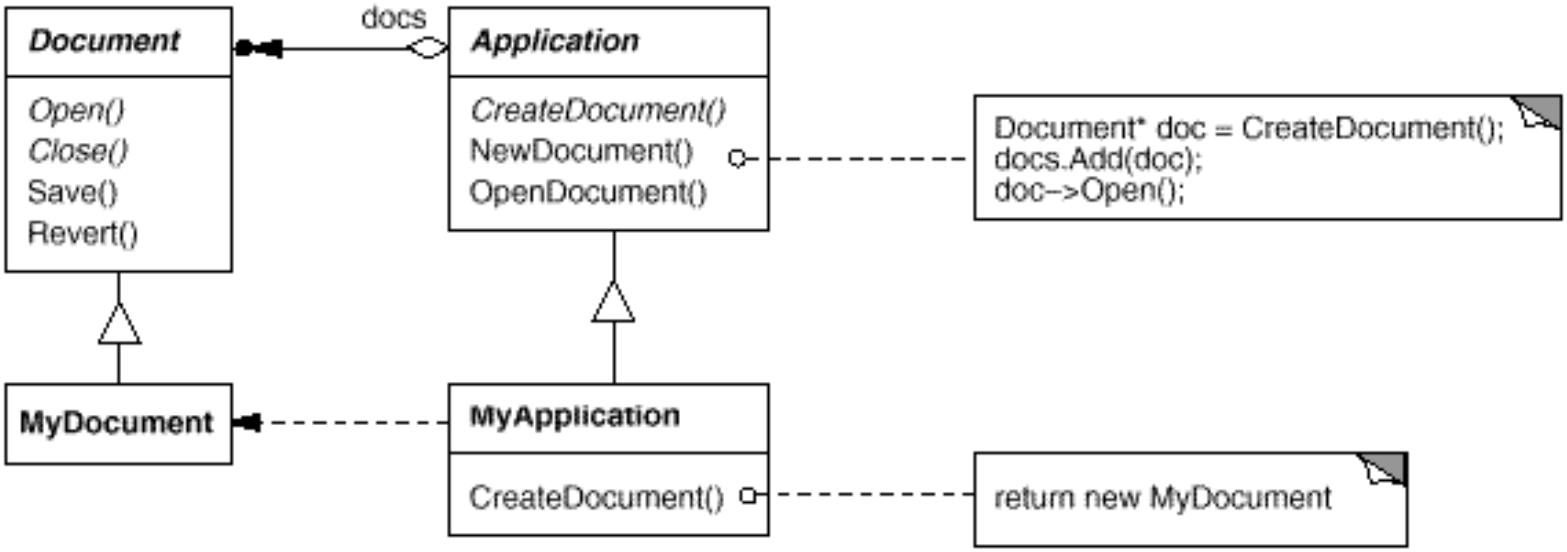
Factory Method (Virtual Constructor)

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Usines RENAULT Magasin de pièces de rechange.

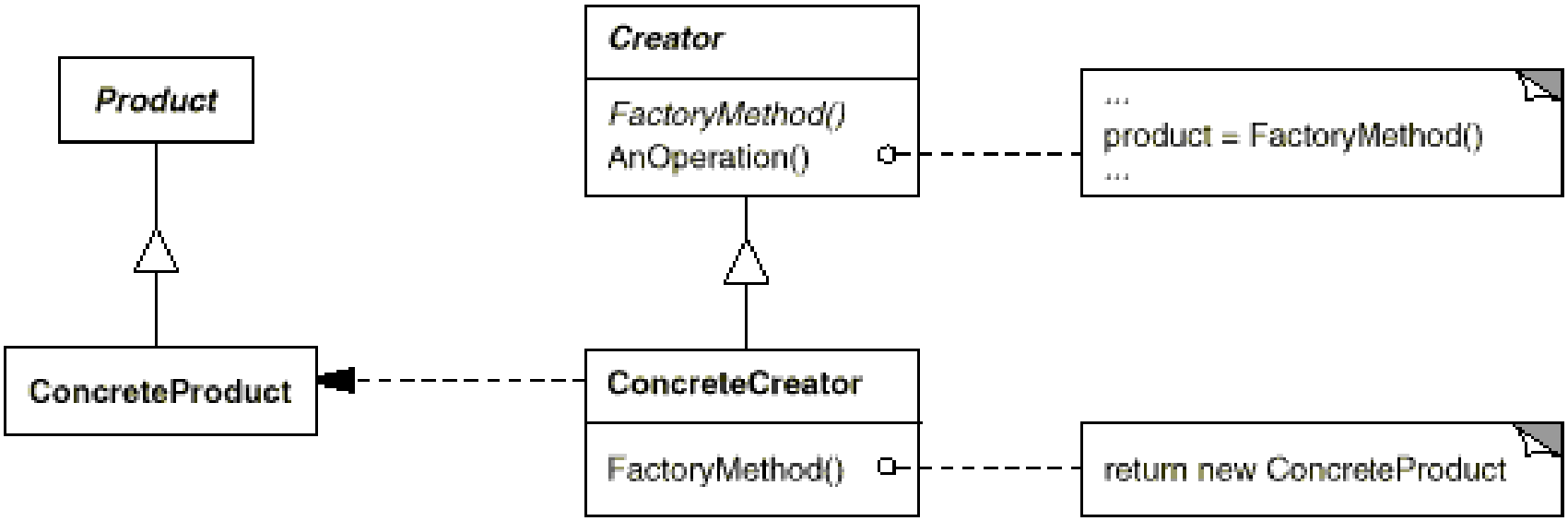
Motivation


Document
Open()
Close()
Save()
Revert()
docs
Application
CreateDocument()
NewDocument()
OpenDocument()
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
MyDocument
MyApplication
CreateDocument()
return new MyDocument

Applicability

- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure


Product
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

Participants

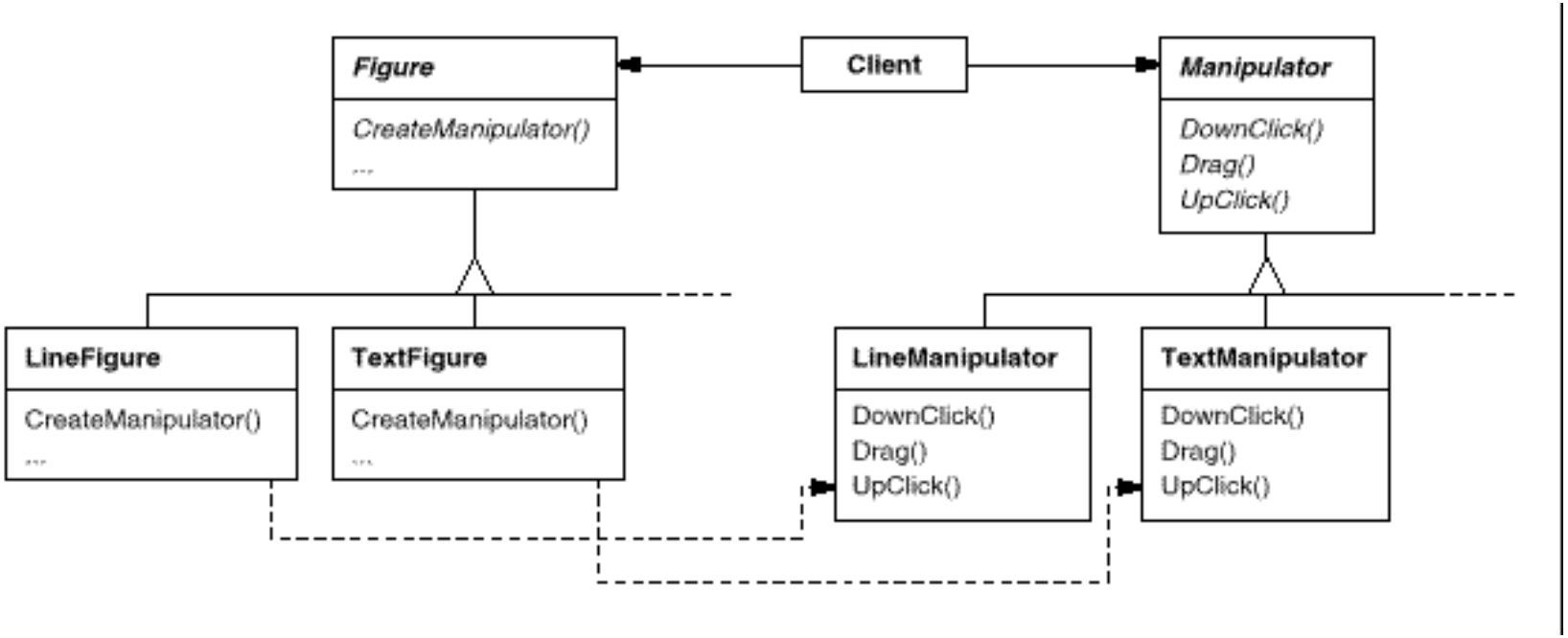
- Product (Document)
  - defines the interface of objects the factory method creates.
- ConcreteProduct (MyDocument)
  - implements the Product interface.
- Creator (Application)
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation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

Collaborations

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

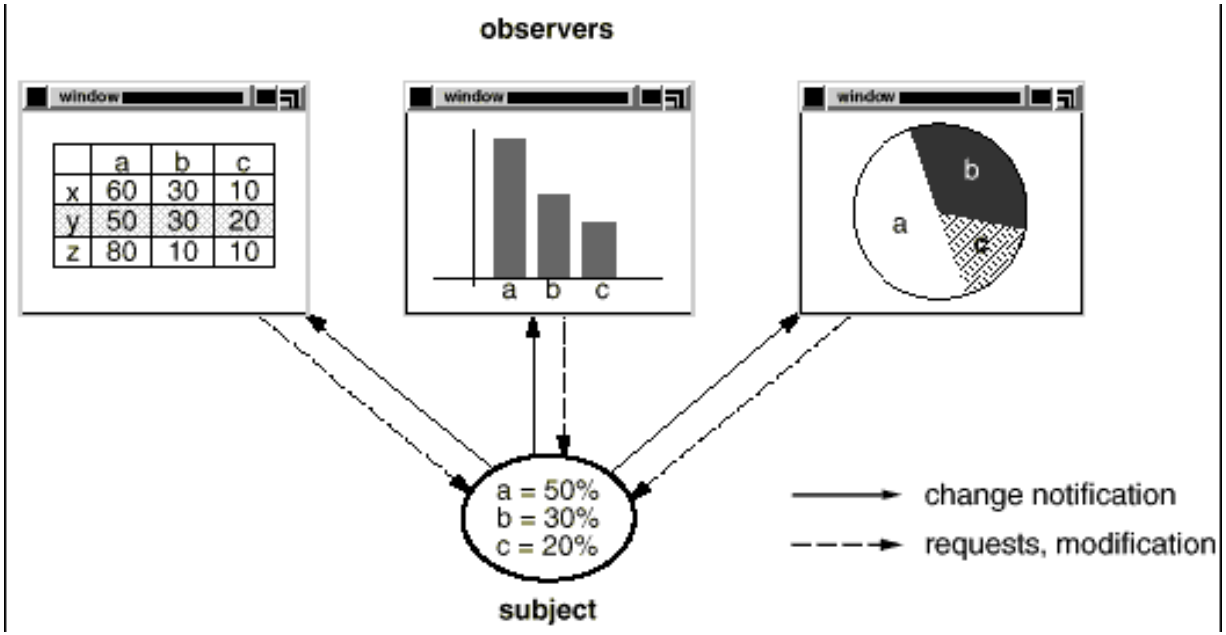
- Provides Hook for subclasses.
- Connect parallel class hierarchies.



Observer

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

Motivation

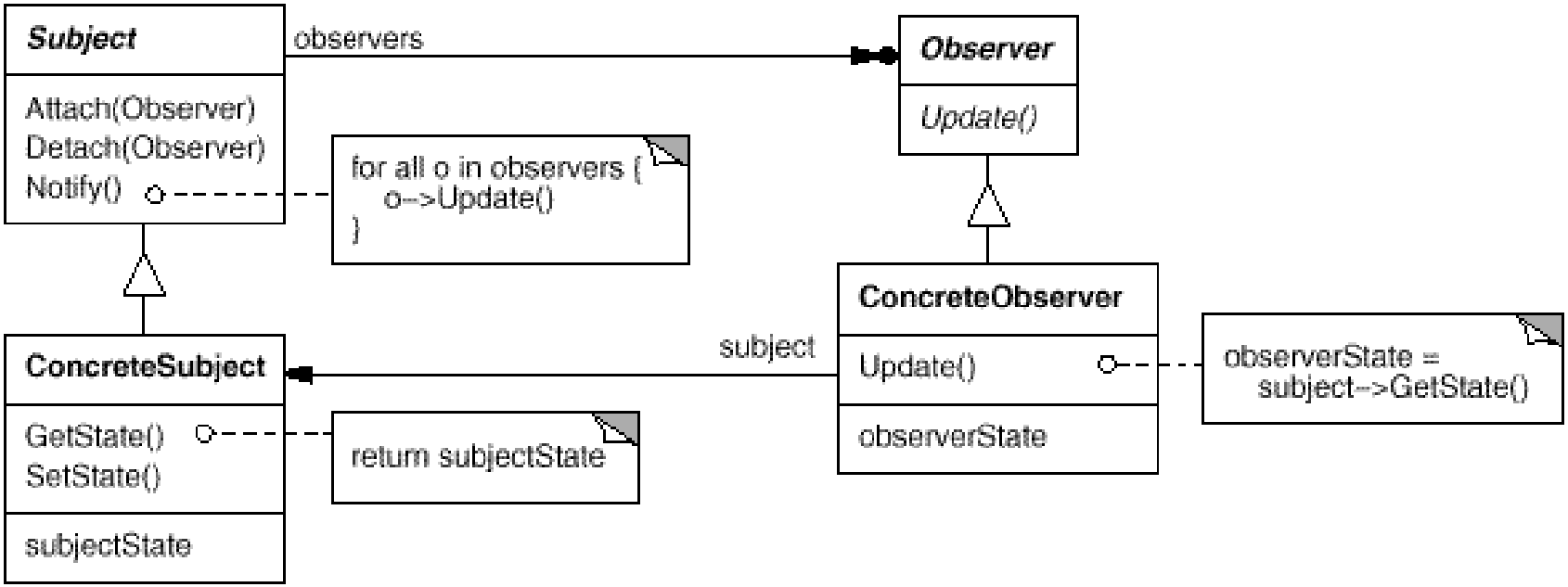

observers
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
b
a
c
a b c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

Applicability

- Use the Observer pattern in any of the following situations:
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

GoF

Structure


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Update()
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

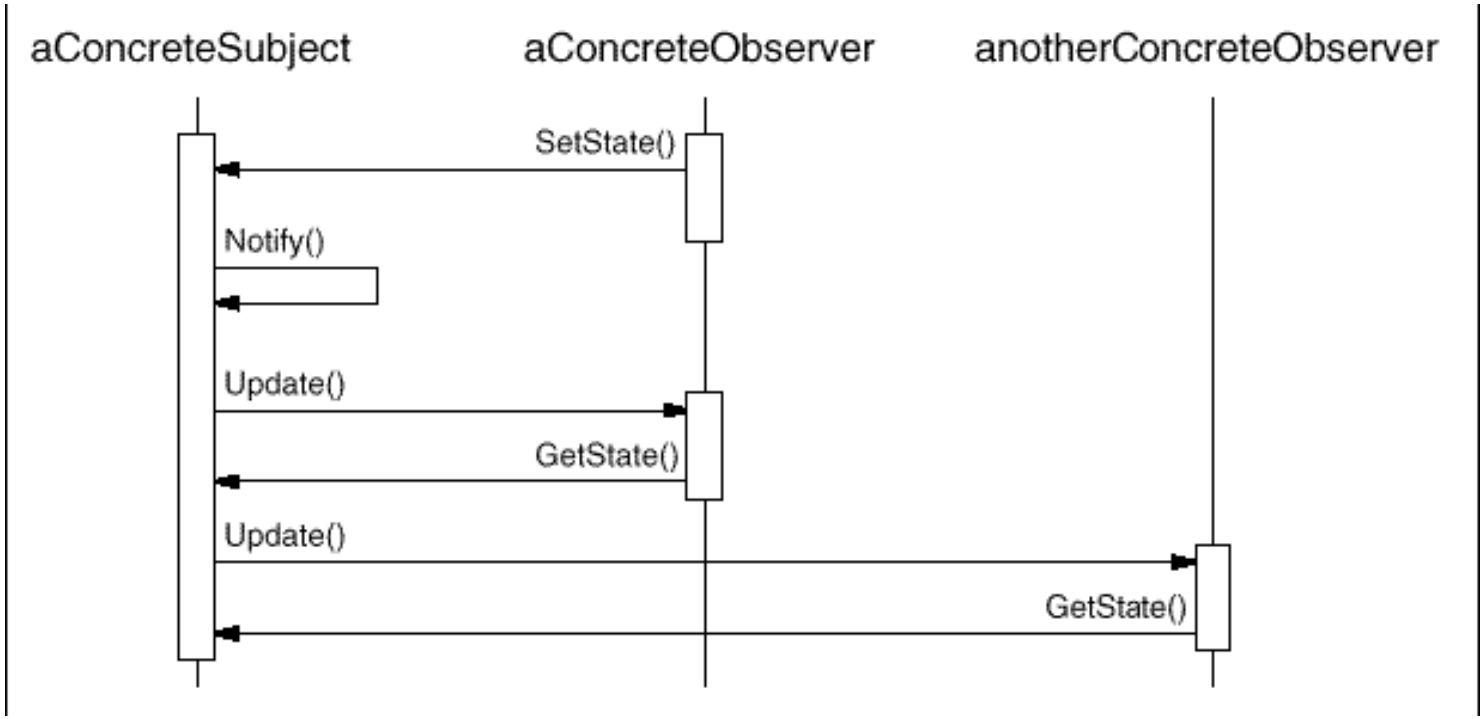
Participants

- Subject
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Collaborations



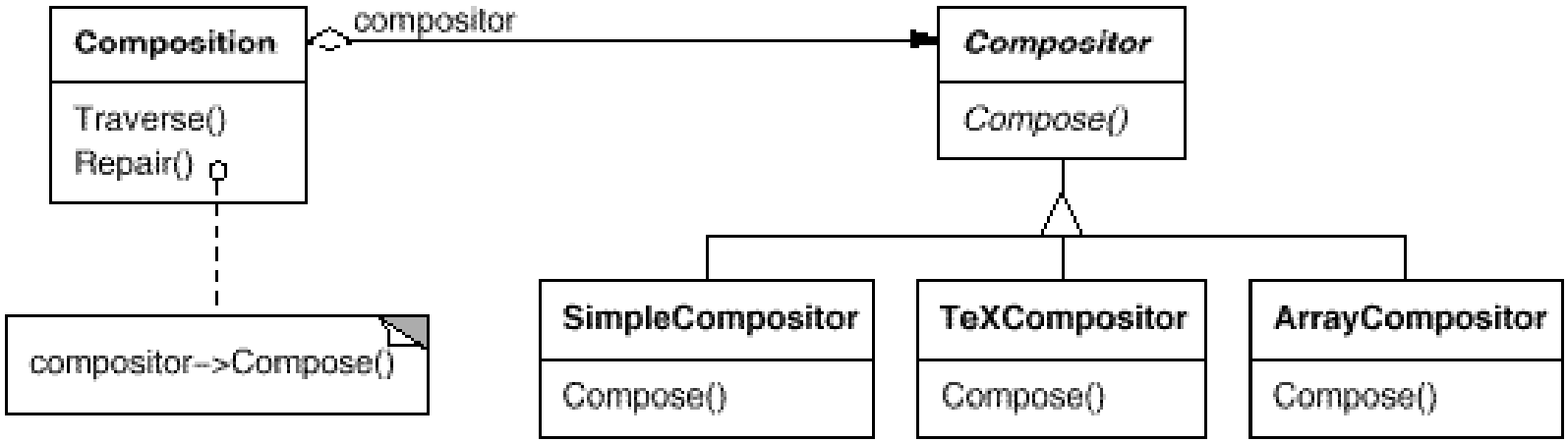
Consequences

- Abstract coupling between Subject and Observer.
- Support for broadcast communication.
- Unexpected updates.

Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.

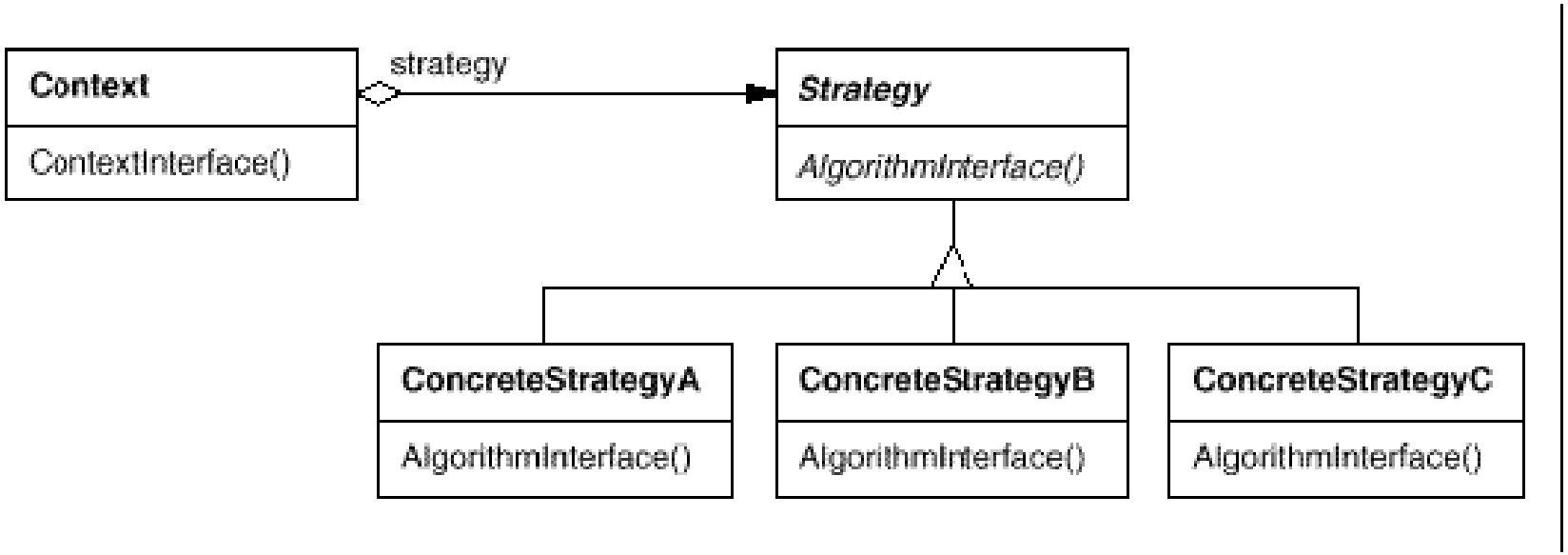
Motivation


Composition
Traverse()
Repair()
compositor
Compositor
Compose()
compositor->Compose()
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()

Applicability

- Use the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms [HO87].
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Participants

- Strategy (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implements the algorithm using the Strategy interface.
- Context (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Collaborations

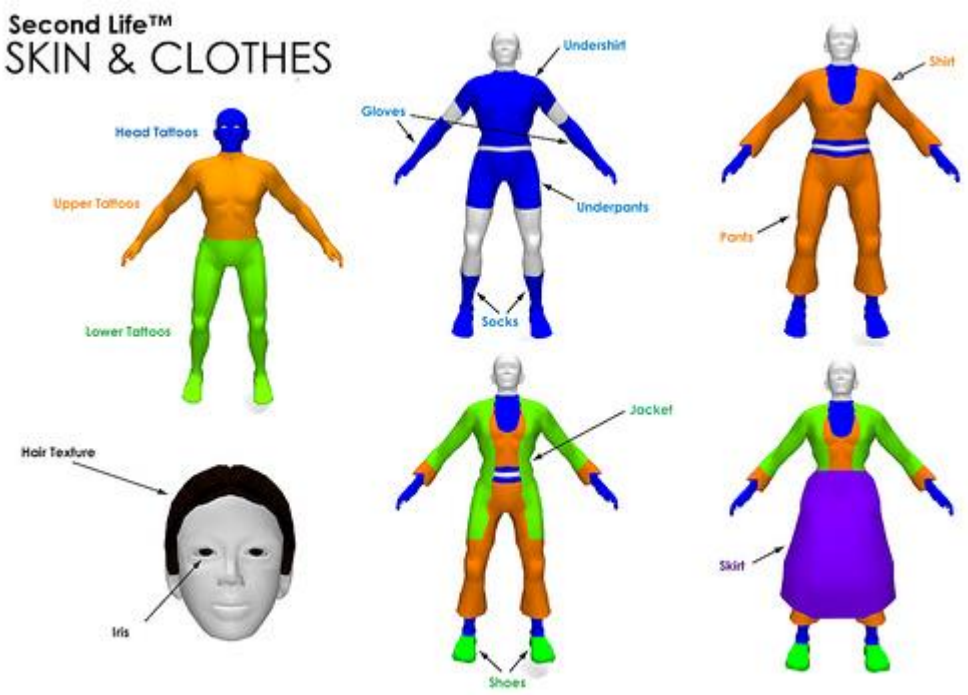
- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

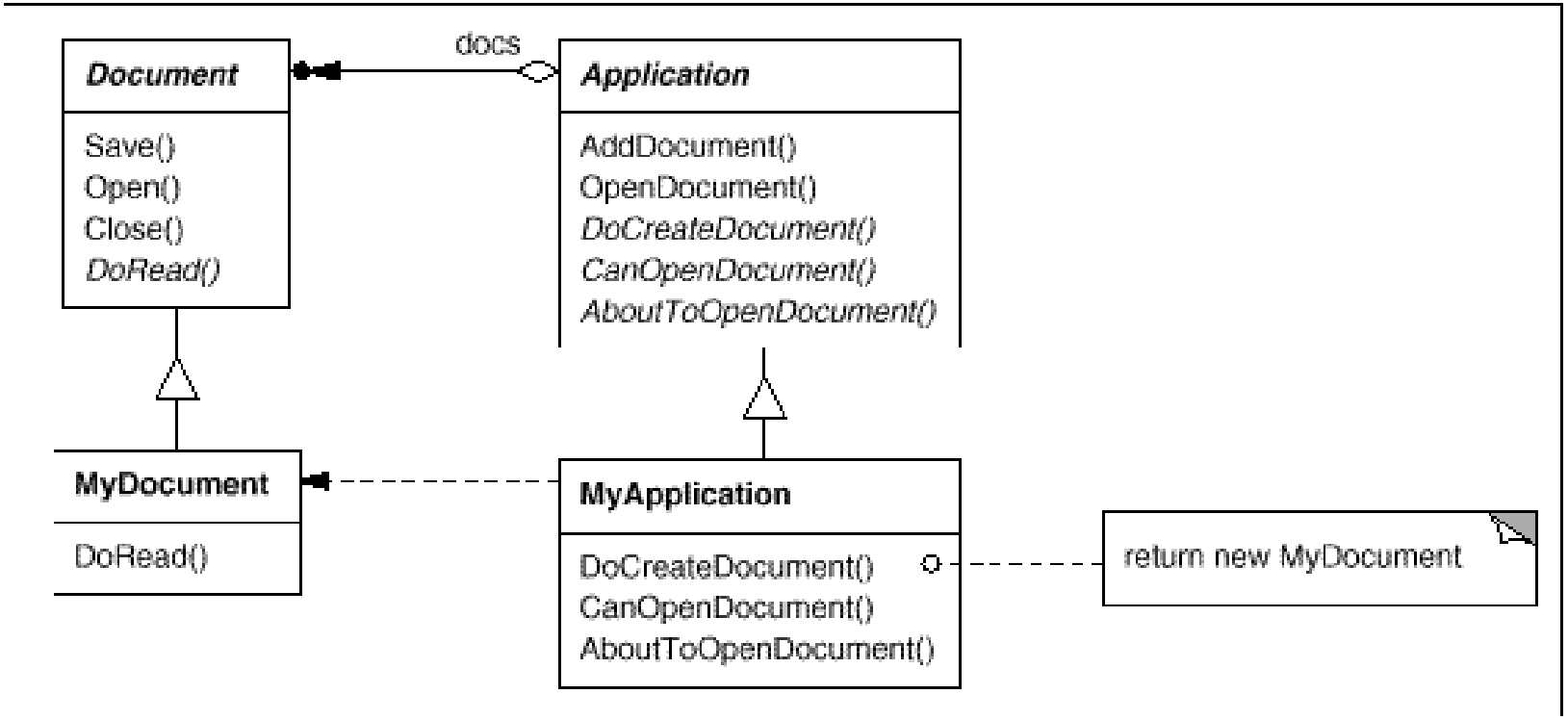
- Families of related algorithms.
- An alternative to subclassing.
- Strategies eliminate conditional statements.
- A choice of implementations.
- Clients must be aware of different Strategies.
- Communication overhead between Strategy and Context.
- Increased number of objects.

Template Method

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



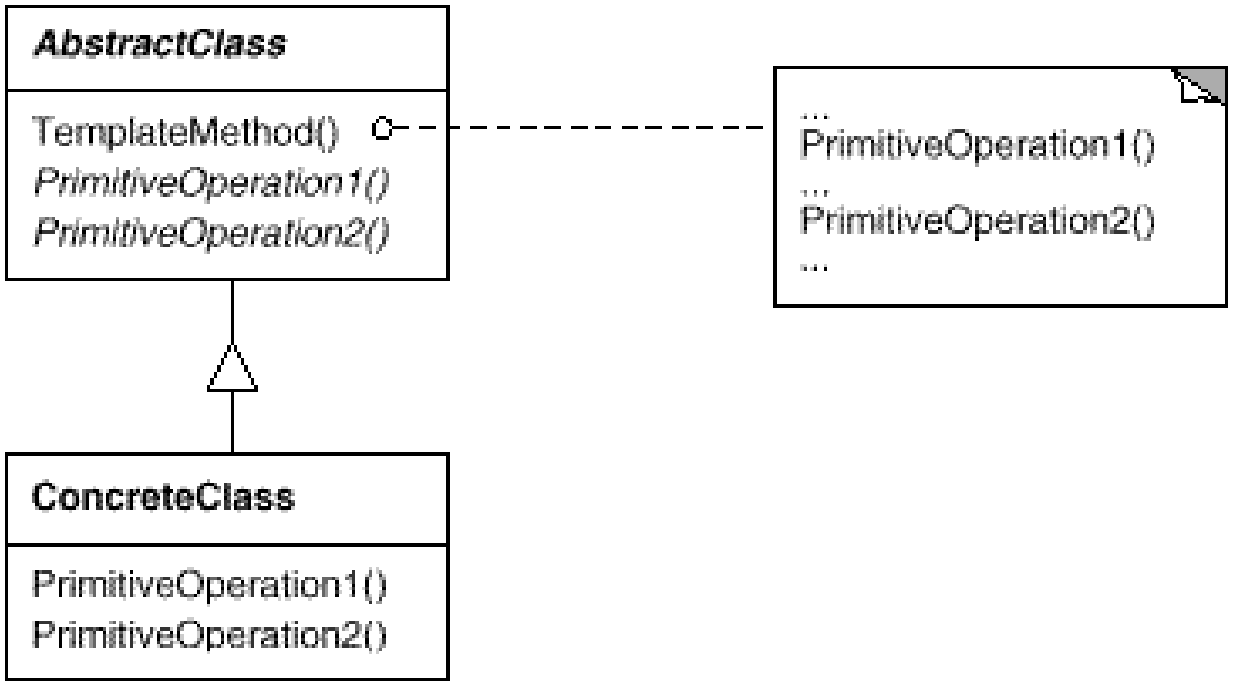
Motivation


Document
Save()
Open()
Close()
DoRead()
docs
Application
AddDocument()
OpenDocument()
DoCreateDocument()
CanOpenDocument()
AboutToOpenDocument()
MyDocument
DoRead()
MyApplication
DoCreateDocument()
CanOpenDocument()
AboutToOpenDocument()
return new MyDocument

Applicability

- The Template Method pattern should be used
  - 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. This is a good example of "refactoring to generalize" as described by Opdyke and Johnson [OJ93]. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
  - to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Structure


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()

Participants

- AbstractClass (Application)
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- ConcreteClass (MyApplication)
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

- Template methods lead to an inverted control structure that's sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you" [Swe85]. This refers to how a parent class calls the operations of a subclass and not the other way around.
- It's important for template methods to specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

How to select a design pattern

- Consider how design patterns solve design problems.
- Scan Intent sections.
- Study how patterns interrelate.

Patterns organisation

- Creational patterns
- Structural patterns
- Behavioral patterns

Creational patterns

- Abstract Factory
  - families of product objects
- Builder
  - how a composite object gets created
- Factory Method
  - subclass of object that is instantiated
- Prototype
  - class of object that is instantiated
- Singleton
  - the sole instance of a class

Structural patterns

- Adapter
  - interface to an object
- Bridge
  - implementation of an object
- Composite
  - structure and composition of an object
- Decorator
  - responsibilities of an object without subclassing
- Facade
  - interface to a subsystem
- Flyweight
  - storage costs of objects
- Proxy
  - how an object is accessed; its location

Behavioral patterns

- Chain of Responsibility
  - object that can fulfill a request
- Command
  - when and how a request is fulfilled
- Interpreter
  - grammar and interpretation of a language
- Iterator
  - how an aggregate's elements are accessed, traversed
- Mediator
  - how and which objects interact with each other
- Memento
  - what private information is stored outside an object, and when

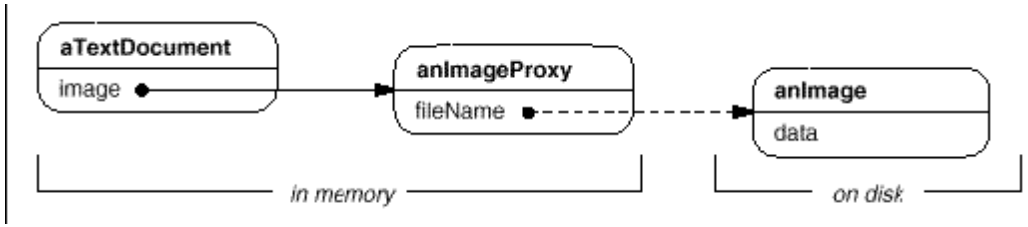
Behavioral patterns

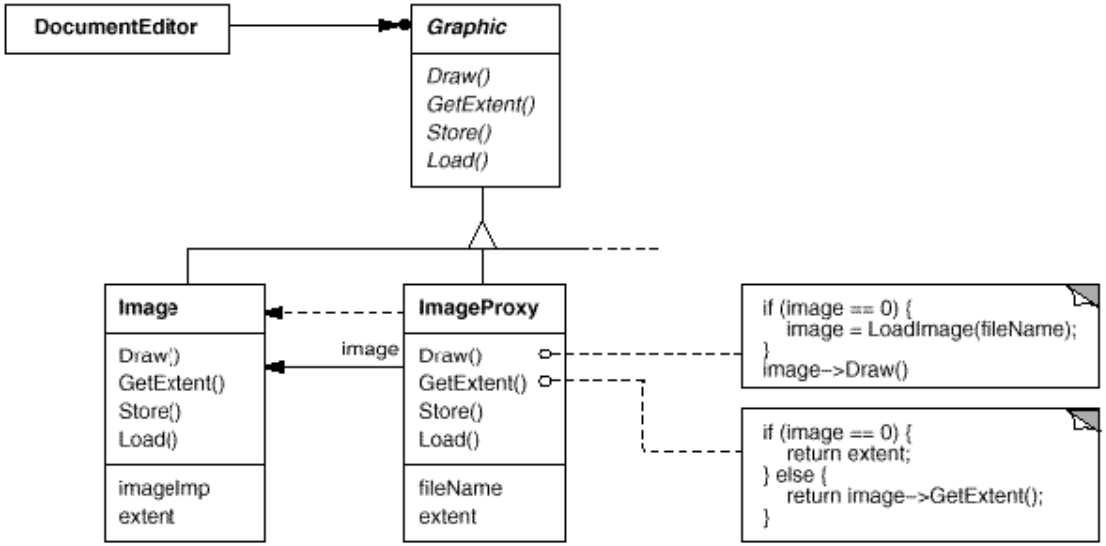
- Observer
  - number of objects that depend on another object; how the dependent objects stay up to date
- State
  - states of an object
- Strategy
  - an algorithm
- Template Method
  - steps of an algorithm
- Visitor
  - operations that can be applied to object(s) without changing their class(es)

Proxy

- Provide a surrogate or placeholder for another object to control access to it.

Motivation

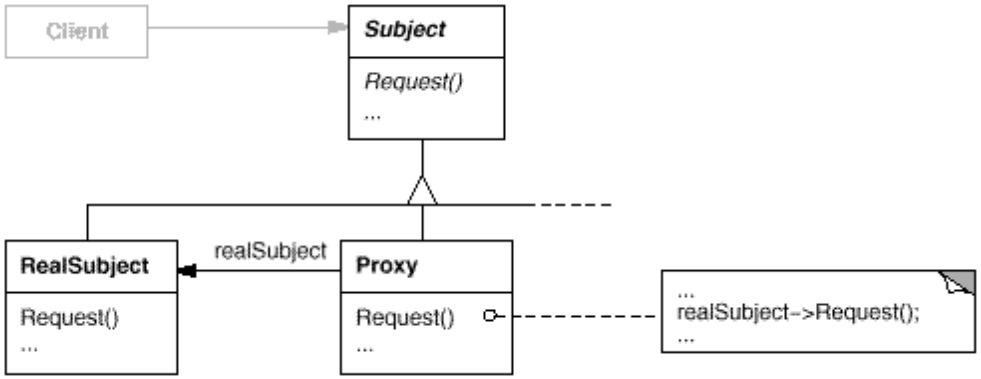

aTextDocument
image
anImageProxy
fileName
anImage
data
in memory
on disk

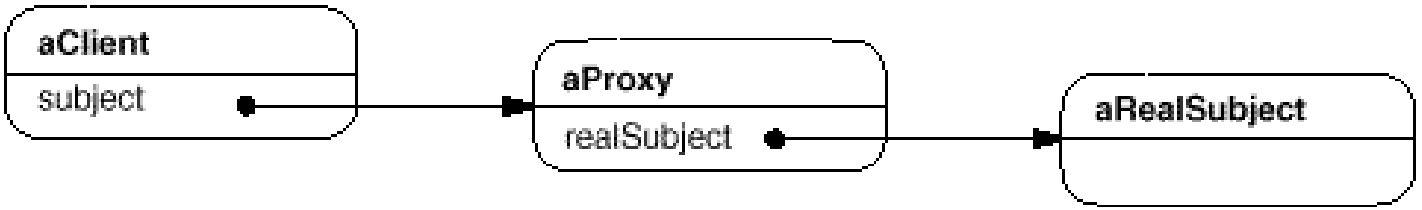

DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent;
} else {
return image->GetExtent();
}

Applicability

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
- A protection proxy controls access to the original object.
- A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed

Structure


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...


aClient
subject
aProxy
realSubject
aRealSubject

Participants

- Proxy (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
- Subject (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject (Image)
  - defines the real object that the proxy represents.

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

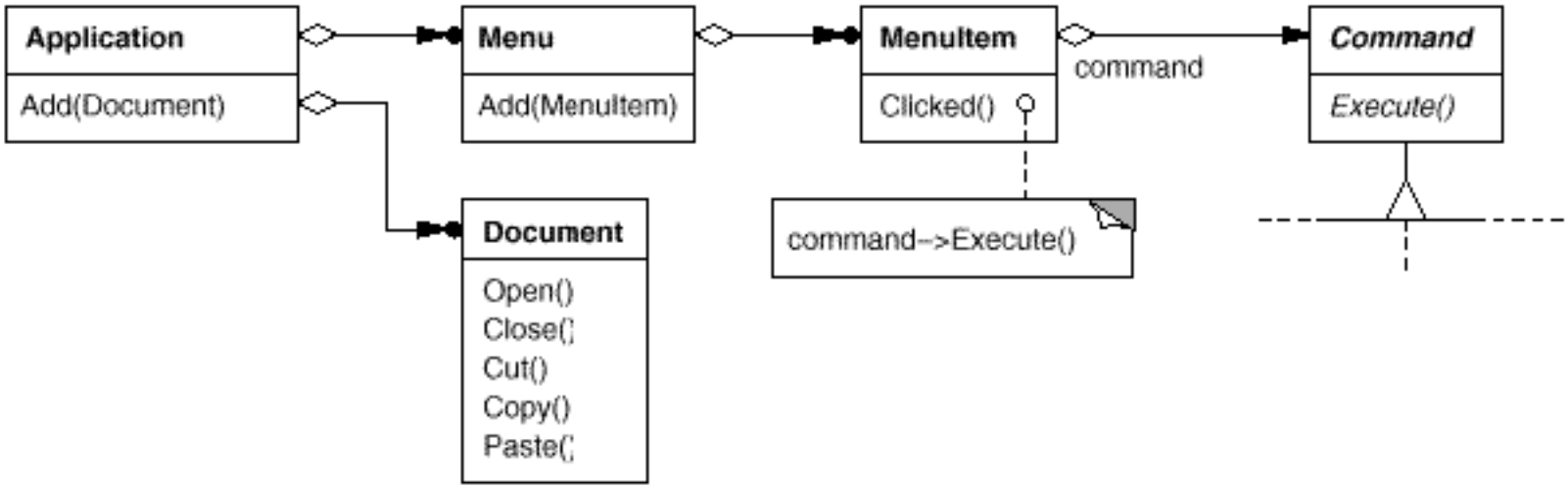
Consequences

- The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:
  - 1. A remote proxy can hide the fact that an object resides in a different address space.
  - 2. A virtual proxy can perform optimizations such as creating an object on demand.
  - 3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

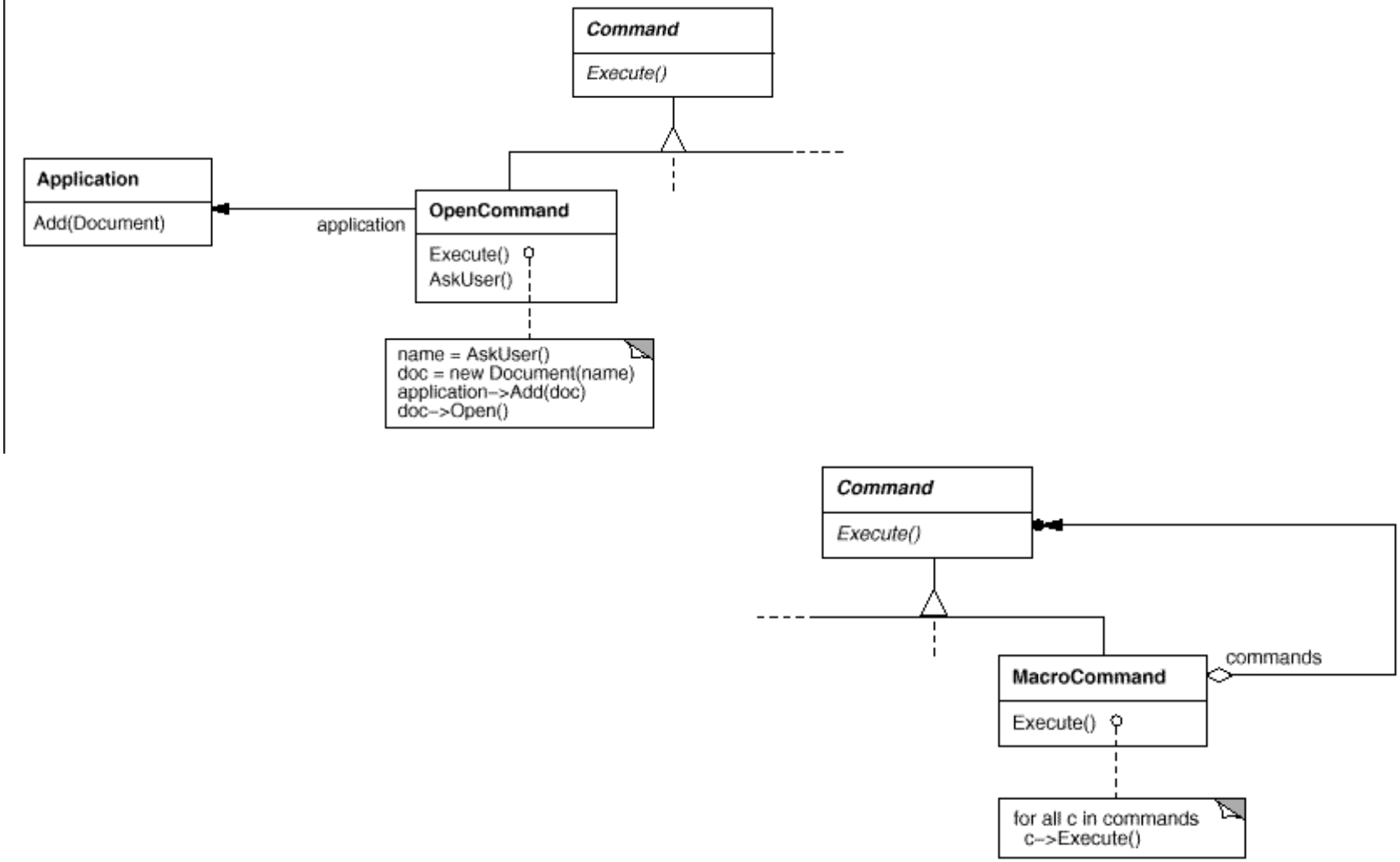
Command

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Motivation


Application
Add(Document)
Menu
Add(MenuItem)
MenuItem
Clicked()
command
Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
command->Execute()

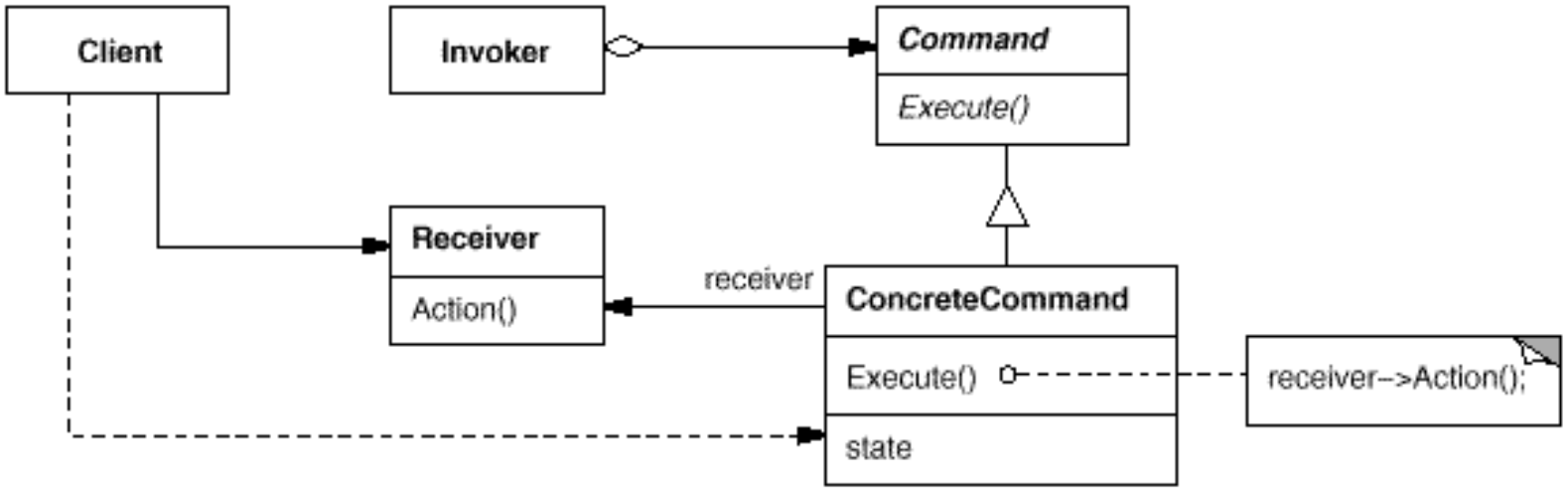
Motivation (2)



Applicability

- Use the Command pattern when you want to
  - parameterize objects by an action to perform, as MenuItem objects did above.
  - specify, queue, and execute requests at different times.
  - support undo. The Command's Execute operation can store state for reversing its effects in the command
  - support logging changes so that they can be reapplied in case of a system crash.
  - structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions.

Structure


Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver->Action();

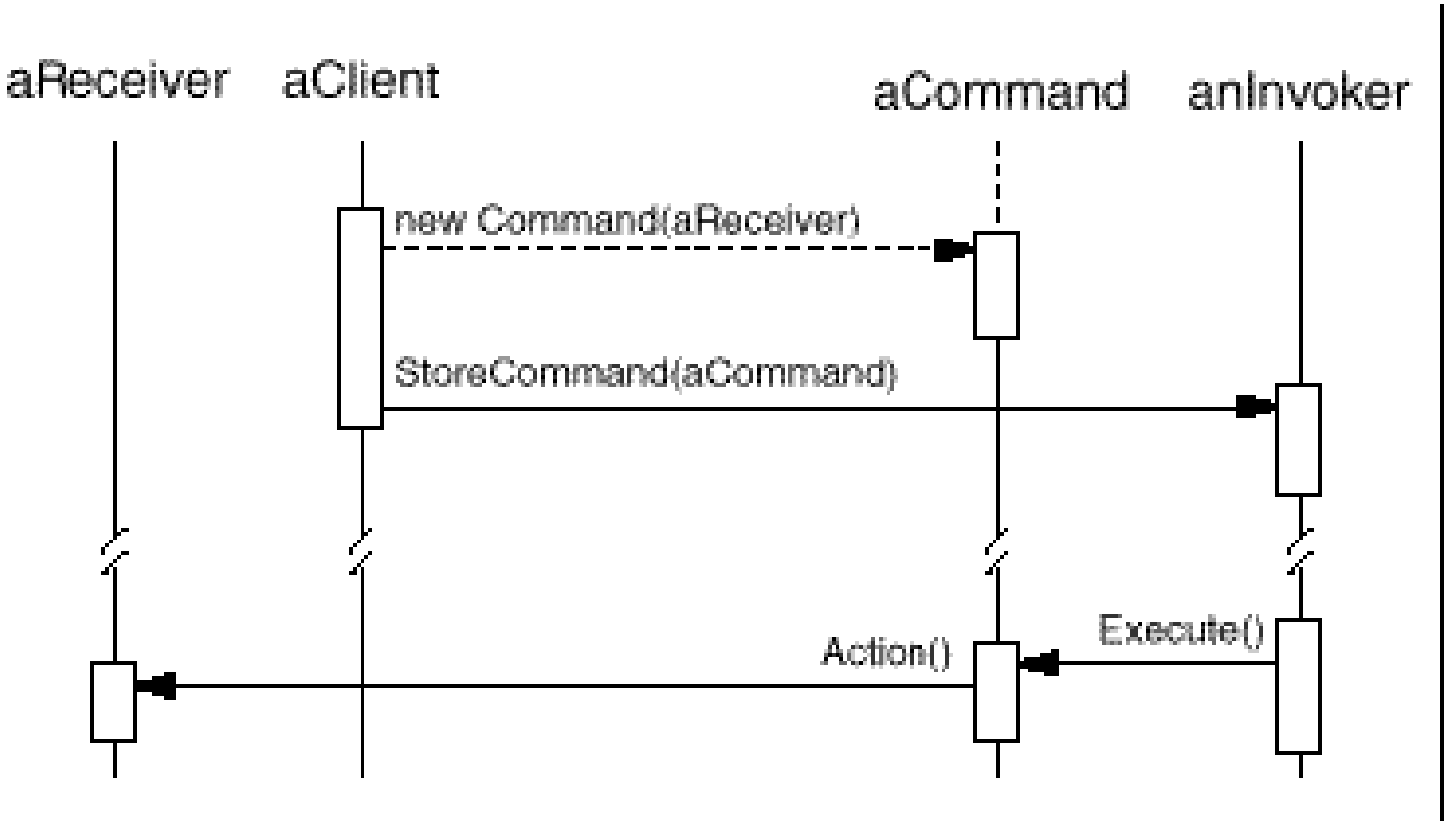
Participants

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

Collaborations

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

Collaborations



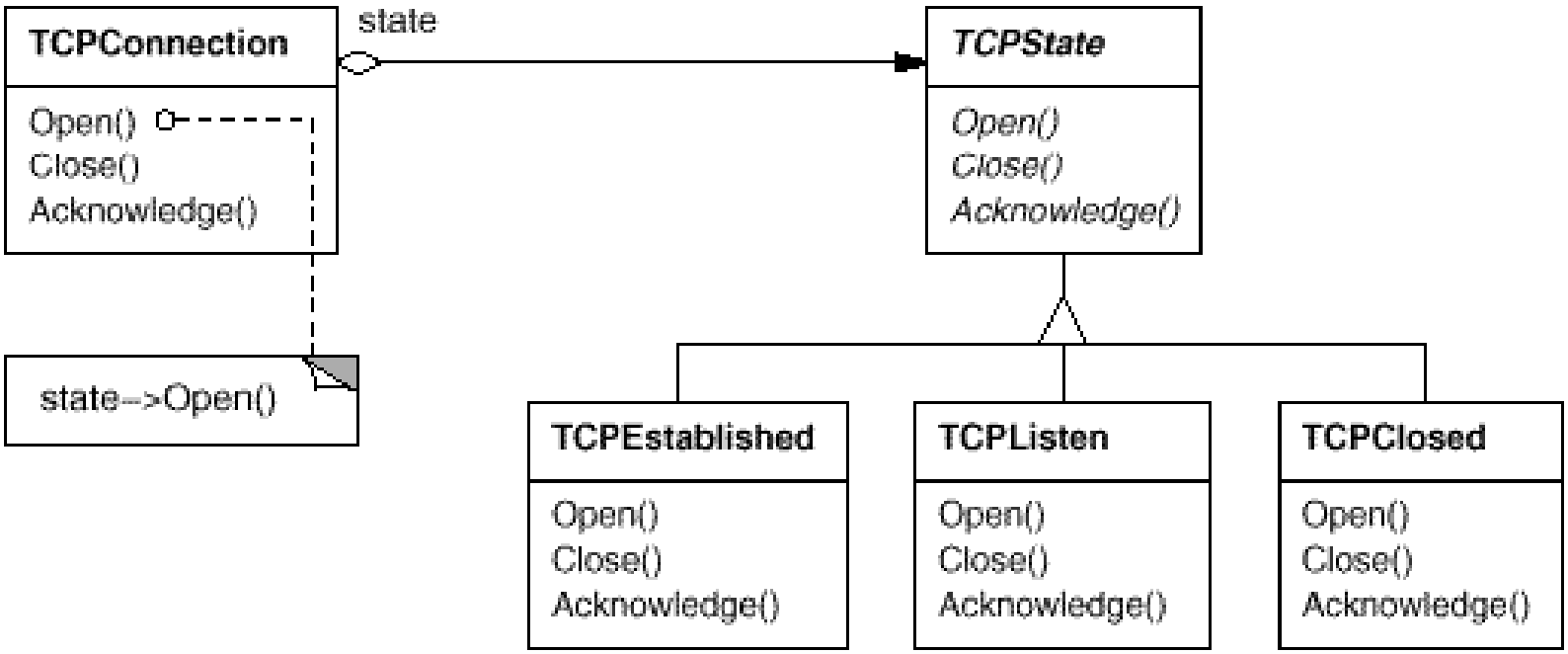
Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- You can assemble commands into a composite command. An example is the MacroCommand class described earlier.
- It's easy to add new Commands, because you don't have to change existing classes.

State

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

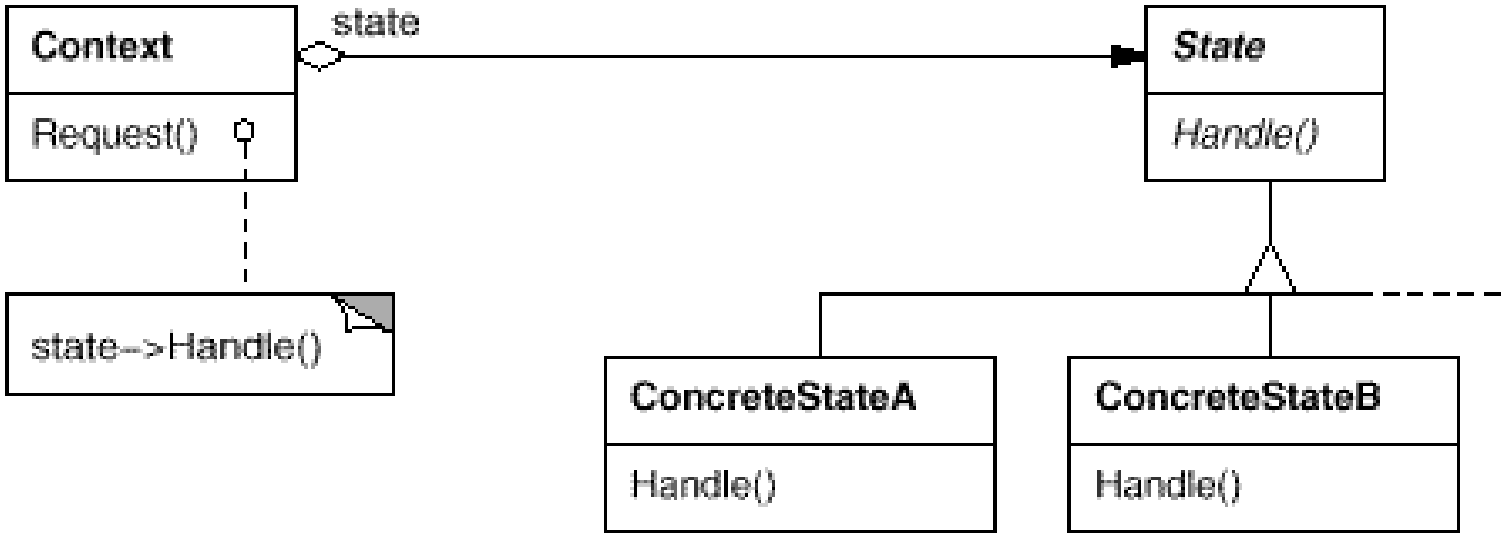
Motivation


TCPConnection
Open()
Close()
Acknowledge()
state
TCPState
Open()
Close()
Acknowledge()
state->Open()
TCPEstablished
Open()
Close()
Acknowledge()
TCPListen
Open()
Close()
Acknowledge()
TCPClosed
Open()
Close()
Acknowledge()

Applicability

- Use the State 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.

Structure


Context
Request()
state
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

Participants

- Context (TCPConnection)
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state.
- State (TCPState)
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)
  - each subclass implements a behavior associated with a state of the Context.

Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
  - A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
  - Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
  - Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

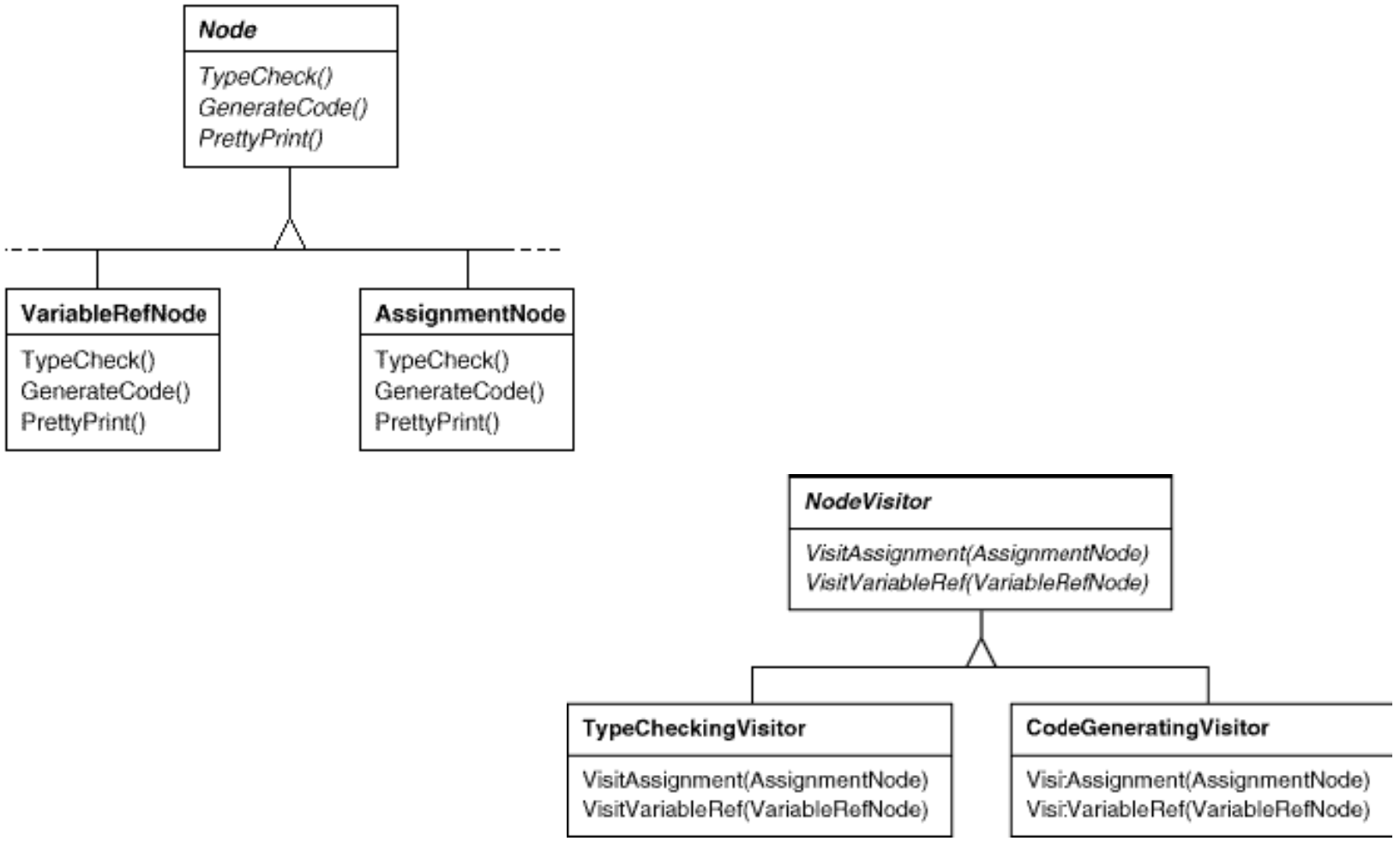
Consequences

- It localizes state-specific behavior and partitions behavior for different states.
- It makes state transitions explicit.
- State objects can be shared.

Visitor

- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation

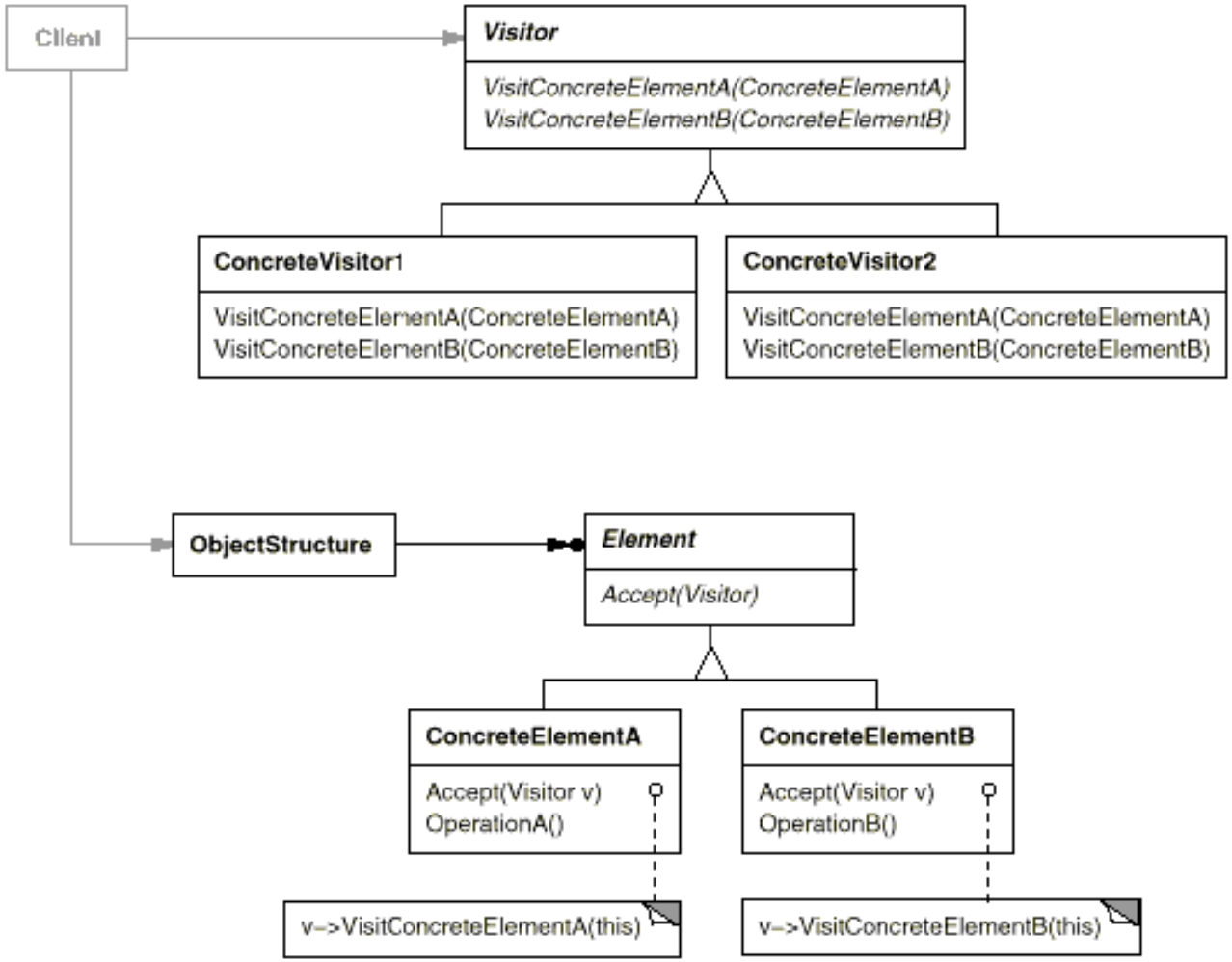

Node
TypeCheck()
GenerateCode()
PrettyPrint()
VariableRefNode
TypeCheck()
GenerateCode()
PrettyPrint()
AssignmentNode
TypeCheck()
GenerateCode()
PrettyPrint()
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

Applicability

- Use the Visitor pattern when
  - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

GoF

Structure

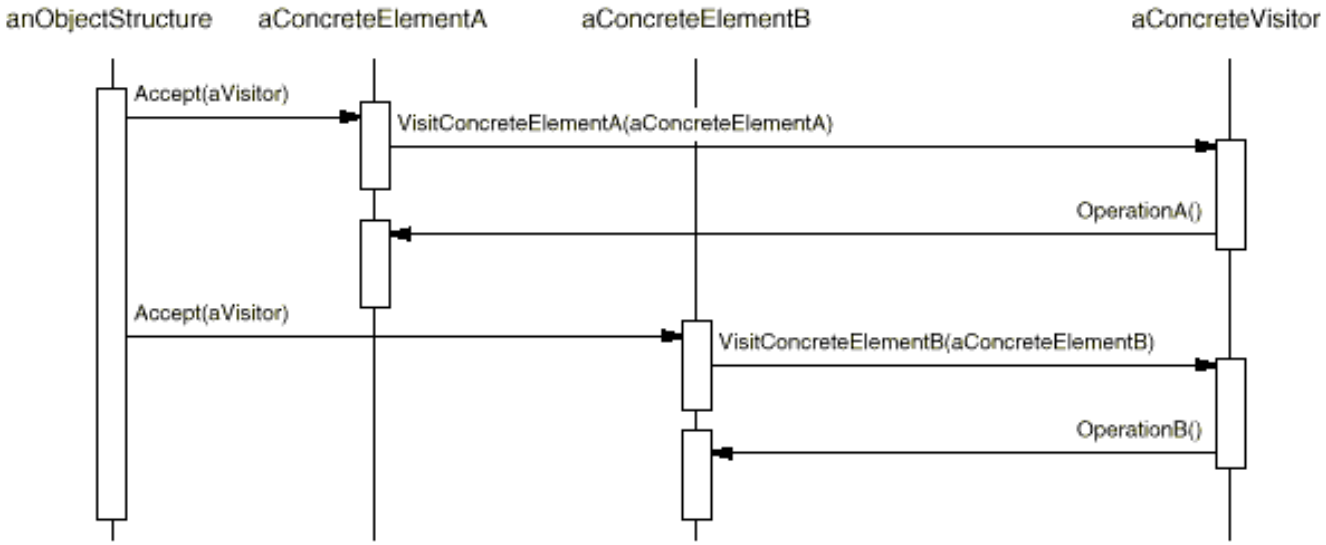

Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure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

Participants

- Visitor (NodeVisitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
- ConcreteVisitor (TypeCheckingVisitor)
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- Element (Node)
  - defines an Accept operation that takes a visitor as an argument.
- ConcreteElement (AssignmentNode, VariableRefNode)
  - implements an Accept operation that takes a visitor as an argument.
- ObjectStructure (Program)
  - can enumerate its elements.
  - may provide a high-level interface to allow the visitor to visit its elements.
  - may either be a composite or a collection such as a list or a set.

Collaborations

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.



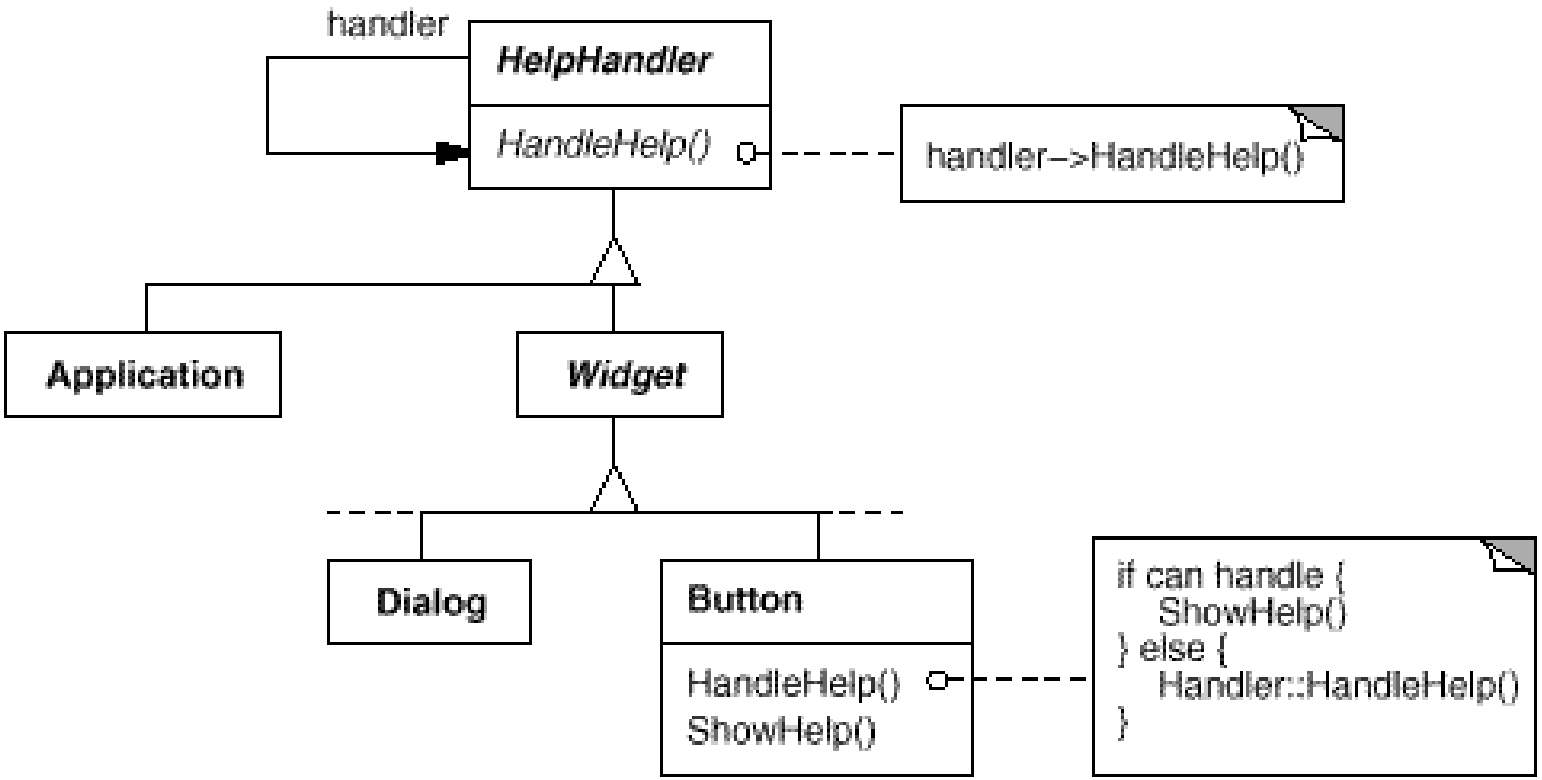
Consequences

- Visitor makes adding new operations easy.
- A visitor gathers related operations and separates unrelated ones.
- Adding new ConcreteElement classes is hard.
- Visiting across class hierarchies.
- Accumulating state.
- Breaking encapsulation.

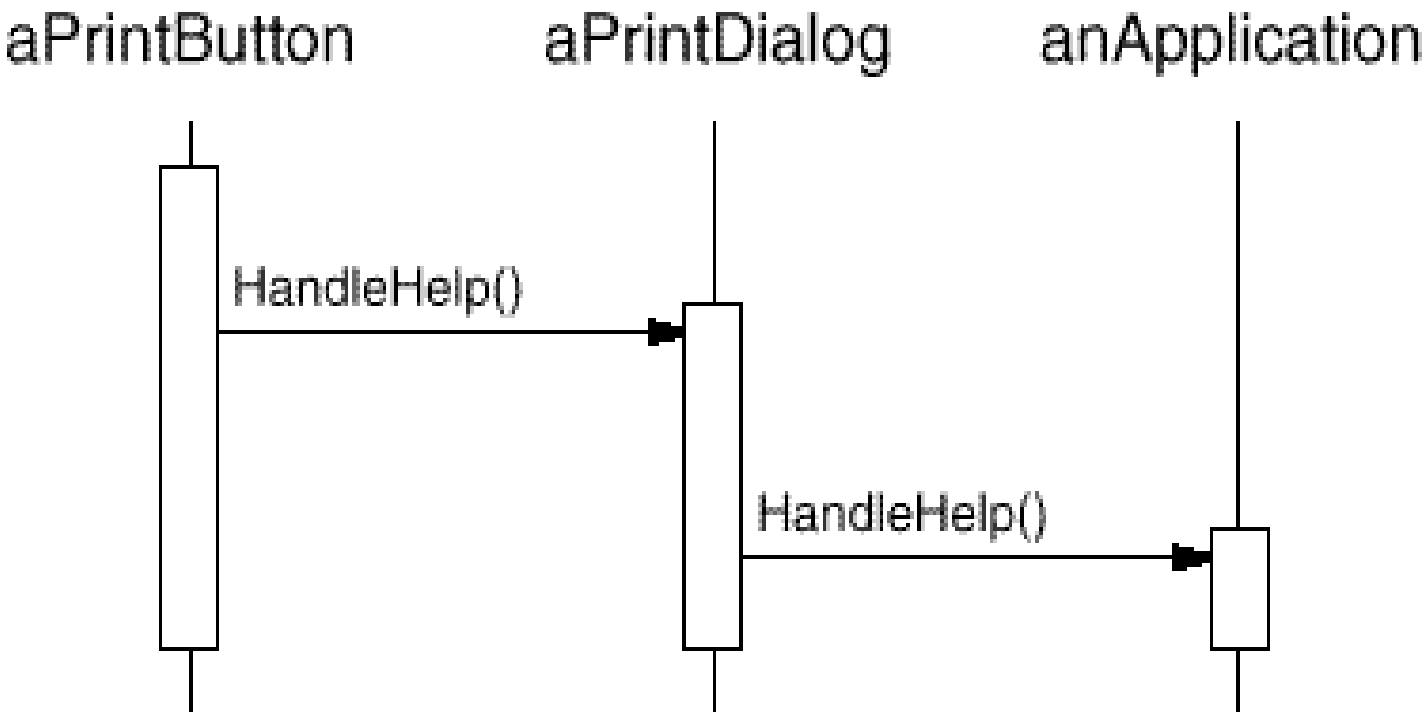
Chain of responsability

- 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.

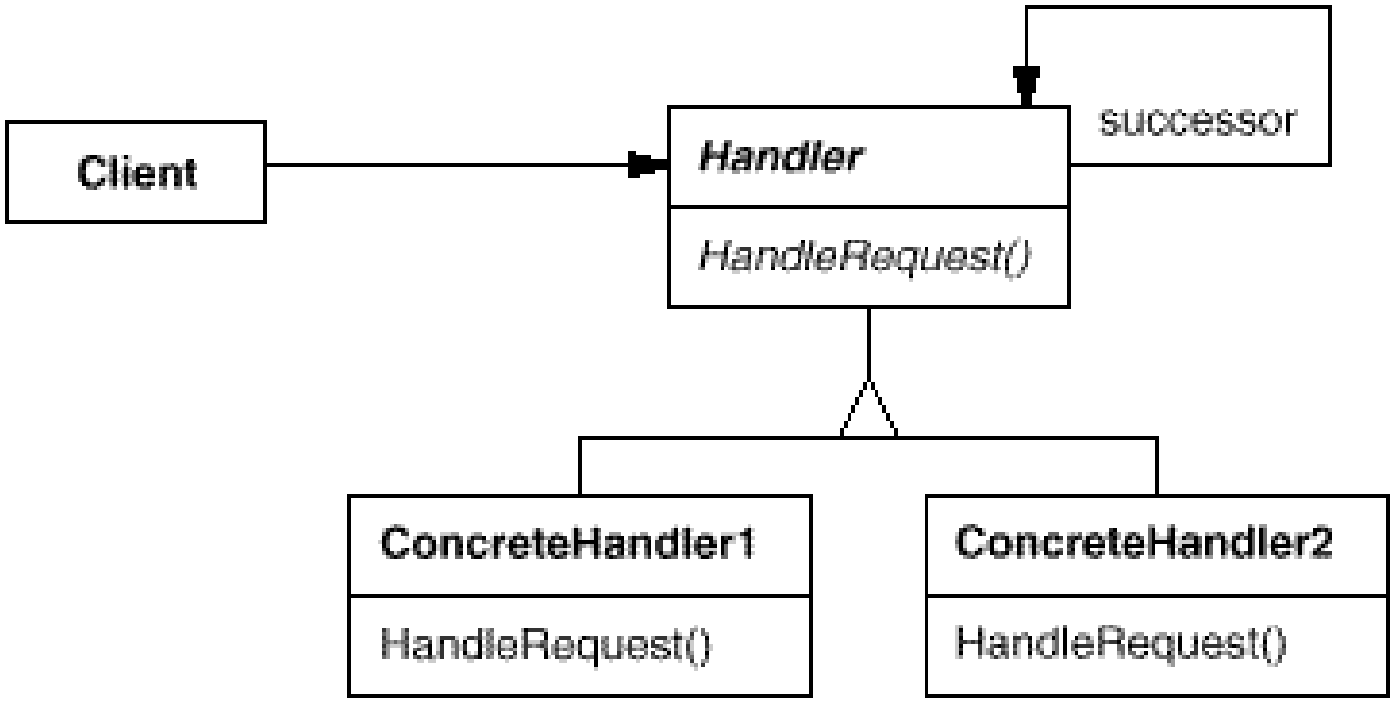
Motivation


handler
HelpHandler
HandleHelp()
handler->HandleHelp()
Application
Widget
Dialog
Button
HandleHelp()
ShowHelp()
if can handle {
ShowHelp()
} else {
Handler::HandleHelp()
}

Motivation


aPrintButton
aPrintDialog
anApplication
HandleHelp()
HandleHelp()

Structure


Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()

Participants

- Handler (HelpHandler)
  - defines an interface for handling requests.
  - (optional) implements the successor link.
- ConcreteHandler (PrintButton, PrintDialog)
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- Client
  - initiates the request to a ConcreteHandler object on the chain.

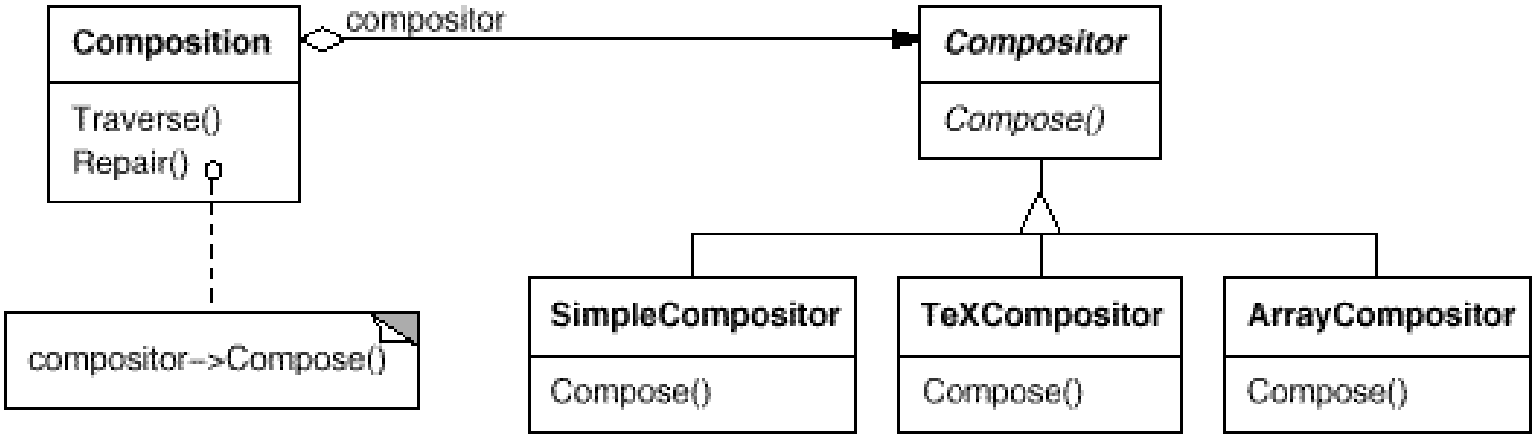
Example…

- Awt 1.0

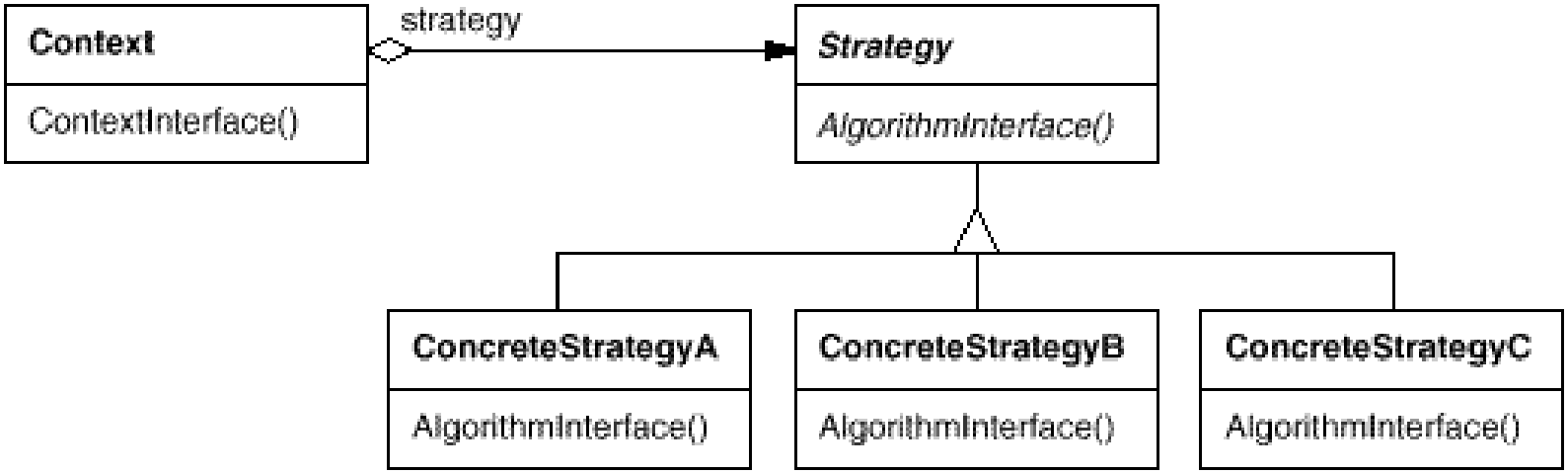
Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation


Composition
Traverse()
Repair()
compositor
Compositor
Compose()
compositor->Compose()
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()

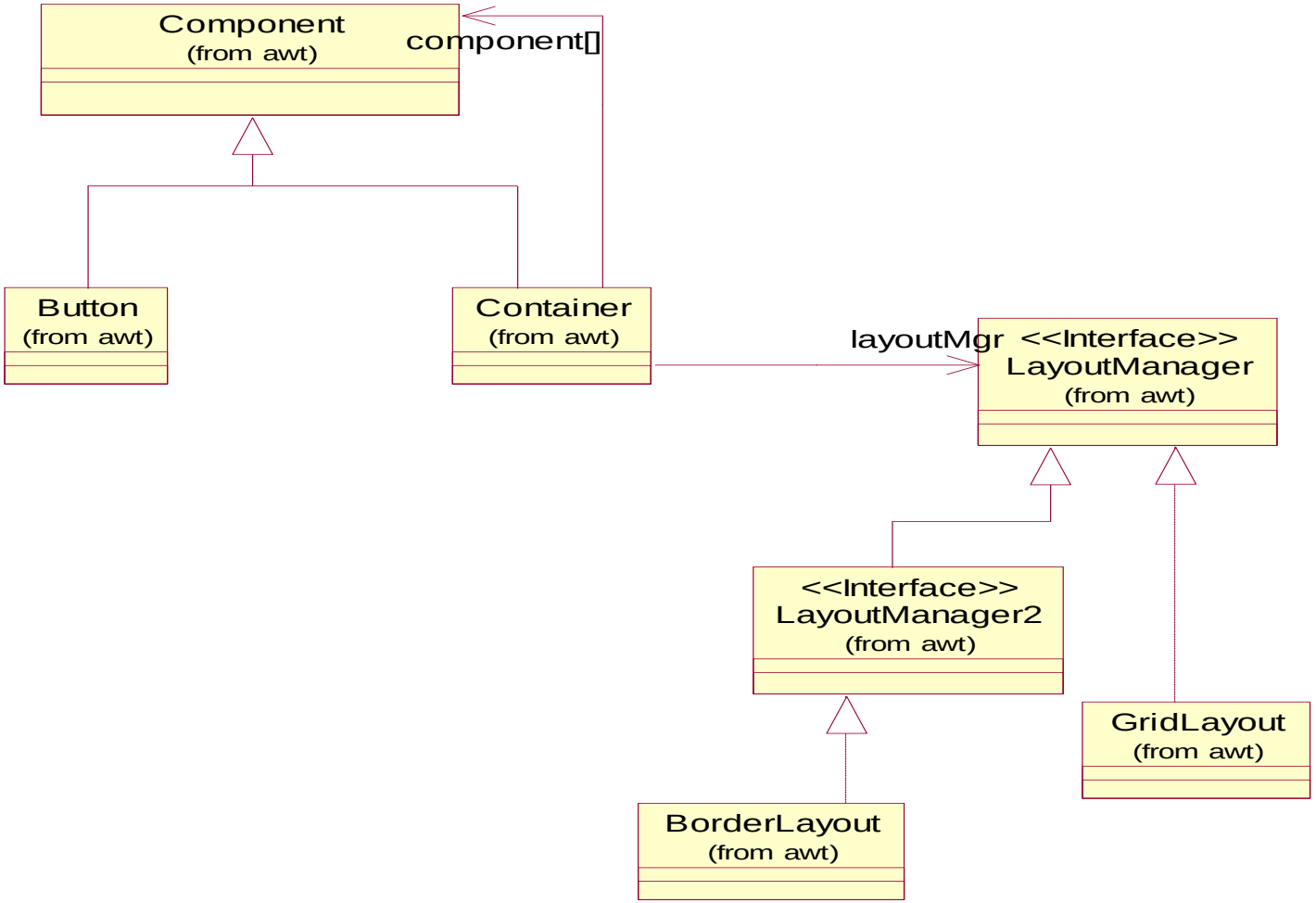
Strategy


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Participants

- Strategy (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implements the algorithm using the Strategy interface.
- Context (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

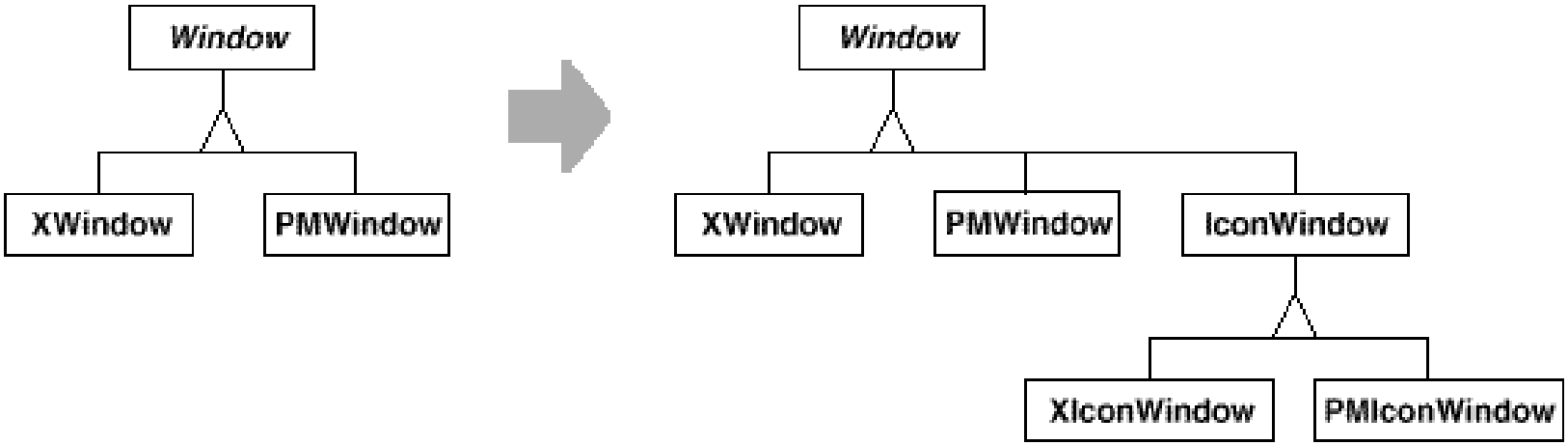
Strategy…


Component
(from awt)
component[]
Button
(from awt)
Container
(from awt)
layoutMgr
<<Interface>>
LayoutManager
(from awt)
<<Interface>>
LayoutManager2
(from awt)
GridLayout
(from awt)
BorderLayout
(from awt)

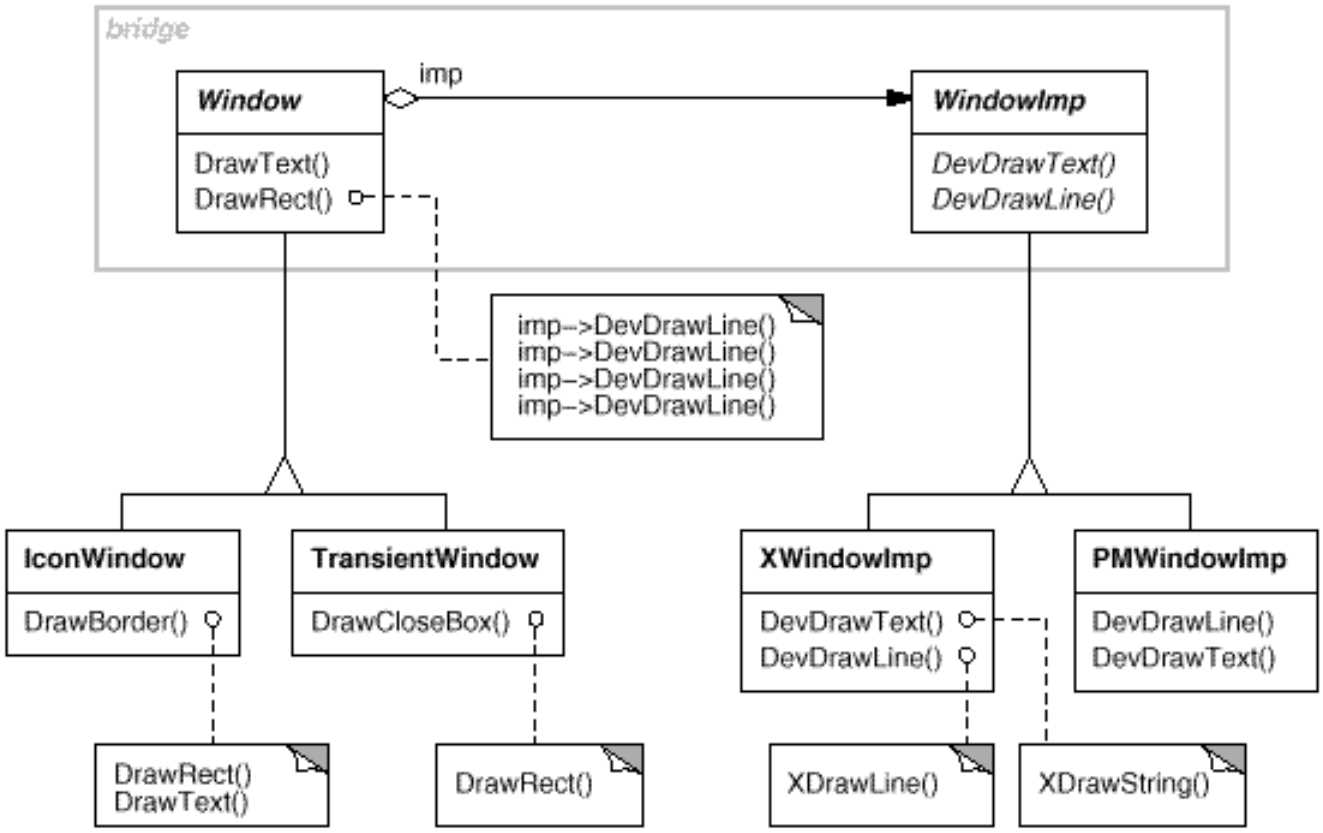
Bridge

- Decouple an abstraction from its implementation so that the two can vary independently.

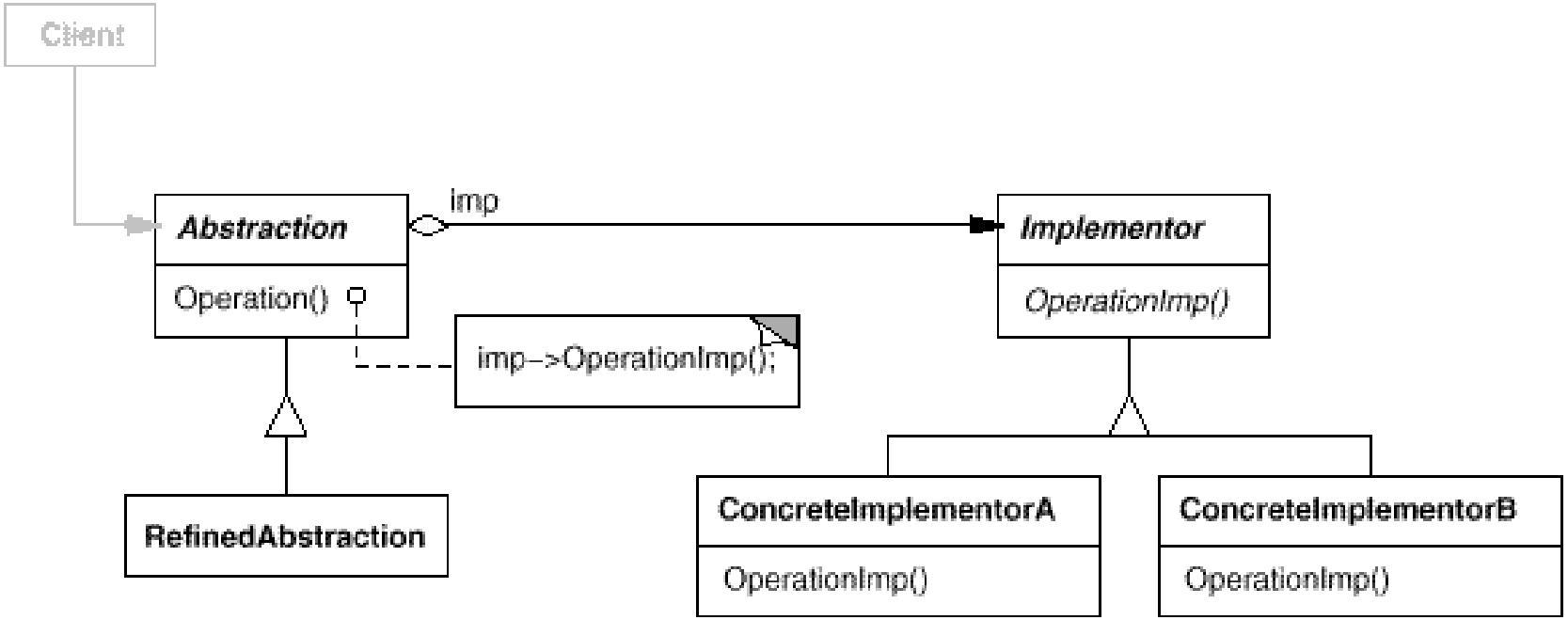
Bridge


Window
XWindow
PMWindow
Window
XWindow
PMWindow
IconWindow
XIconWindow
PMIconWindow

Bridge


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Bridge Structure…



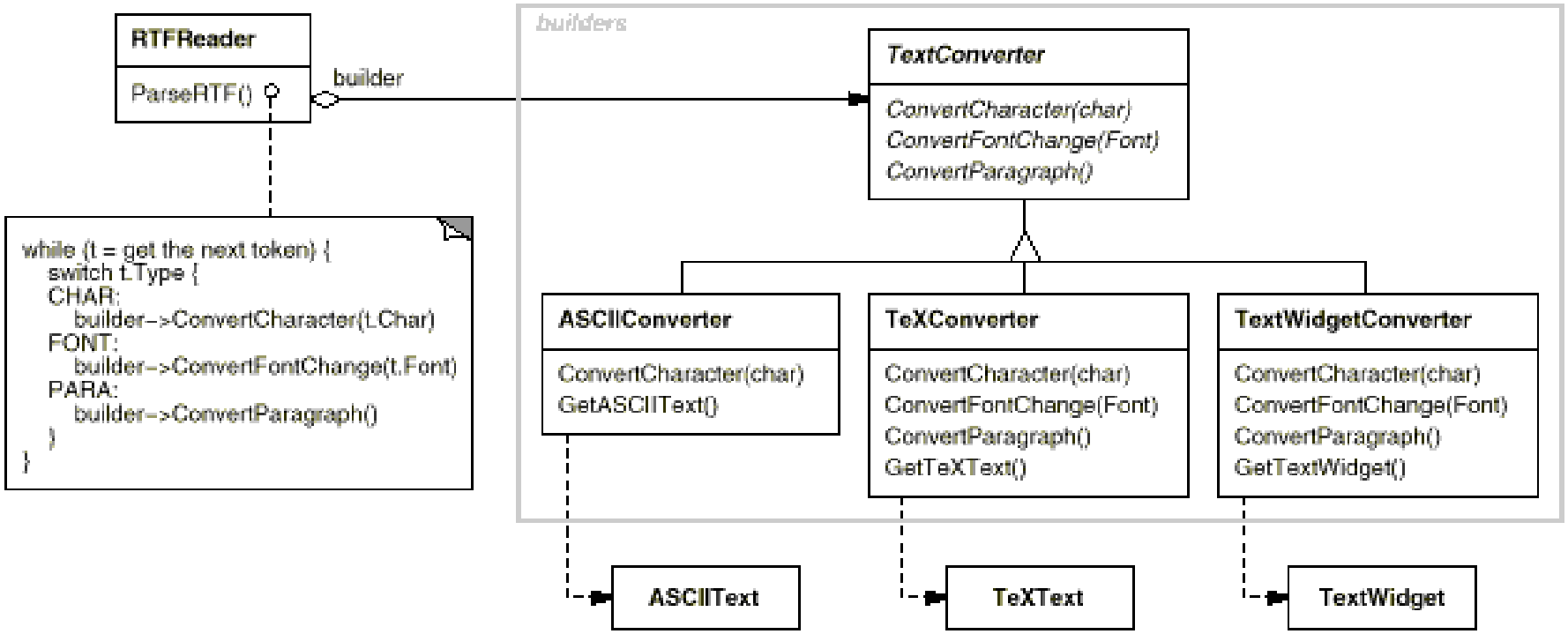
Bridge

- Decoupling interface and implementation
- Improved extensibility
- Hiding implementation details from clients

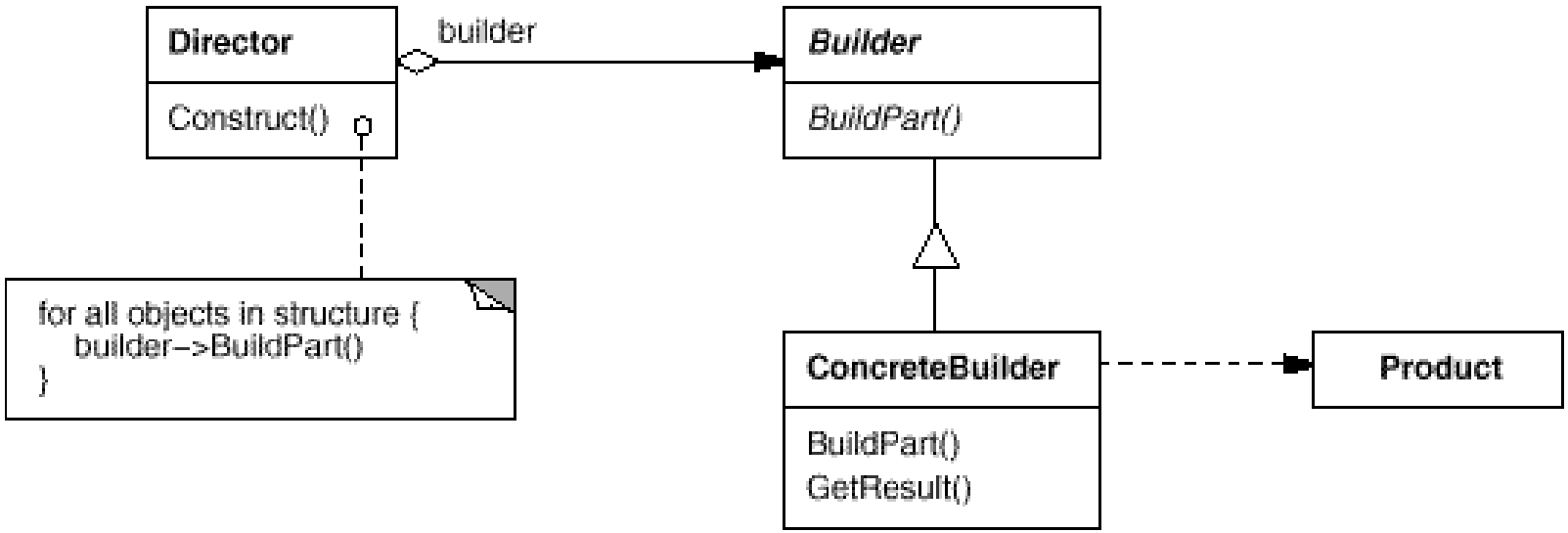
Builder

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

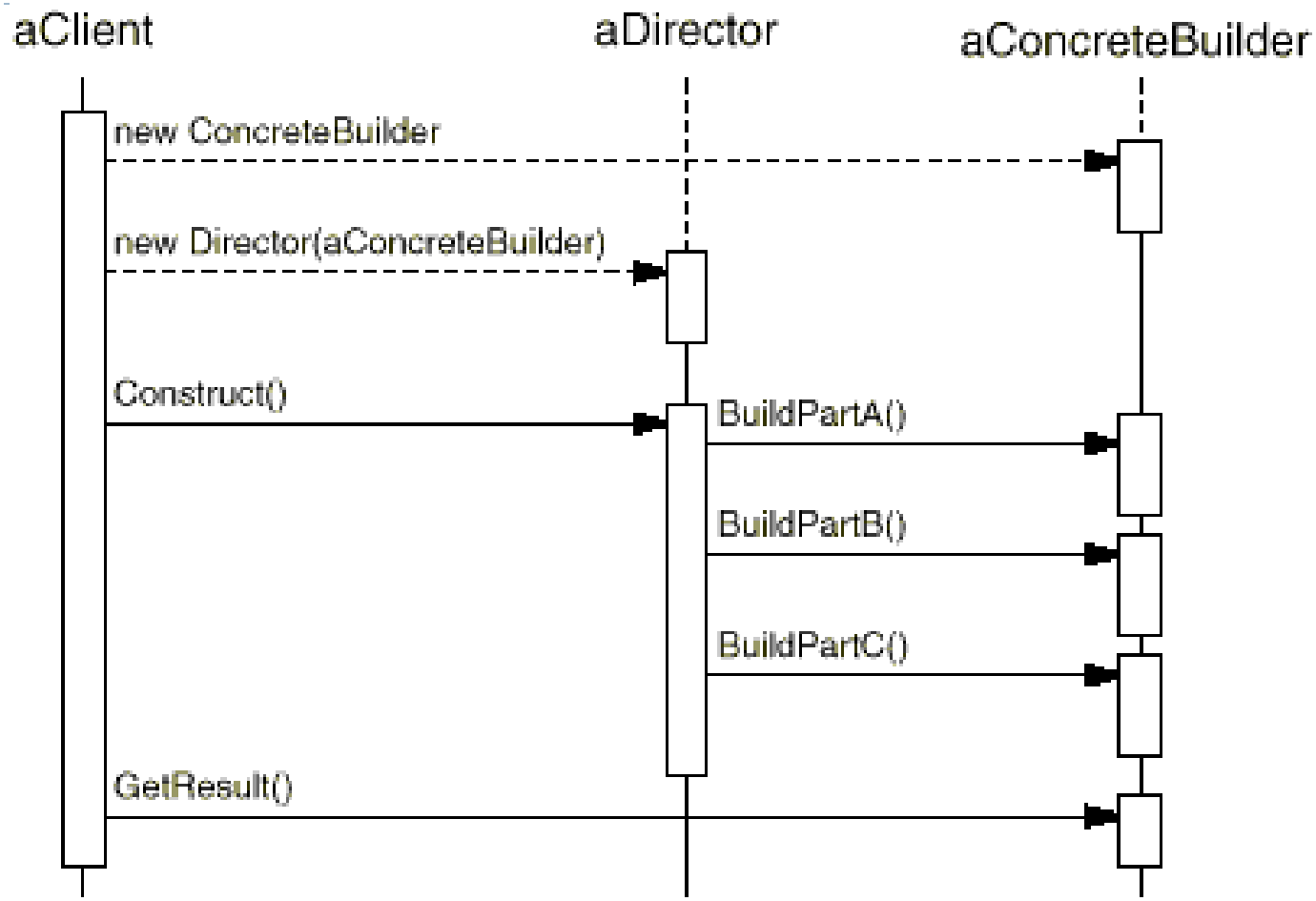
Builder


RTFReader
ParseRTF()
builder
while (t = get the next token) {
switch t.Type {
CHAR:
builder->ConvertCharacter(t.Char)
FONT:
builder->ConvertFontChange(t.Font)
PARA:
builder->ConvertParagraph()
}
}
builders
TextConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
ASCIIConverter
ConvertCharacter(char)
GetASCIIText()
TeXConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTeXText()
TextWidgetConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()
ASCIIText
TeXText
TextWidget

Builder Structure...



Builder


aClient
aDirector
aConcreteBuilder
new ConcreteBuilder
new Director(aConcreteBuilder)
Construct()
BuildPartA()
BuildPartB()
BuildPartC()
GetResult()

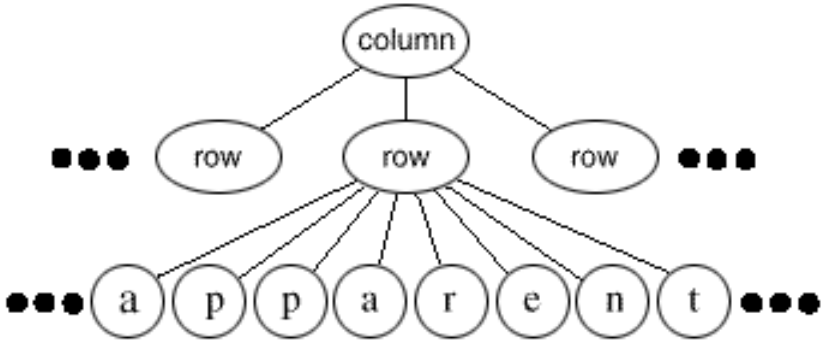
Builder Consequences

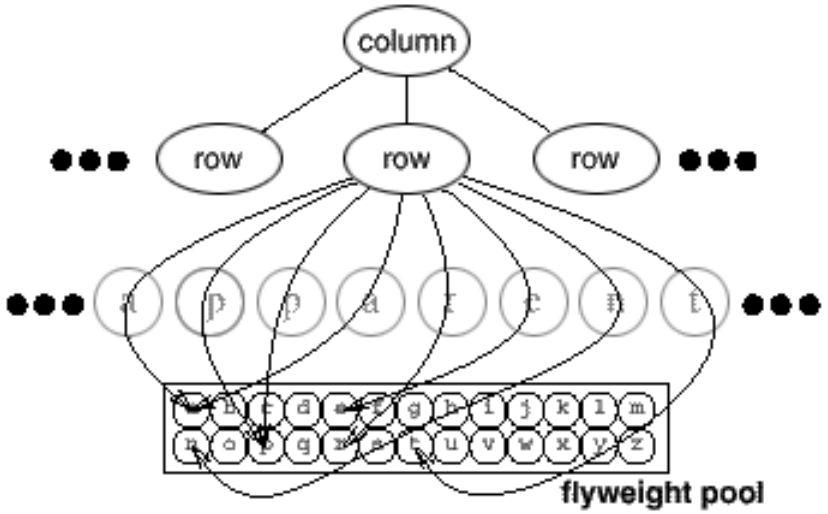
- It lets you vary a product's internal representation
- It isolates code for construction and representation
- It gives you finer control over the construction process

FlyWeight

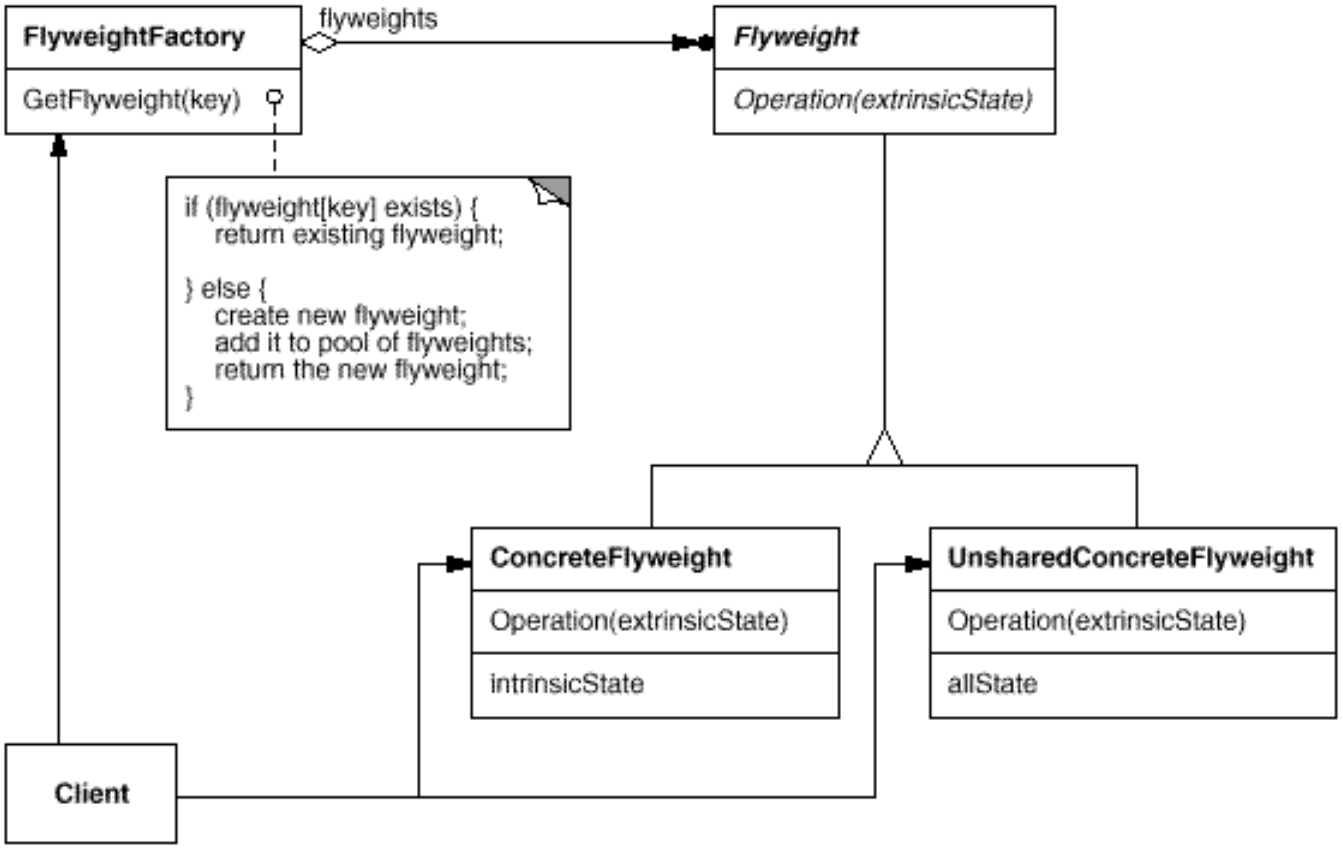
- Use sharing to support large numbers of fine-grained objects efficiently.

FlyWeight

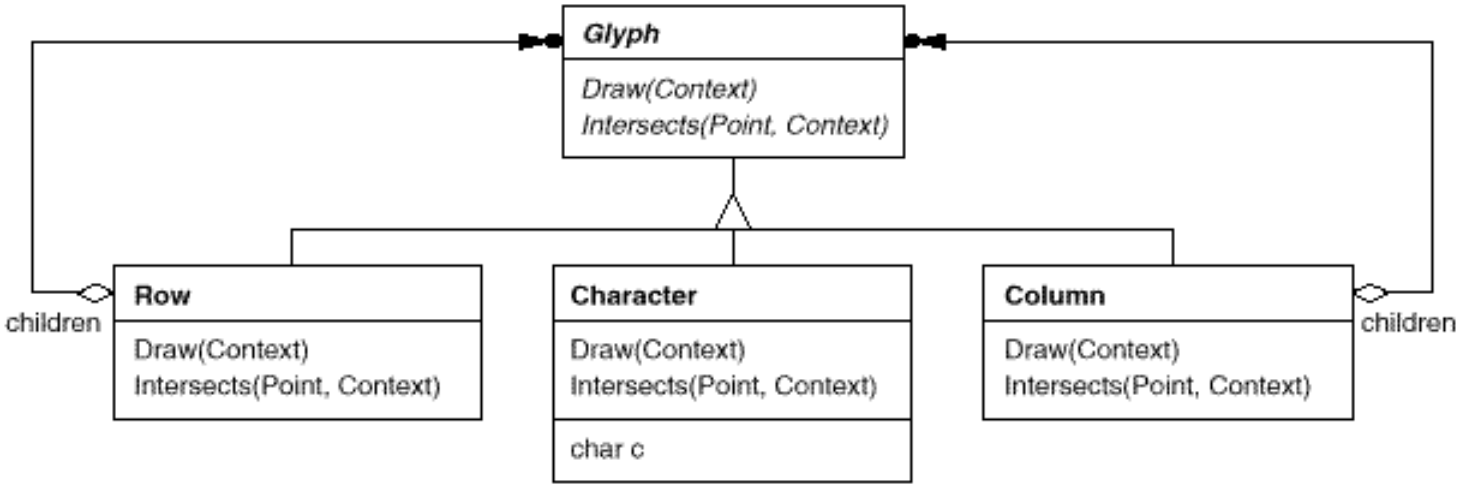

column
row
row
row
a
p
p
a
r
e
n
t


column
row
row
row
flyweight pool

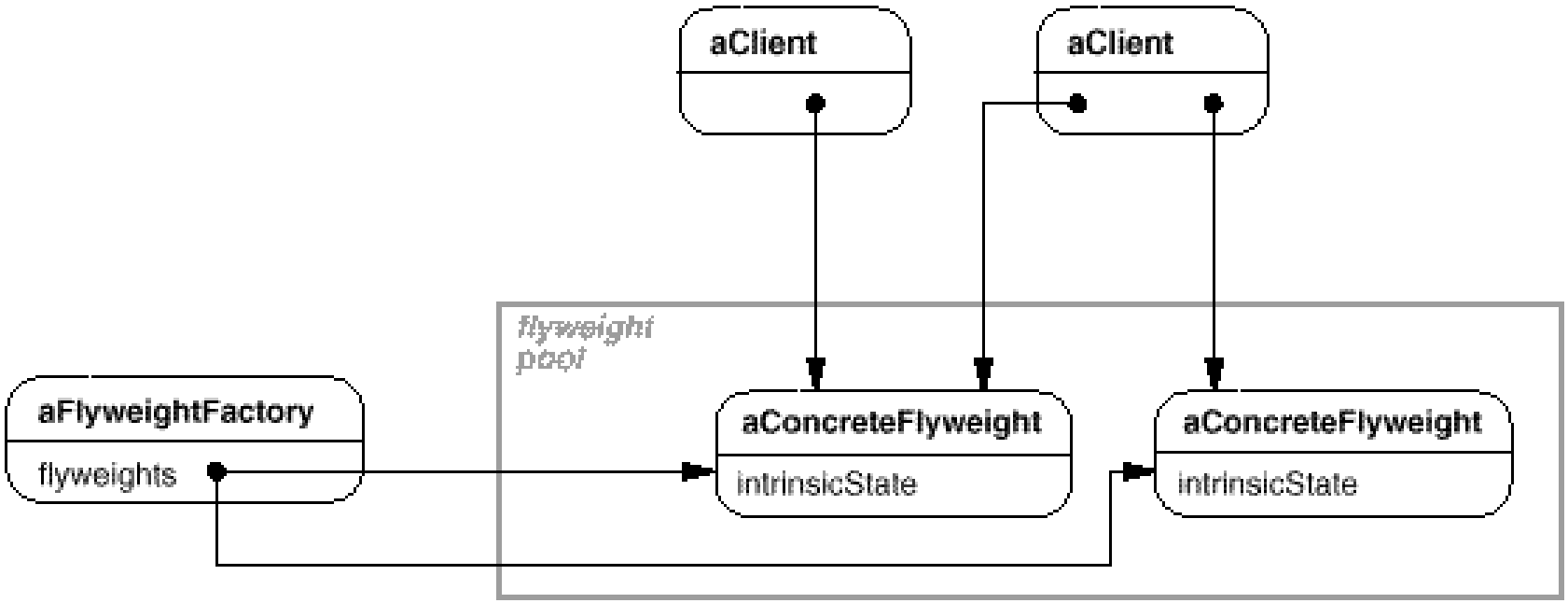
Flyweight: Structure



Flyweight example



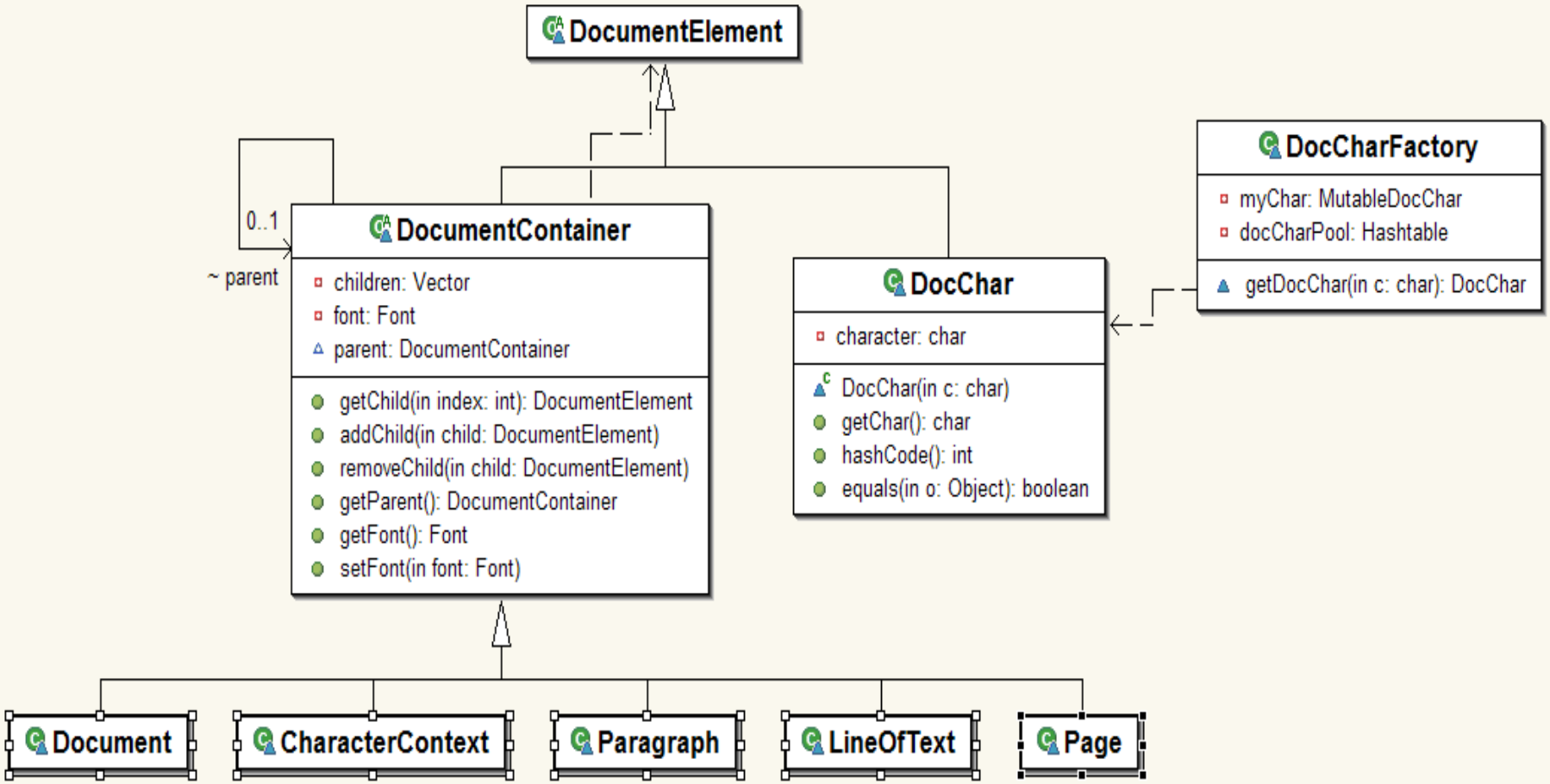
Flyweight: Instances



Flyweight: Applicabilité

- Etat intrinsèque/extrinsèque…
- Les états extrinsèques peuvent être calculés…

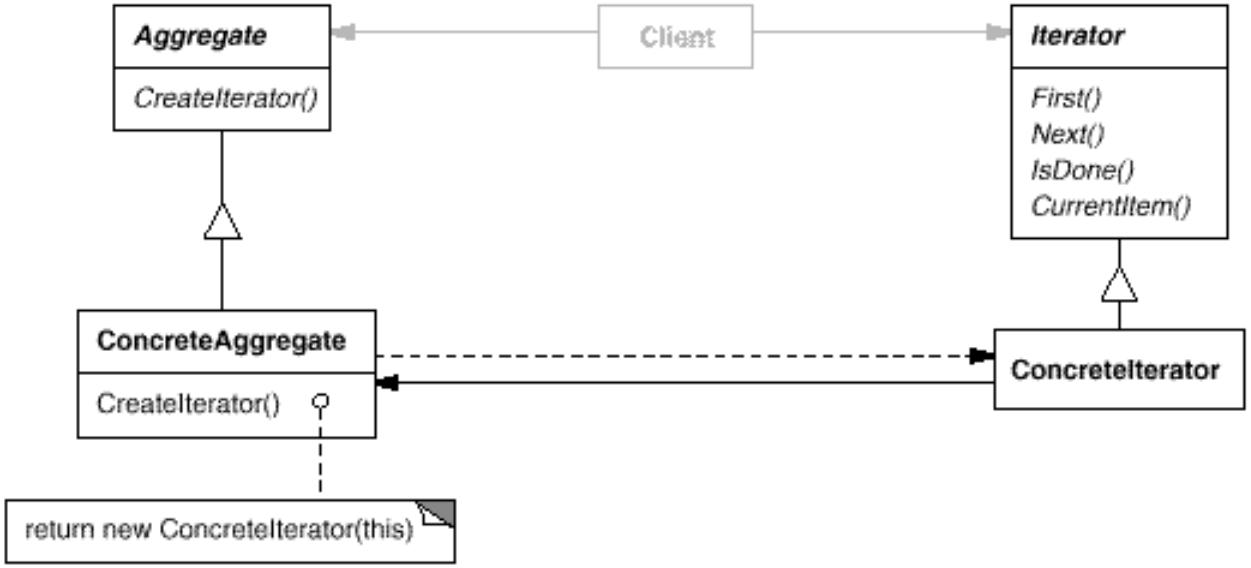
Flyweight


DocumentElement
DocCharFactory
myChar: MutableDocChar
docCharPool: Hashtable
getDocChar(in c: char): DocChar
0..1
~ parent
DocumentContainer
children: Vector
font: Font
parent: DocumentContainer
getChild(in index: int): DocumentElement
addChild(in child: DocumentElement)
removeChild(in child: DocumentElement)
getParent(): DocumentContainer
getFont(): Font
setFont(in font: Font)
DocChar
character: char
DocChar(in c: char)
getChar(): char
hashCode(): int
equals(in o: Object): boolean
Document
CharacterContext
Paragraph
LineOfText
Page

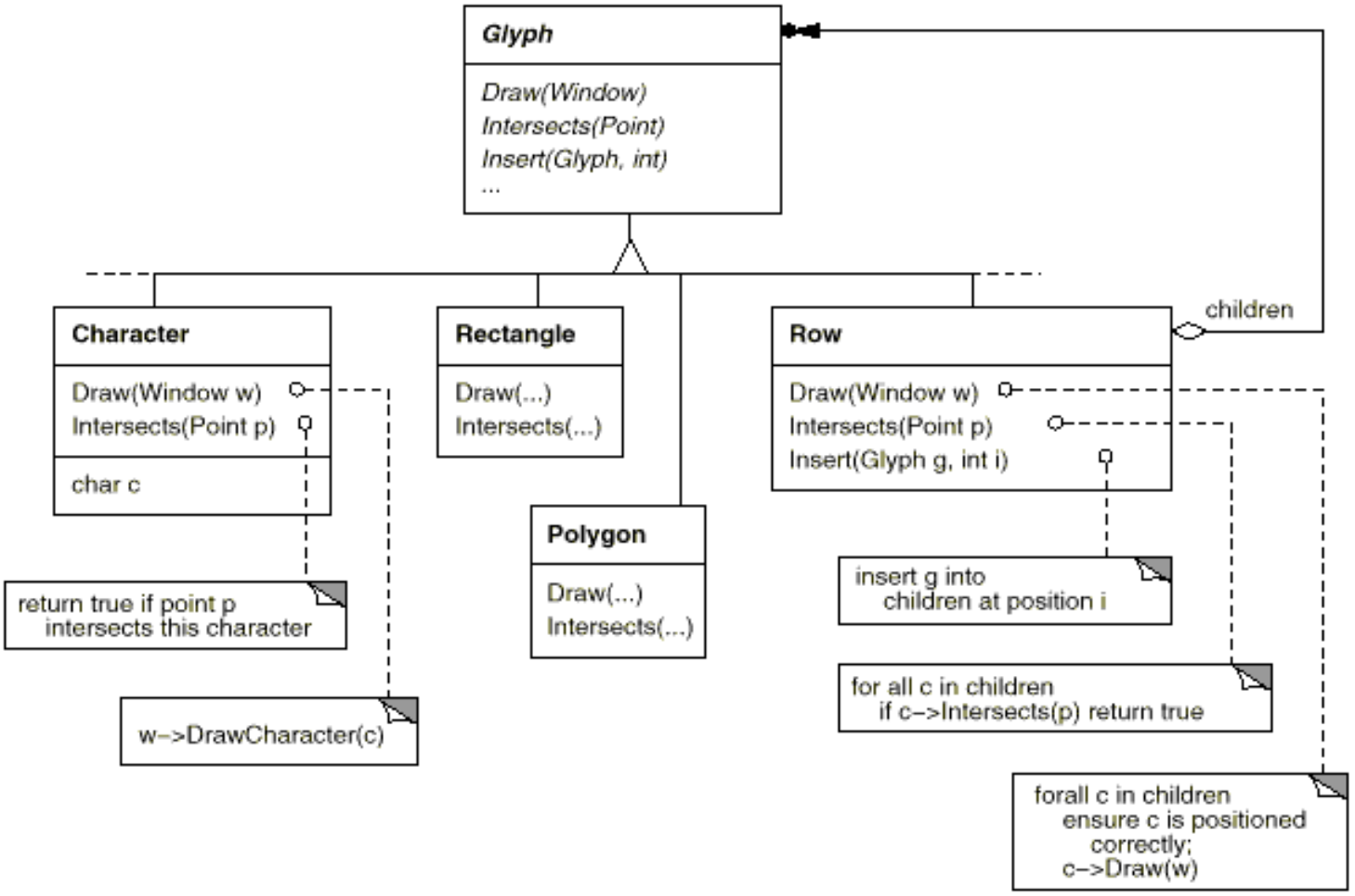
Iterator

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

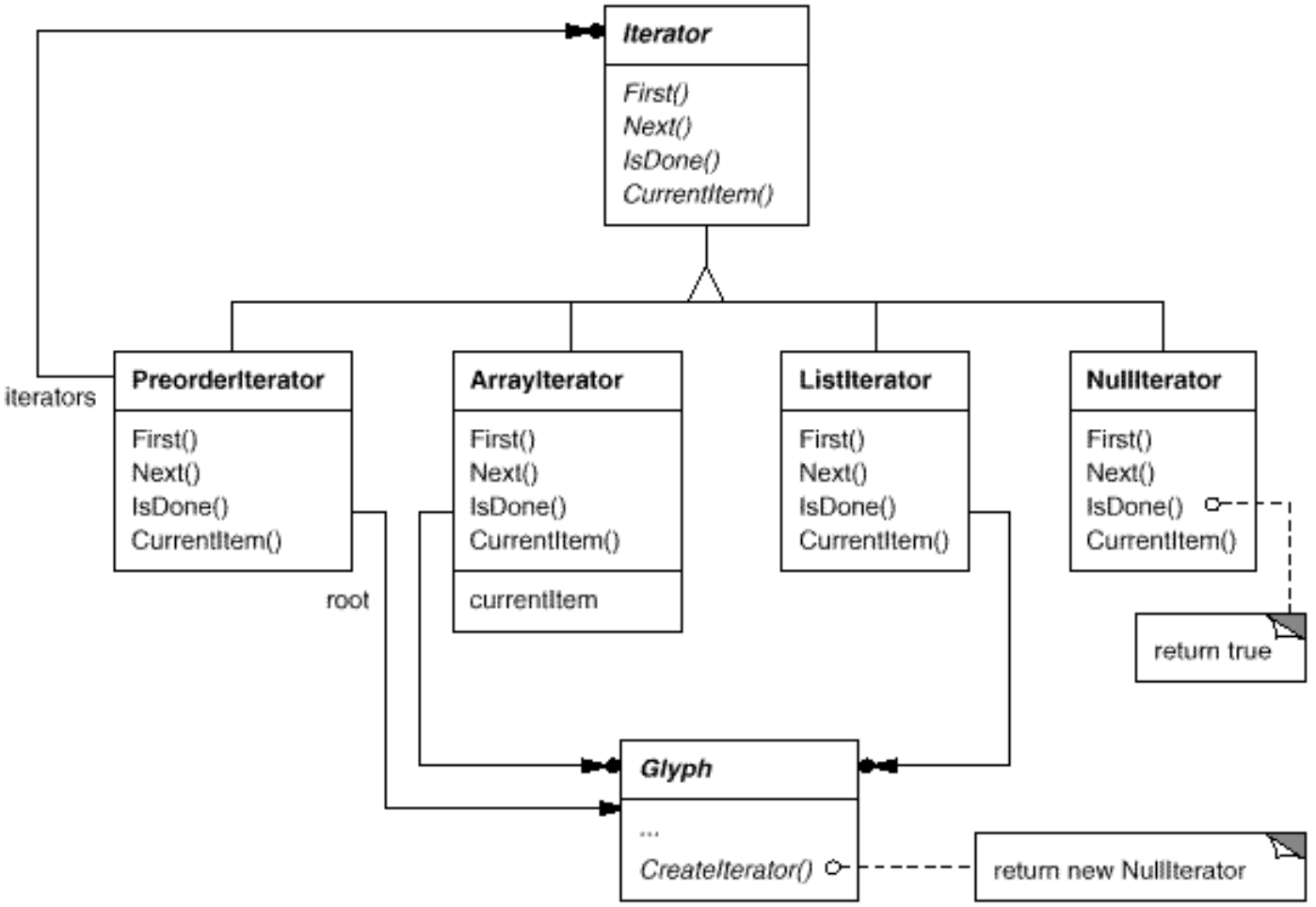
Iterator


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Iterator example:



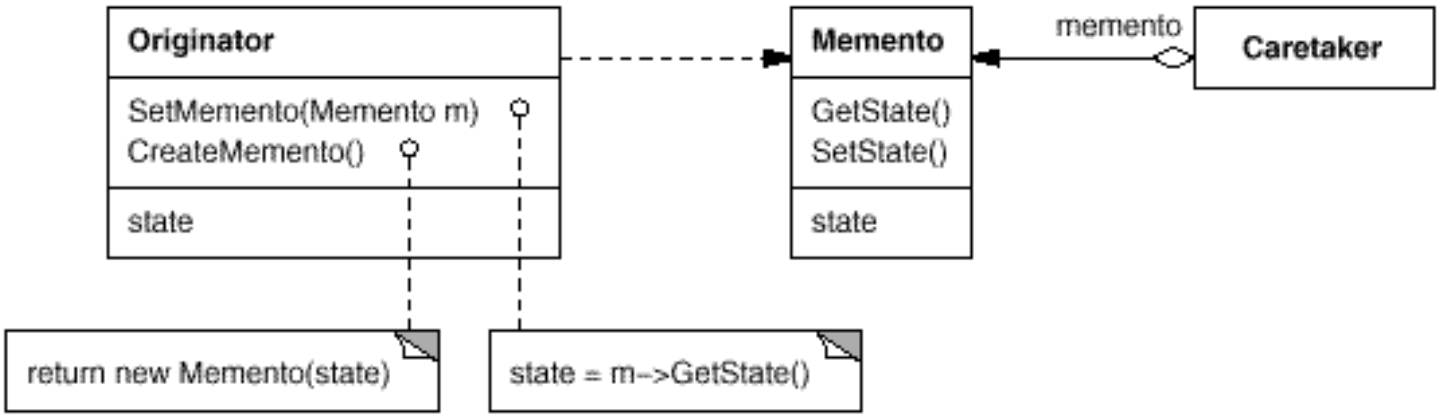
Exemple

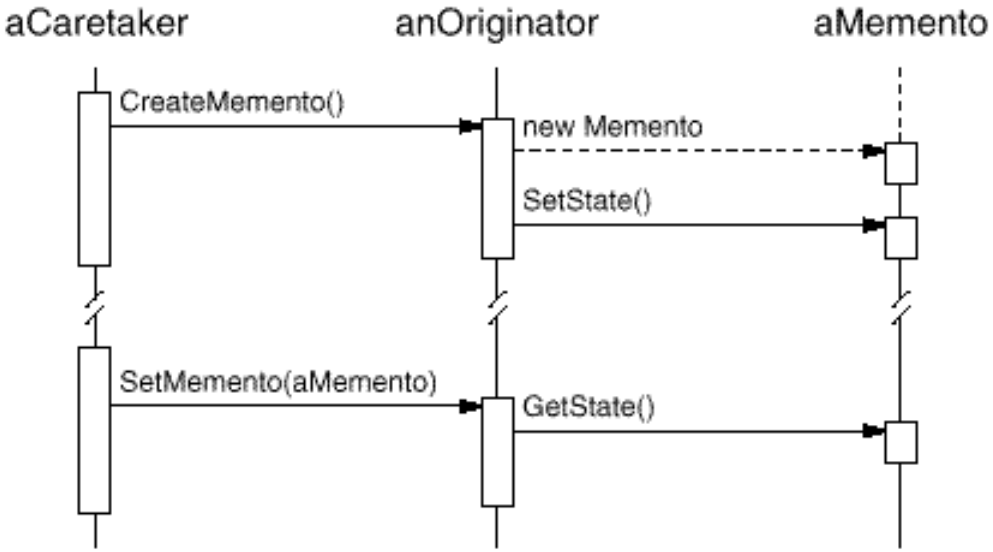

Iterator
First()
Next()
IsDone()
CurrentItem()
iterators
PreorderIterator
First()
Next()
IsDone()
CurrentItem()
root
ArrayIterator
First()
Next()
IsDone()
CurrentItem()
currentItem
ListIterator
First()
Next()
IsDone()
CurrentItem()
NullIterator
First()
Next()
IsDone()
CurrentItem()
return true
Glyph
...
CreateIterator()
return new NullIterator

Memento

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Memento Structure…

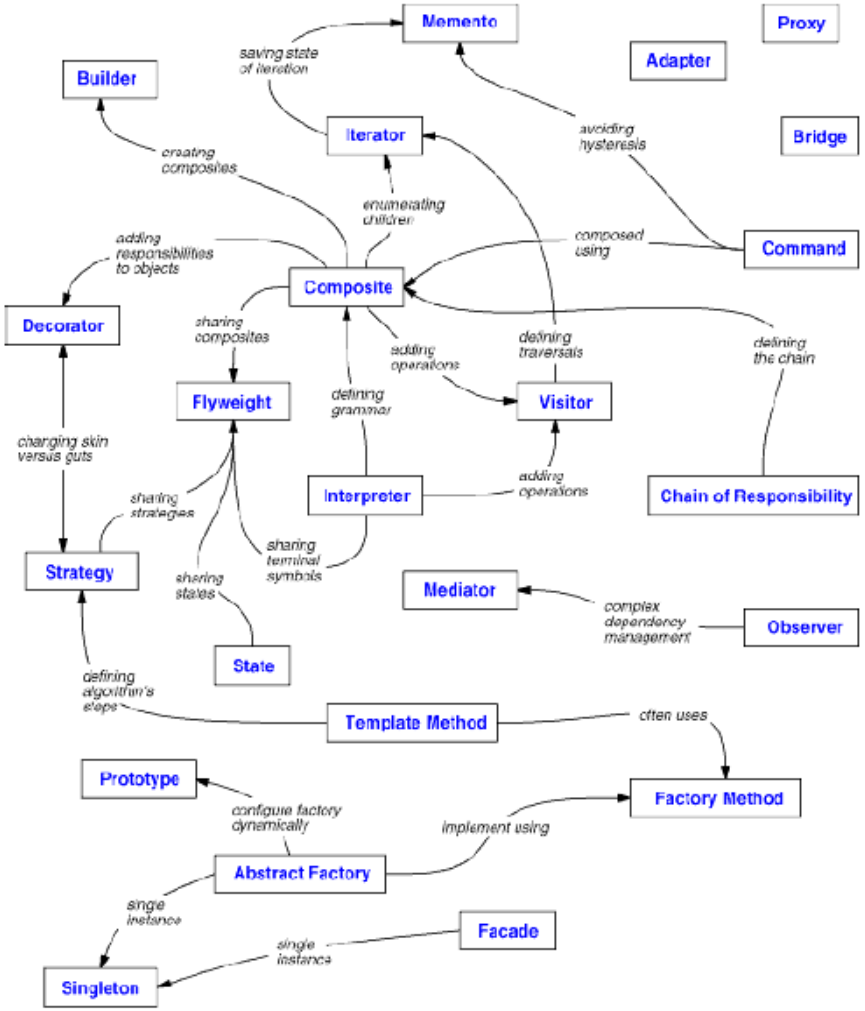




Memento…

- Preserving encapsulation boundaries
- It simplifies Originator
- Using mementos might be expensive.
- Defining narrow and wide interfaces
- Hidden costs in caring for mementos

Patterns relationships



What to Expect from Design Patterns

- A Common Design Vocabulary
- A Documentation and Learning Aid
- An Adjunct to Existing Methods
- A Target for Refactoring