Design of Software Systems (Ontwerp van SoftwareSystemen)

Design Patterns Reference

Roel Wuyts
2015-2016

Visitor

See lecture on design patterns

Composite

See lecture on design patterns

Patterns Catalogue Reference

- – Strategy
- – Decorator
- – Command
- – Factory Method
- – Singleton
- – Abstract Factory
- – Proxy
- – Adapter
- – Observer
- – Chain of Responsibility
- – FlyWeight
- – Facade

Strategy

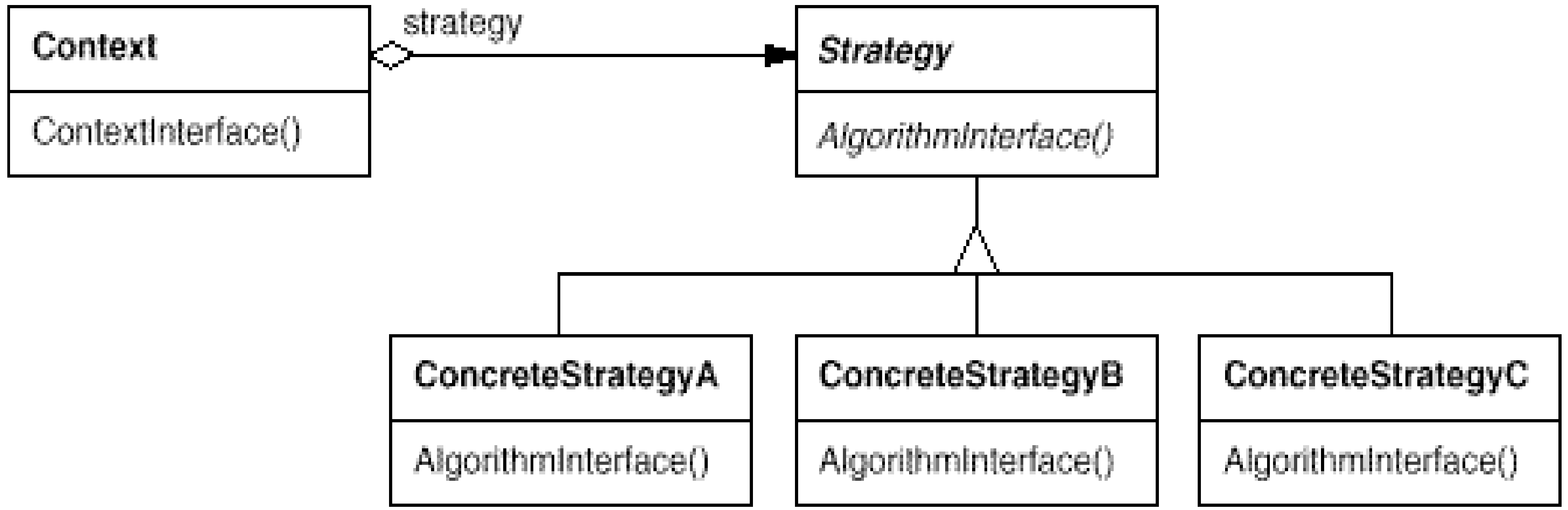
Strategy

Goal: encapsulate algorythms to make them interchangeable

variation of algorithms becomes possible

Considerations for implementation:

- Coupling between Context – Strategy:
  - Pass data as parameters to Strategy
  - Pass Context as parameter or as alias from Strategy



Consequences

Enables family of related algorithms. However common functionality needs to be possible in the abstract root class of the hierarchy.

Alternative for subclassing. Use subclassing at algorithmic level instead of context level.

No conditional statements needed – switch is implicit in used strategy object

Can make different implementations of the same behavior (for example with different execution speed/memory trade-off)

Caller needs to be aware of different possible strategies, and potentially select one

…

Consequences (ctd)

…

Communication-overhead between Context and Strategy
=> all potentially useful info needs to be passed but may be unused

Increasing number of objects in the system

=> Consider making Strategy a stateless object that can be shared between context objects (see Flyweight)

Decorator

Problem Illustration: UI for text editor

How to add borders, scroll bars, … to a text widget ?

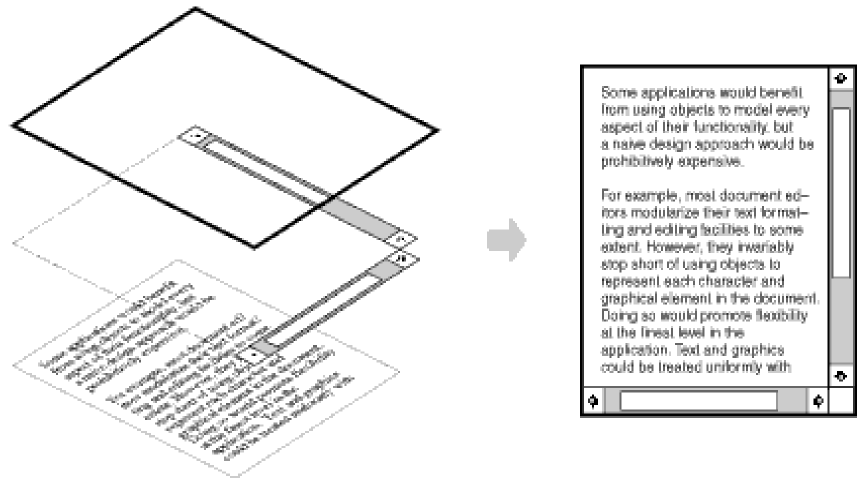
- Want to dynamically (at runtime) remove and combine them
  - (e.g. only add scrollbar when text flows over text box, and remove if unneeded)
- Transparant usage of UI objects

Inheritance solution:

- Combinatorial explosion (1 class/combination)
- Static solution

Object composition solution:

- Dynamic solution
- Does Glyph own Border, or does Border owns Glyph ???



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

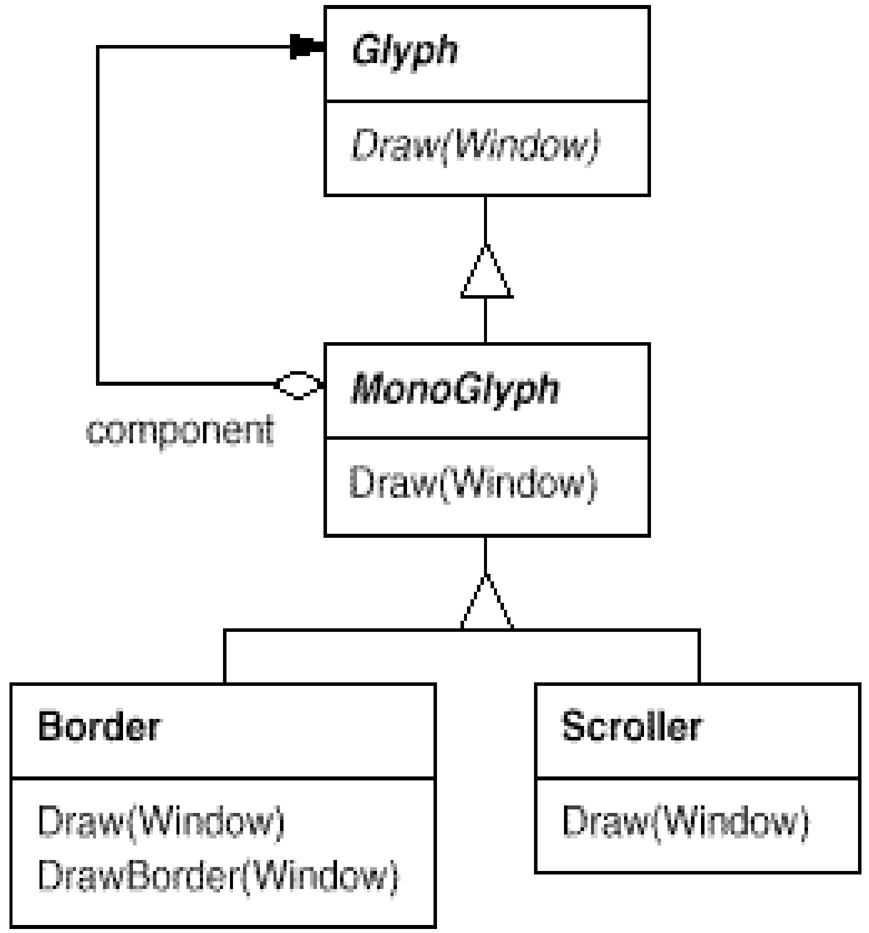
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application.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

Problem Illustration: decorator approach

Design of class Border:

- – Is visible element, so has to be subclass of Glyph
- – Border can be treated as any other Glyph

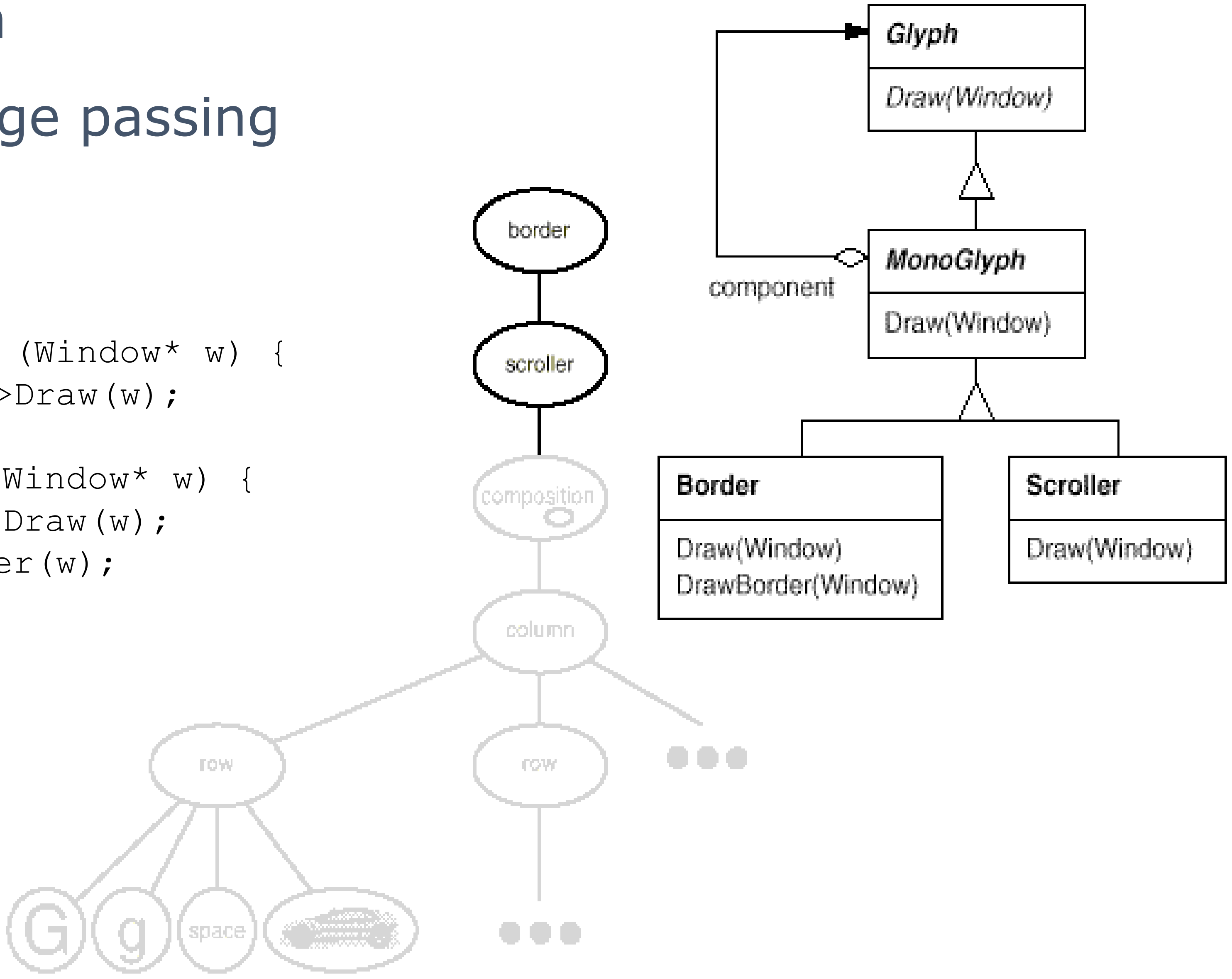
- Conceptual solution:
  - Singular component
  - Compatible interfaces



Problem illustration: decorator approach

- Dynamic configuration
- Execution uses message passing

```
void MonoGlyph::Draw (Window* w) {
          _component->Draw(w);
                   }
 void Border::Draw (Window* w) {
          MonoGlyph::Draw(w);
             DrawBorder(w);
                   }
```

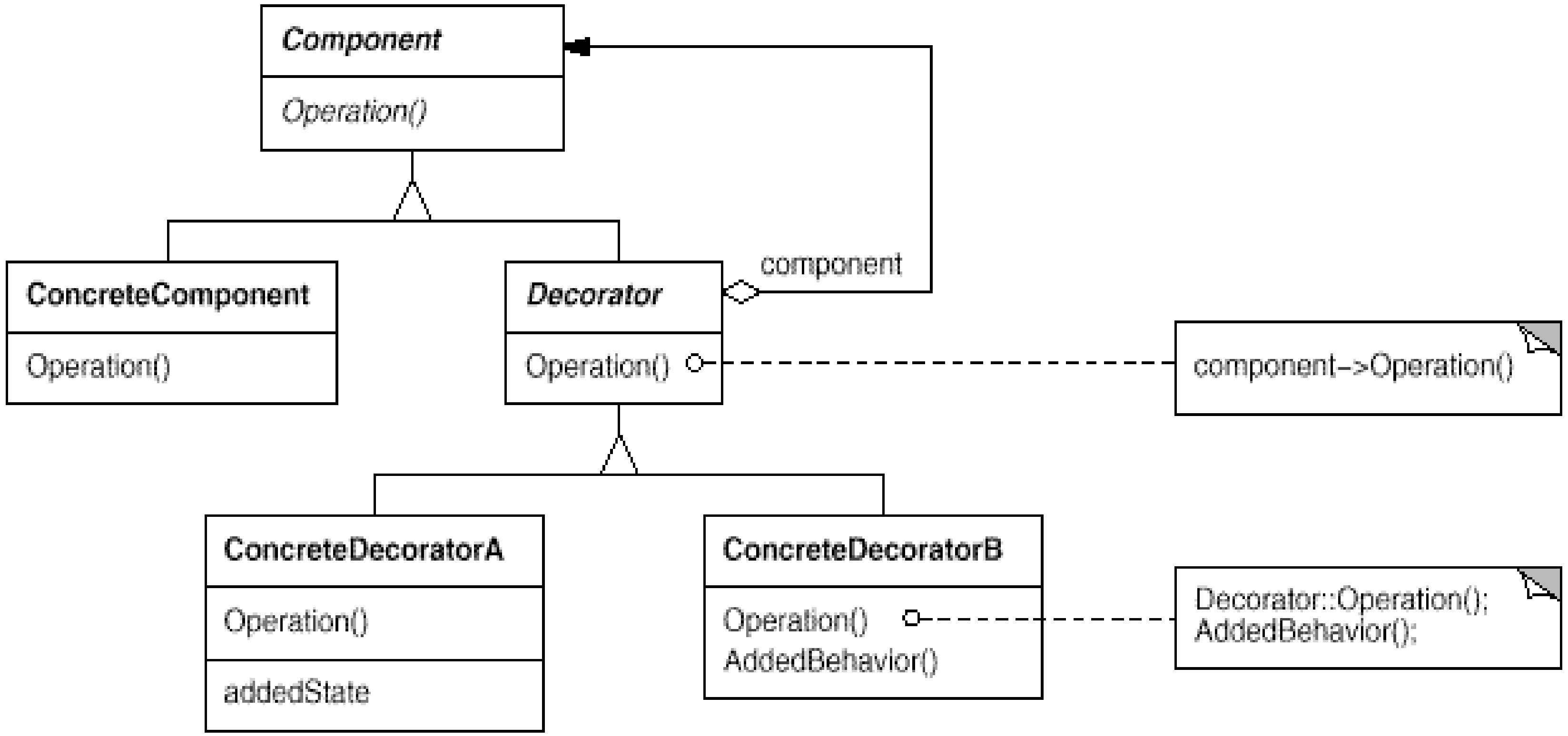


Decorator Pattern

Goal:

– Add behavior to individual objects dynamically and transparently

– Alternative for extension by subclassing



Consequences

Offers more flexibility than (static) subclassing

“pay as you go”: add small functional pieces instead of all-in-one

Decorated object has a different identity

Can result in many small objects

Implementation consideratons:

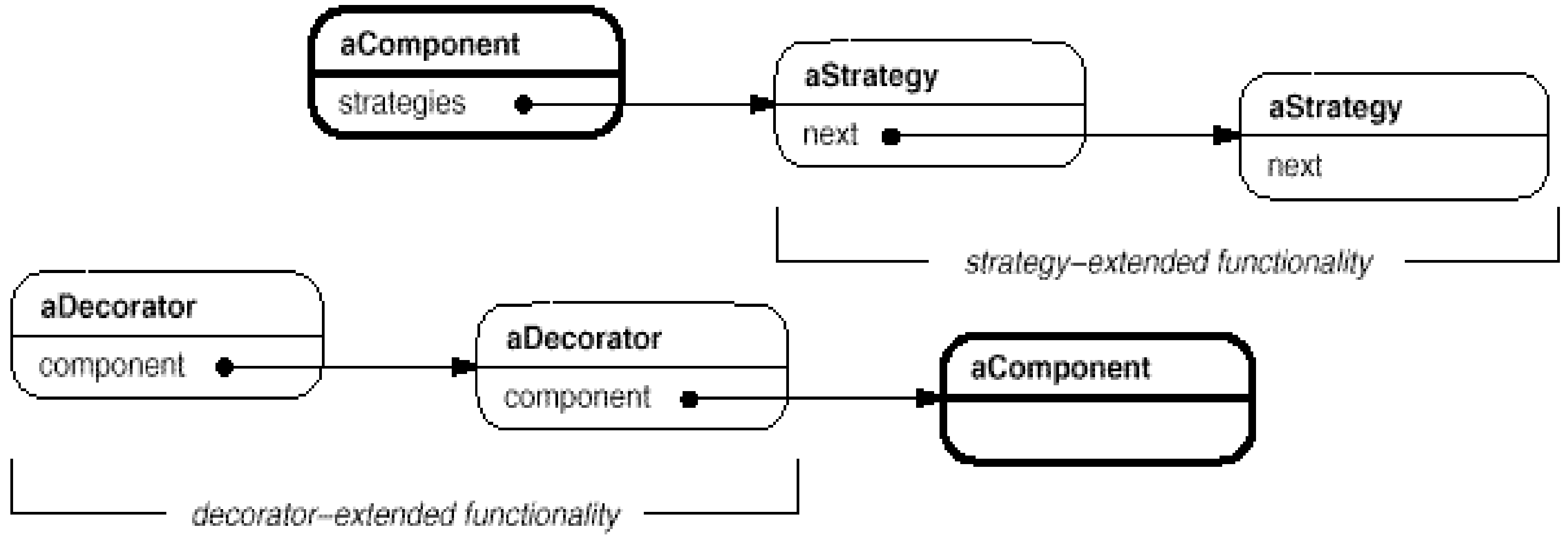
- keep the component class lightweight
- Only use an abstract decorator when there are multiple responsibilities

Decorator or Strategy ?

They both adapt object behaviour

Strategy has to be known by component, but can have different interface

Strategy preferable when the Component class is heavyweight



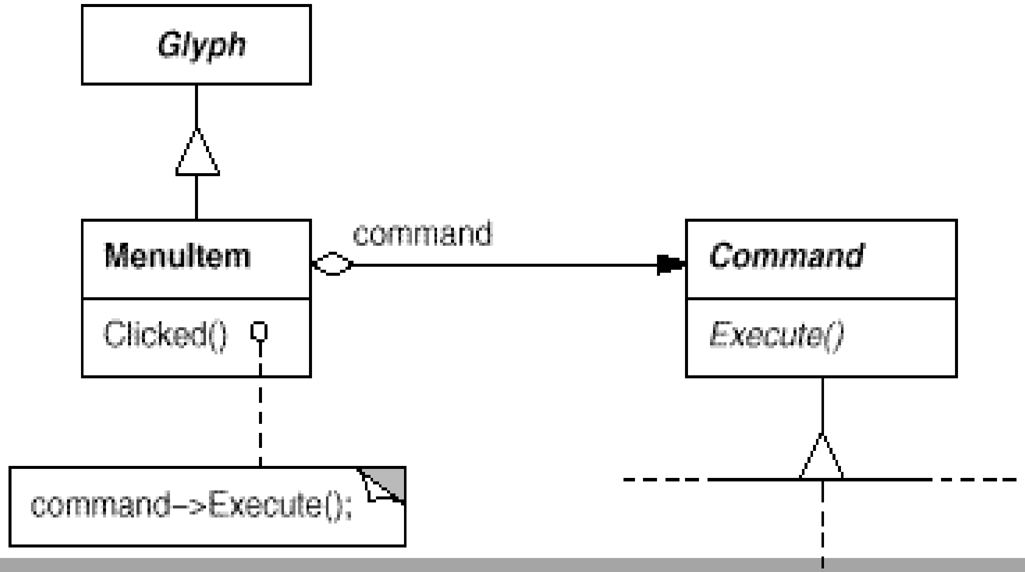
Command

Example: separating UI from domain-level functionality in text editor

Gols:

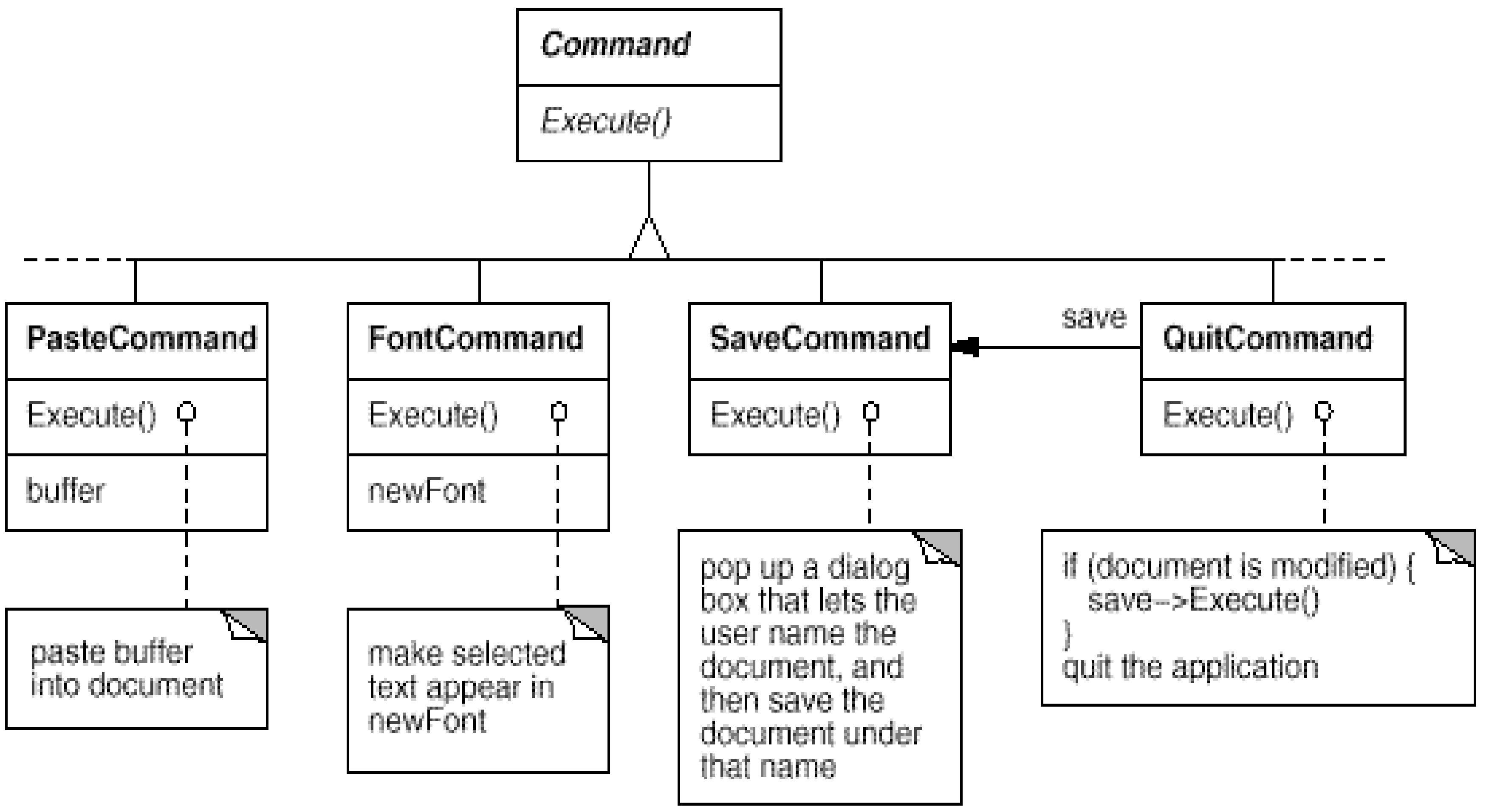
- Separate domain operations from UI
  - Show a single operation in different ways in the GUI
  - Otherwise results in high coupling between UI classes and application
- Support undo and redo of operations

Solution: introduce Command class



Command Pattern: example

Every concrete Command class has information about and implements a single operation

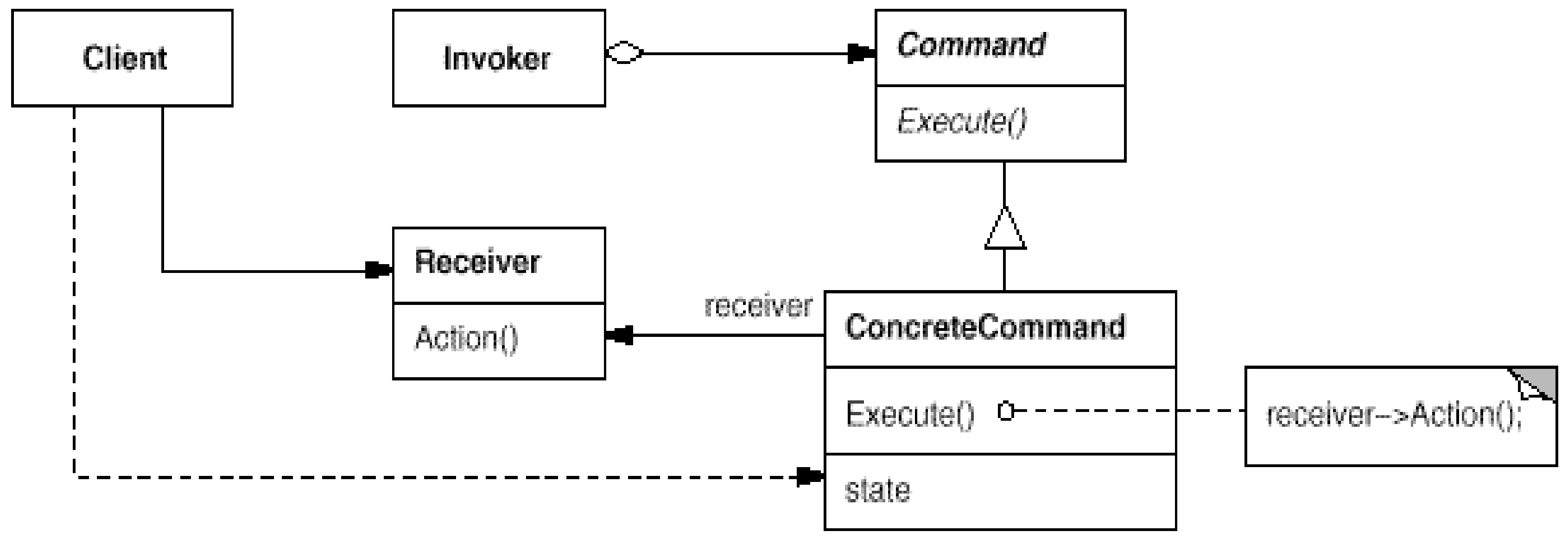

Command
Execute()
PasteCommand
Execute()
buffer
paste buffer into document
FontCommand
Execute()
newFont
make selected text appear in newFont
SaveCommand
Execute()
pop up a dialog box that lets the user name the document, and then save the document under that name
save
QuitCommand
Execute()
if (document is modified) { save->Execute() } quit the application

Command

Goal:

- Turn operations into first-order objects in order to manipulate them (parametrization, queueing, logging, undoing, …)

Structure:



Consequences

Decouples caller and executor

Commands become first-class entities that can be manipulated and extended

Commands can be grouped in composite commands

Makes it easy to add new commands

- – no need to change a given base class

Factory Method

Factory Method

Category

- Creational

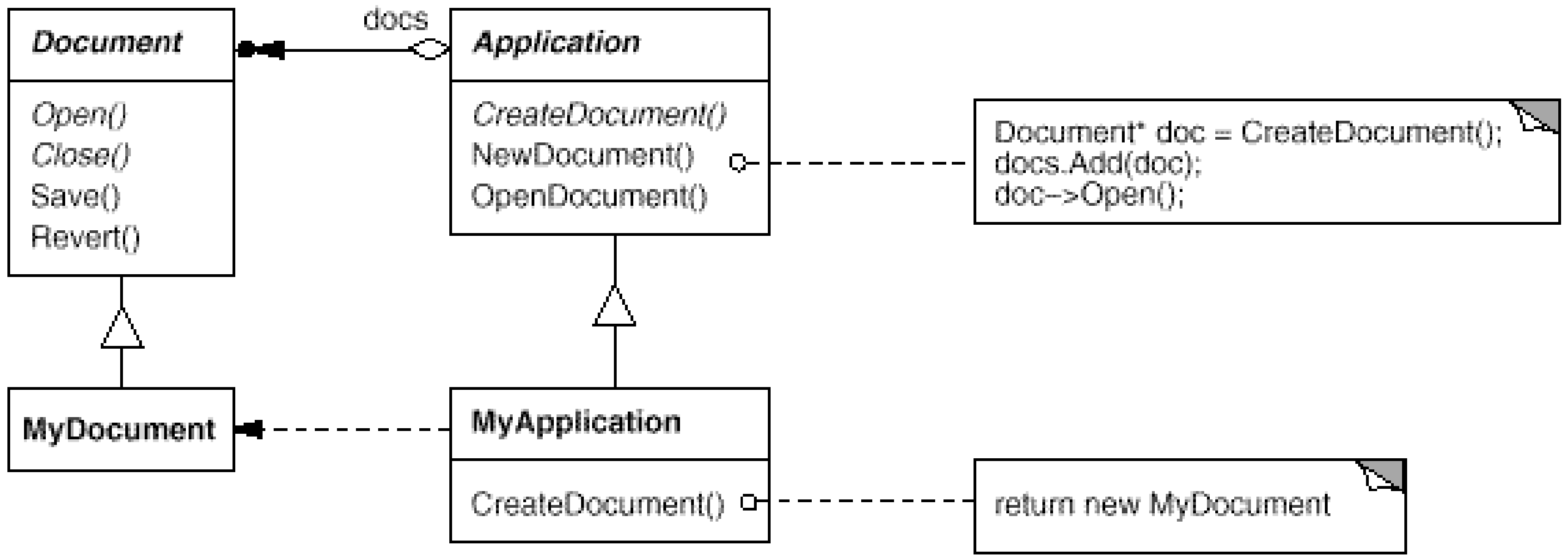
Intent

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

Motivation

- When frameworks or toolkits use abstract classes to define and maintain relationships between objects and are responsible for creating the objects as well.

Motivation (cont)



```
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
```

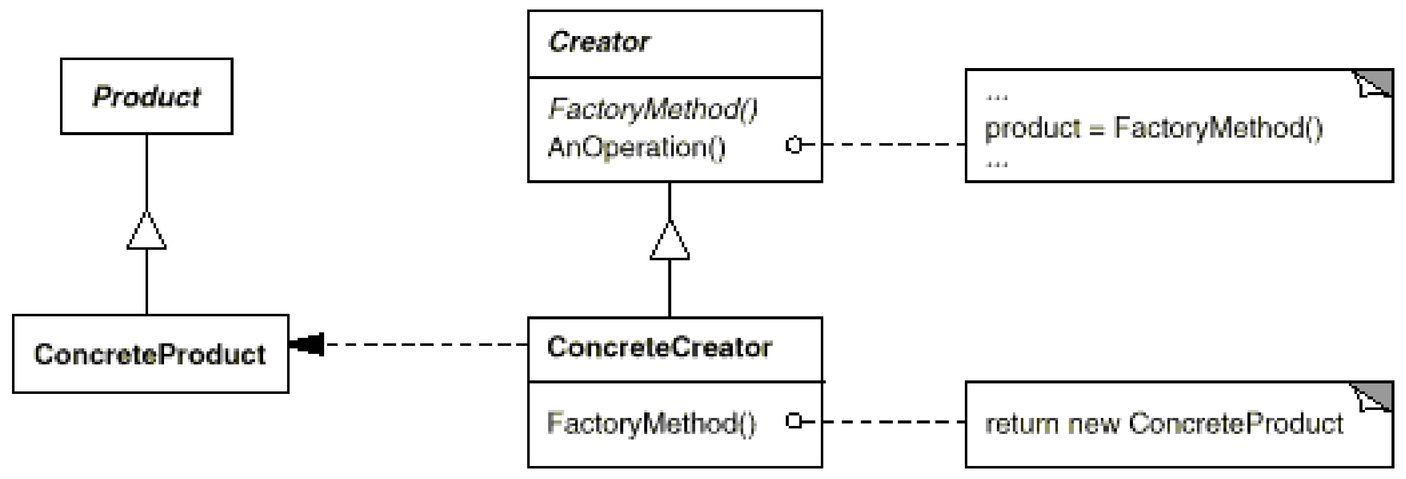
```
return new MyDocument
```

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

Participants

Product

- Defines the interface of objects the factory method creates.

ConcreteProduct

- Implements the Product interface.

Creator

- Declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- They call the factory method to create a Product object.

ConcreteCreator

- Overrides the factory method to return an instance of a ConcreteProduct.

Collaboration

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

Consequences

Eliminates the need to bind application specific classes into your code.

Clients might have to subclass the Creator class just to create a particular ConcreteProduct object.

Provides hooks for subclasses

- the factory method gives subclasses a hook for providing an extended version of an object.

Connects parallel class hierarchies

- a client can use factory methods to create a parallel class hierarchy (parallel class hierarchies appear when objects delegate part of their responsibilities to another class).

Known Uses

Toolkits and frameworks

Class View in Smalltalk-80

- contains a defaultController method which is a Factory Method.

Class Behavior in Smalltalk-80

- contains a parserClass method which also is a factory method.

Could also be used to generated an appropriate type of proxy when an object requests a reference to an object. Factory Method makes it easy to replace the default proxy with another one.

Singleton

Singleton

Category

- Creational

Intent

- Ensure a class only has one instance, and provide a global point of access to it.

Motivation

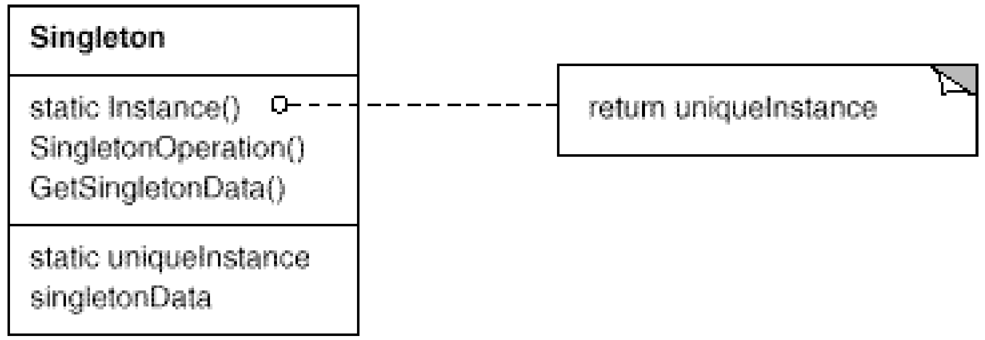
- There should be only one instance.
- For example, many printers, but only one printspooler.
- Using a global variable containing the single instance?
  - Cannot ensure no other instances are created.
- Let the class control single instance.

Applicability and Structure

Use Singleton pattern when

- There must be exactly one instance of al class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants and Collaborations

Participants

- Singleton
  - Defines an instance operation that lets clients access its unique instance. Instance is a class operation that will either return or create and return the sole instance.
  - May be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton solely through Singleton's instance operation.

Consequences

Controlled access to sole instance.

- Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

Reduced name space.

- The Singleton pattern is an improvement over global variables that store sole instances.

Consequences (cont)

Permits refinement of operations and representation.

- The Singleton class may be subclassed, an application can be configured with an instance of the class you need at runtime.

Permits a variable number of instances.

- The same approach can be used to control the number of instances that can exist in an application, only the operation that grants access to the instance(s) must be provided.

More flexible than class operations.

Known Uses

Every time you want to limit the creation of additional object after the instantiation of the first one. This is usefull to limit memory usage when multiple objects are not necessary.

Questions

What is the difference with a global variable?

Gamma (one of the authors of the book on Design Patterns) recently pointed out that he was very unhapy with this pattern. More specifically he claims that it usually indicates bad design. Can you imagine why he thinks this ?

Abstract Factory

Abstract Factory

Category

- Creational

Intent

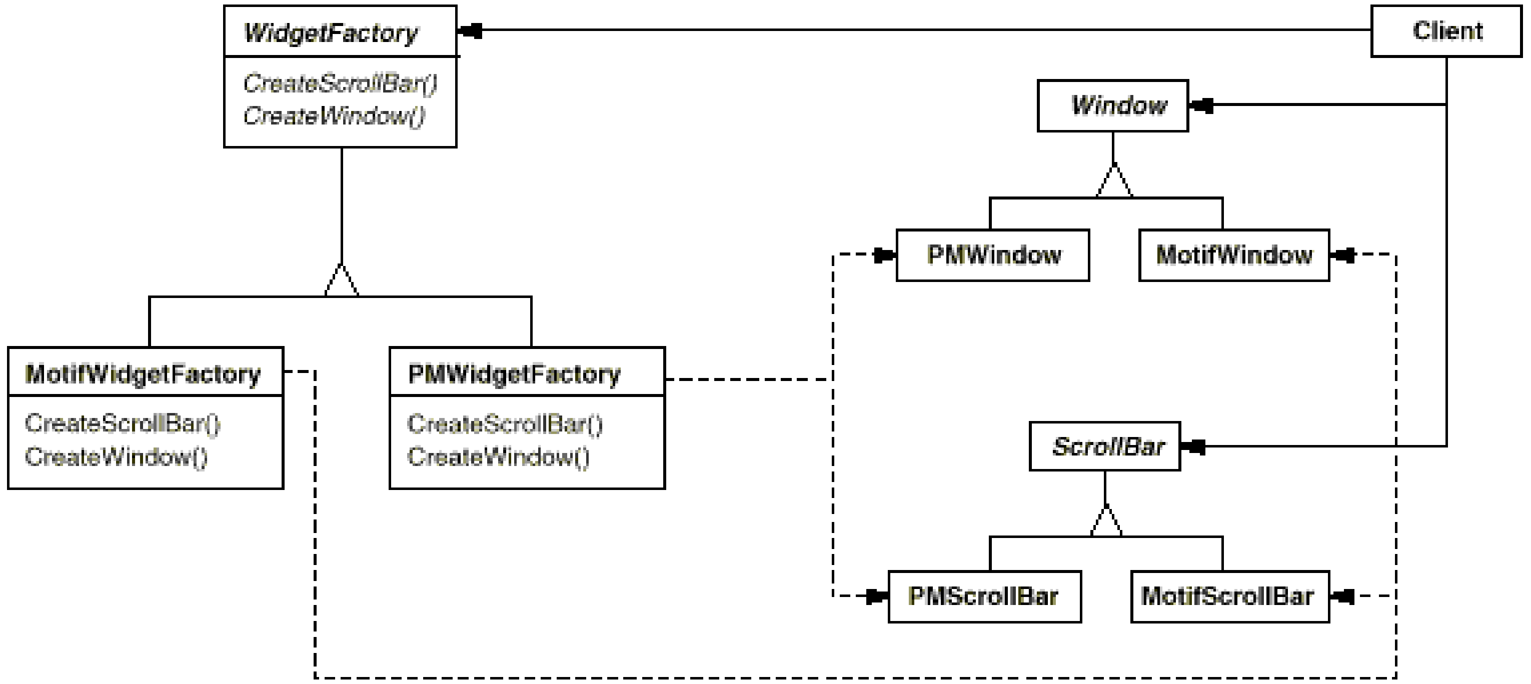
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Motivation

- User interface toolkit for multiple look-and-feel standards.
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Motivation (cont)

Motivation (cont)

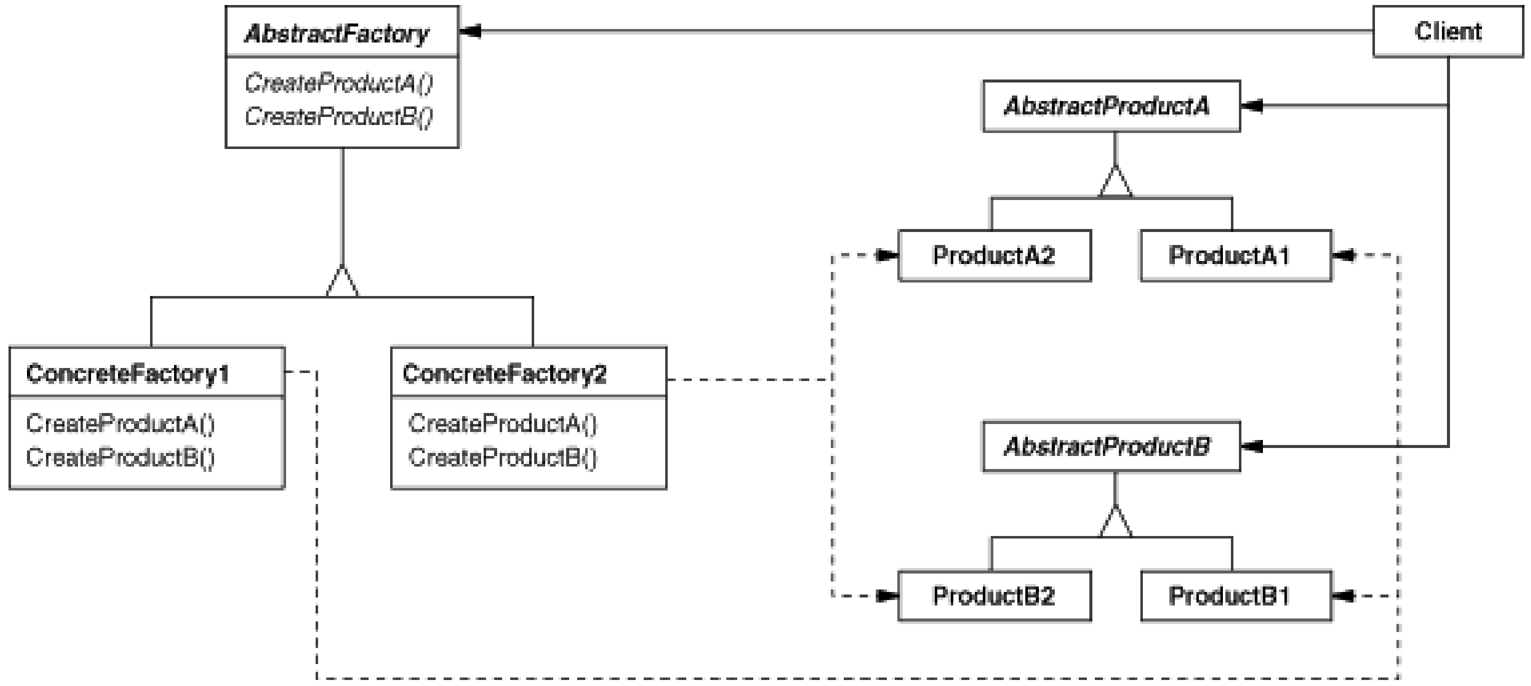


Applicability

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

- Declares an interface for operations that create abstract product objects.

ConcreteFactory

- Implements the operations to create concrete product objects.

AbstractProduct

- Declares an interface for a type of product object.

Participants (cont)

ConcreteProduct

- 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 ConcreteFactory is created at run-time. This concrete factory creates products having a particular implementation.

AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

It isolates concrete classes.

- The abstract factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes.
- Product class names are isolated in the implementation of the concrete factory and do not appear in the client code.

It makes exchanging product families easy.

- The concrete factory appears only one in the application – that is, where it is instantiate – to it is easy to replace.

Consequences (cont)

It promotes consistency among products.

- When products of one family are designed to work together, it is important for an application to use objects from one family only.
- The abstract factory makes this easy to enforce.

Supporting new kinds of products is difficult.

- Because the abstract factory interface fixes the set of products that can be created, it is not easy to add new products.
- This would require extending the factory interface which involves extending changing the abstract factory and all its subclasses.

Known Uses

Usually used in toolkits for generating look-and-feel specific user interface objects.

Also used to achieve portability across different window systems.

Questions

Describe the working of the abstract factory pattern with your own words.

What pattern(s) is (are) often used together with the abstract factory pattern?

Proxy

Proxy

Category

– Structural

Intent

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

Motivation

– Defer the full cost of the creation and initialisation of an object until we actually need it.

– For example: a document with lots of graphical objects can be expensive to create, but opening it should be fast.

– A proxy could act as a stand-in for the real objects.

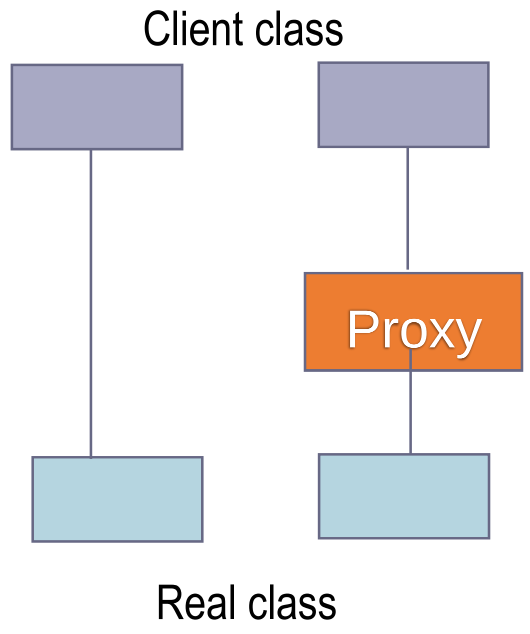
Kinds of Proxies

a remote proxy provides a local representative for an object in a different address space.

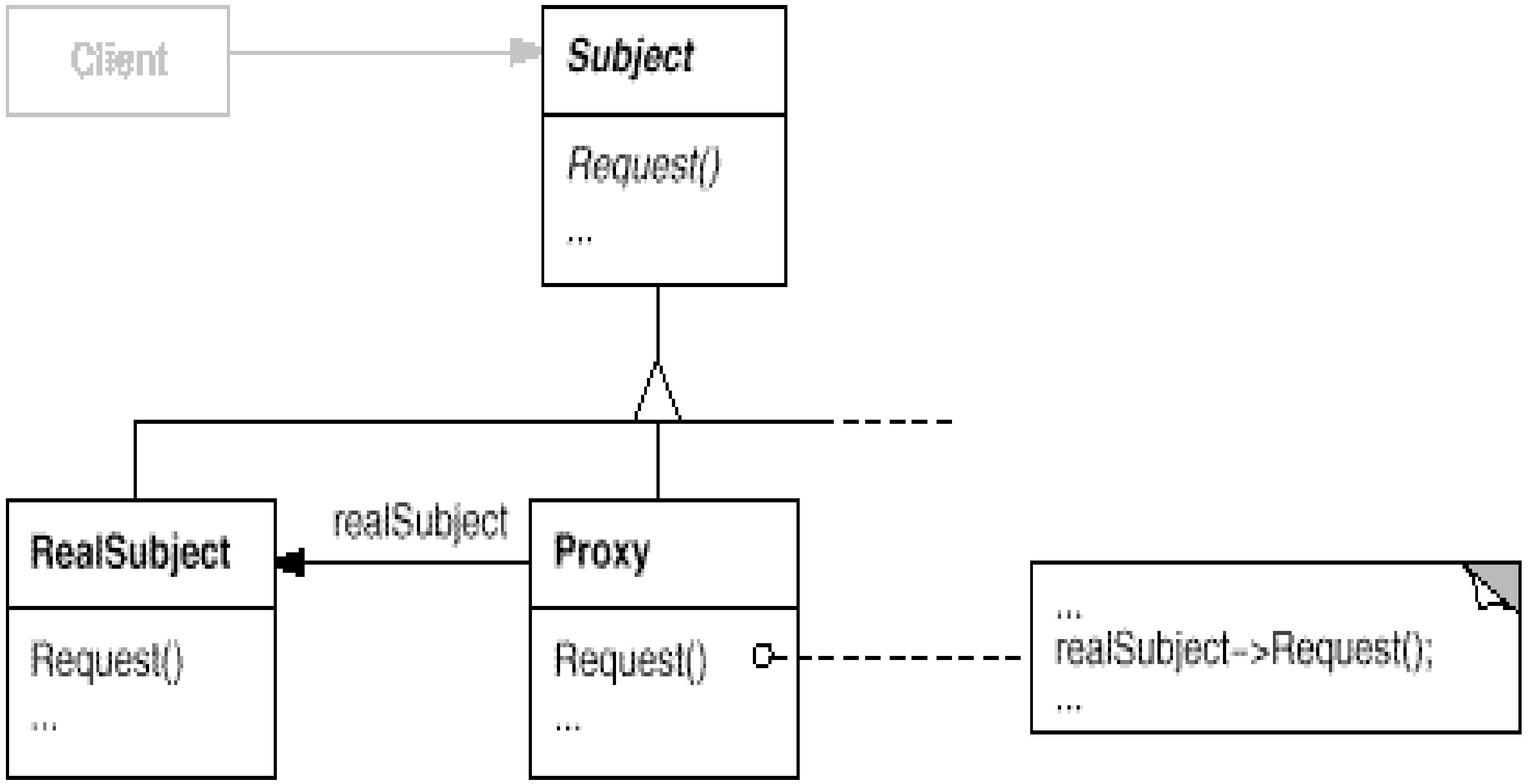
a virtual proxy creates expensive objects on demand.

a protection proxy controls access to the original object and are useful when objects have different access rights.

a smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed: e.g. counting references, loading a persistent object when it is first referenced, locking the real object, ...



Structure


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

Participants

Proxy

- Maintains a reference that lets the proxy access the real subject.
- Provides an interface identical to the 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.
- Remote proxies are responsible for encoding a request and its arguments and for sending the request to the real subject in the other address space.
- Virtual proxies may cache information about the real subject so that they can postpone accessing it.
- Protection proxies check that the caller has the access permission to perform a request.

Participants (ctd) and Collaboration

Participants (cont)

- Subject
  - Defines a 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.

Collaboration

- 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. This indirection has many uses:

- A remote proxy can hide the fact that the object resides in a different address space.
- A virtual proxy can perform optimisations.
- Both protection proxies and smart pointers allow additional housekeeping.

Creative Commons License 4

Consequences (cont)

The proxy patterns can be used to implement “copy-on-write”.

- To avoid unnecessary copying of large objects the real subject is referenced counted.
- Each copy requests increments this counter but only when a clients requests an operation that modifies the subject the proxy actually copies it.

Known Uses

Encapsulators can be implemented as proxies.

They are often used to represent local representatives for distributed objects.

They have been used in textbuilding tools to enhance performance.

Questions

If a Proxy is used to instantiate an object only when it is absolutely needed, does the Proxy simplify code?

Adapter

Adapter

Category

- Structural

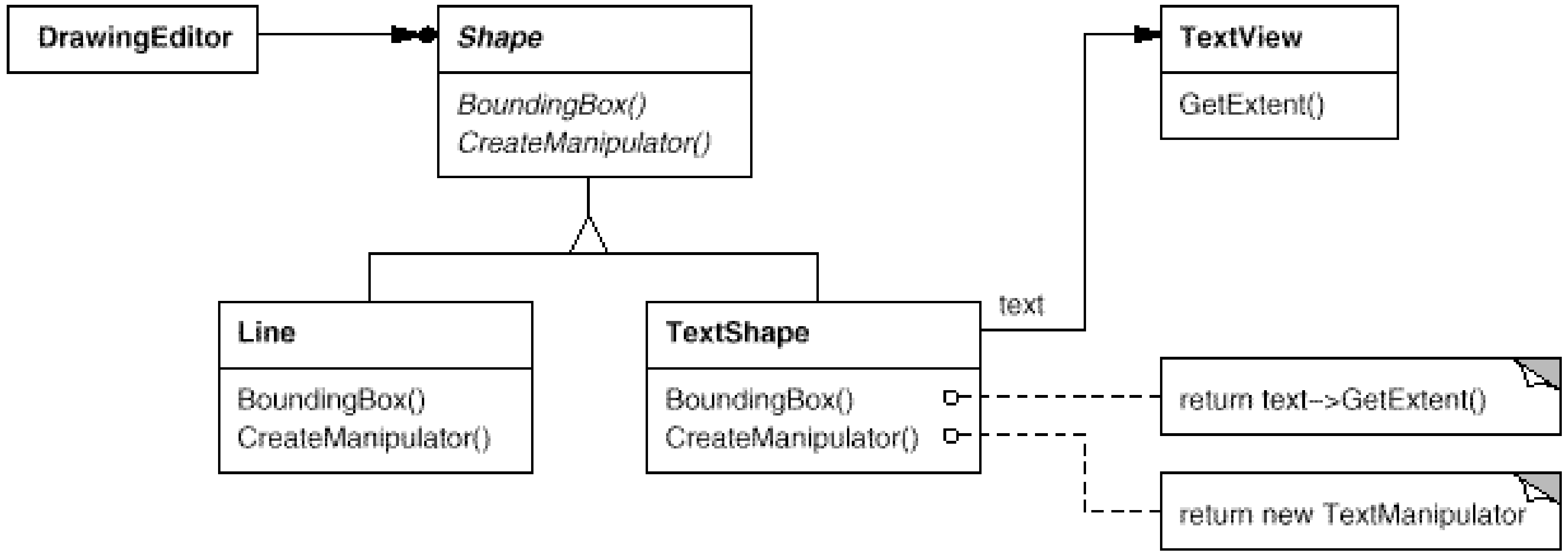
Intent

- Convert the interface of a class into another interface clients expect. Lets classes with incompatible interfaces work together.

Motivation

- Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires.
- A drawing editor has one abstraction for lines and textboxes, but textbox has a different interface and implementation.

Motivation (cont)



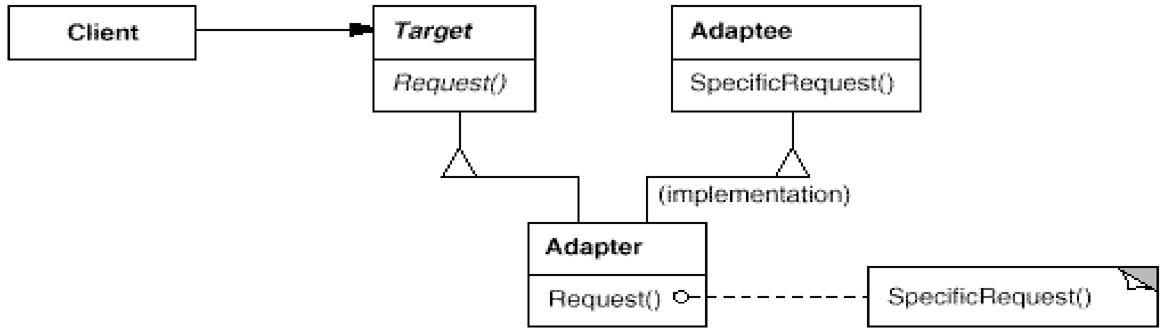
Applicability

Use Adapter when

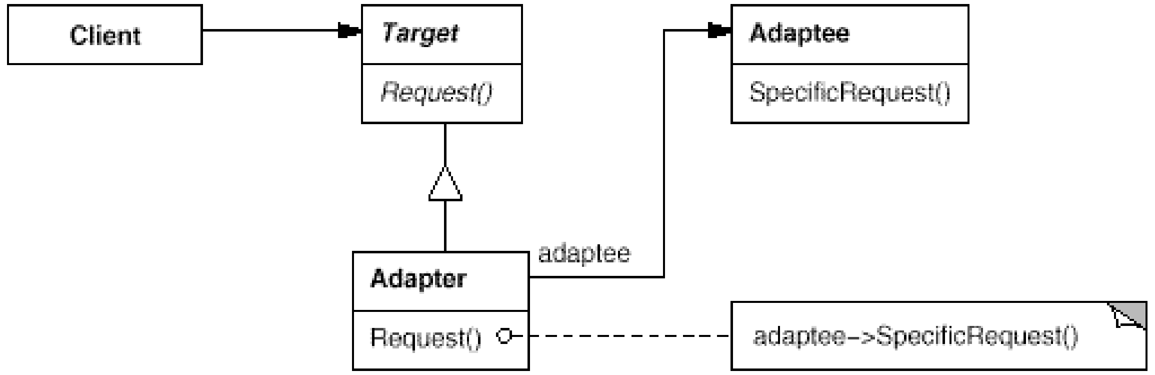
- You want to use an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, which do not necessarily have compatible interfaces.
- (object adapter only) You need to use several existing subclasses, but it’s impractival to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure

– Class adapter



– Object adapter



Participants and Collaborations

Participants

- Target
  - Defines the domain-specific interface that Client uses.
- Client
  - Collaborates with objects conforming to the Target interface.
- Adaptee
  - Defines an existing interface that needs adapting.
- Adapter
  - 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

How much adapting does Adapter do?

- Ranges from simple interface conversion to supporting an entirely different set of operations.

Pluggable adapters.

- By building interface adaption into a class, it becomes more reusable because it does not assume the same interface to be used by other classes.

Using two-way adapters to provide transparency.

- An adapted object no longer conforms to the Adaptee interface, so it can’t be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency.

Questions

Would you ever create an Adapter that has the same interface as the object which it adapts? Would your Adapter then be a Proxy?

Observer

Observer

Category

– Behavioral

Intent

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

Motivation

– different types of GUI elements depicting the same application data.

– different windows showing different views on the same application model.

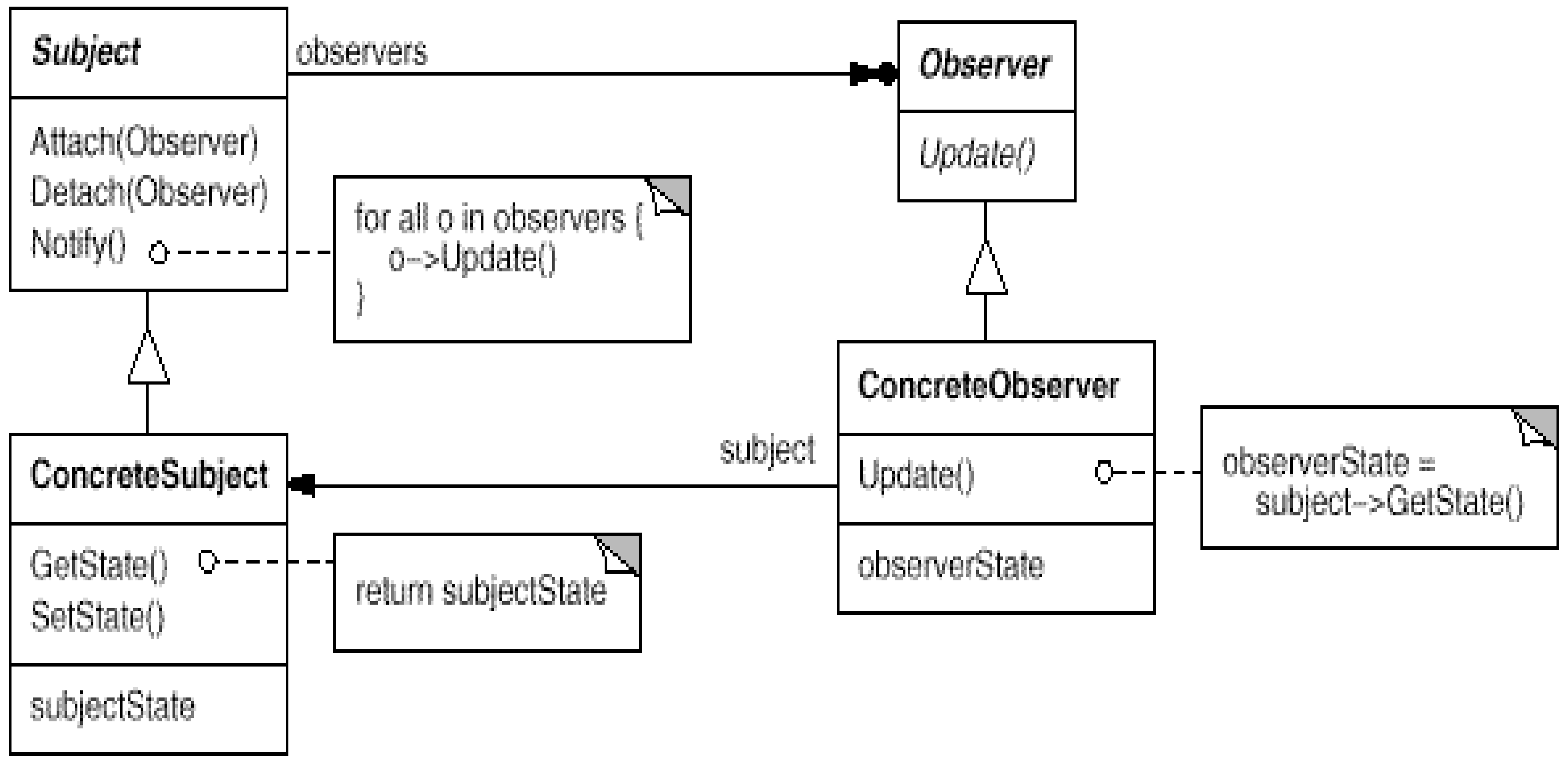
Applicability

When an abstraction has two aspects, one dependant on the other. Encapsulating these aspects in seperate 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 do not want these objects tightly coupled.

Structure


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

Participants

Subject

- knows its observers. Any number of Observer objects may observe an object.
- provides an interface for attaching and detaching Observers.

Observer

- defines an updating interface for objects that should be notified of changes in a subject.

Participants (cont)

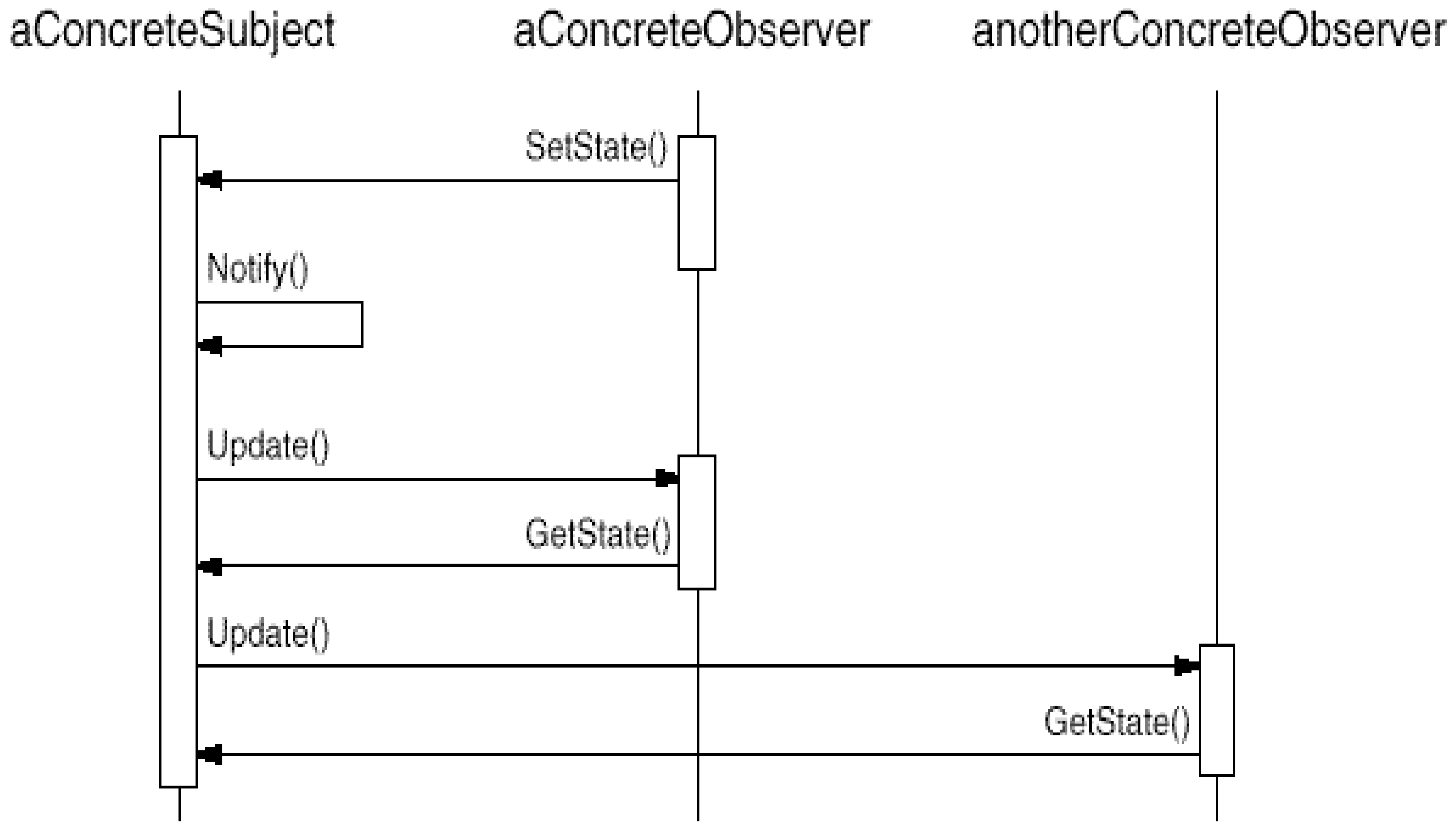
ConcreteSubject

- 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 consistant with the subject's.
- implements the Observer updating interface.

Collaborations



Consequences

Abstract and minimal coupling between Subject and Observer.

– The subject does not know the concrete class of any observer. Concrete subject and concrete observer classes can be reused independently.

Support for broadcast communication.

– The notification a subject sends does not need to specify a receiver, it will broadcast to all interested parties.

Consequences (cont)

Unexpected updates.

- Observers don’t have knowledge about each other’s presence, a small operation may cause a cascade of updates.

Known Uses

Best known use is Smalltalk Model/View/Controller.

Questions

There are two methods for propagating data to observers with the Observer design pattern: the *push* model and the *pull* model. Why would one model be preferable over the other? What are the trade-offs of each model?

In what real-world system can we expect encounter the Observer pattern quite often?

Chain of Responsibility

Chain of Responsibility

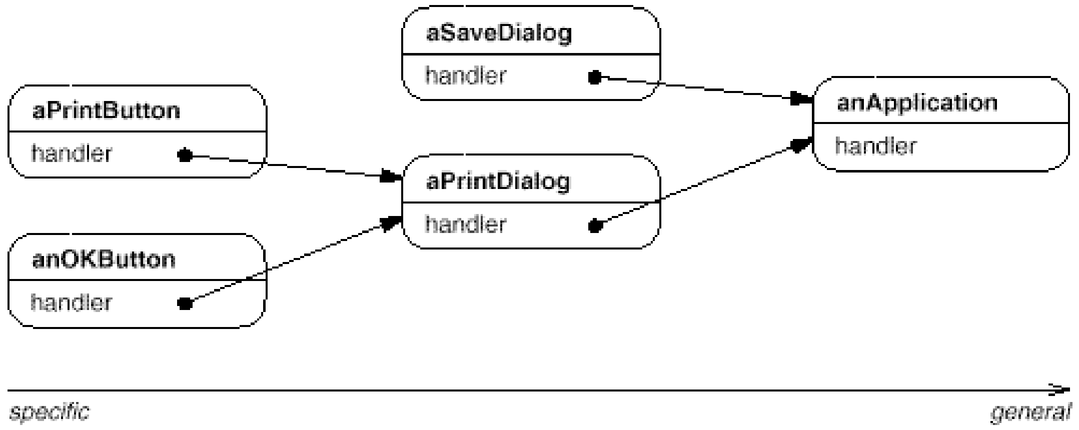
Category

– Behavioral

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

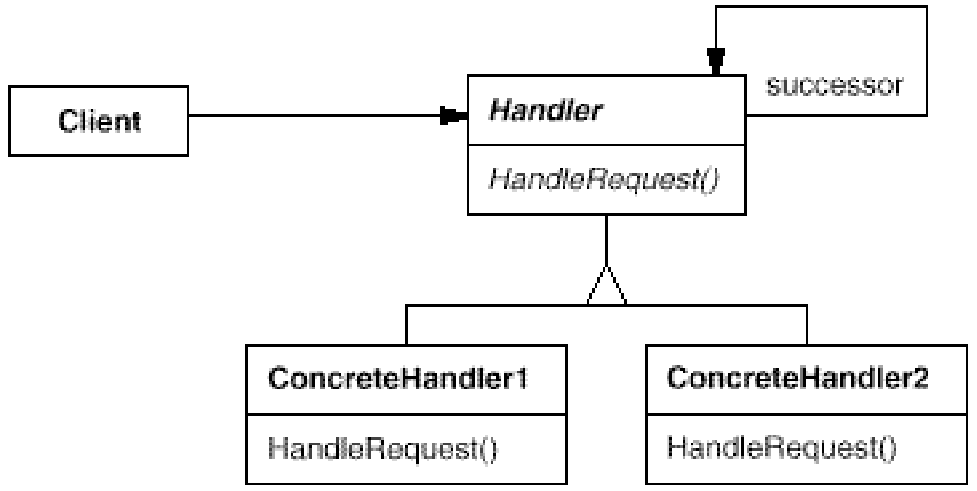


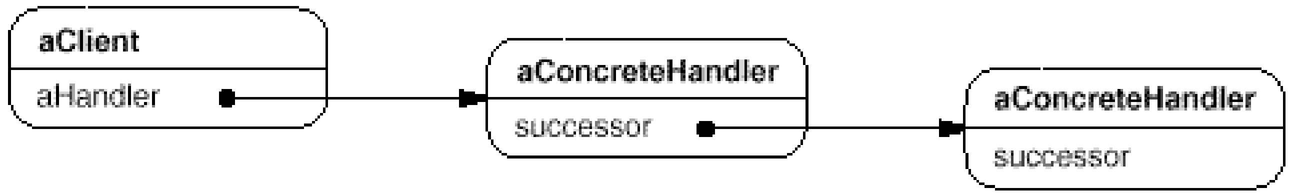
Applicability

Use Chain of Responsibility when

- more than one object may handle a request, and the handler is not known a priori.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Structure


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


aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

Participants

Handler

- defines an interface for handling objects.
- (optional) implements the successor link.

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.

Client

- initiates the request to a ConcreteHandler object on the chain.

Collaborations and Consequences

Collaborations

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility to handle it.

Consequences

- Reduced Coupling
  - The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled appropriately.

Consequences (cont)

Added flexibility in assigning responsibilities to objects.

- You can add or change responsibilities for handling a request by adding or changing the chain at runtime.
- Receipt is not guaranteed.
  - Since a request has no implicit receiver, there is no guarantee that it will be handled, it could fall of the end of the chain without being handled.

Known Uses

Different class libraries use this pattern, giving different names to handlers, e.g. when a user clicks on a mouse button, an event gets generated and passed along the chain.

Is also used in graphical systems, where a graphical object propagates the request for an update to its enclosing container object, because that object has more information about its context.

Questions

What pattern(s) would you use in combination with the Chain of Responsibility? Why?

Flyweight

Flyweight

Category

- Structural

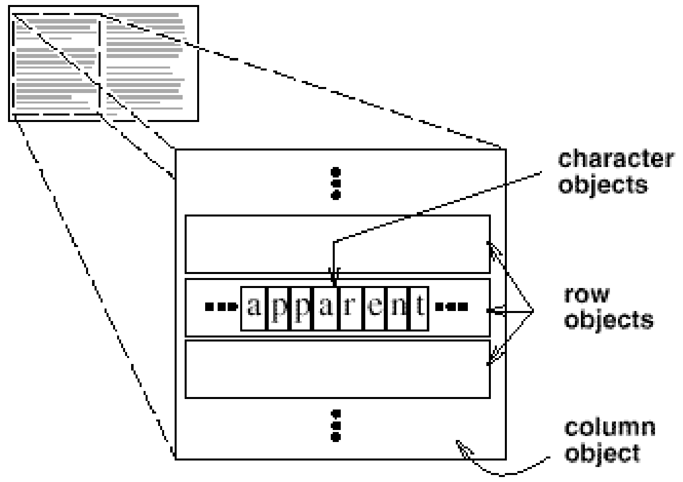
Intent

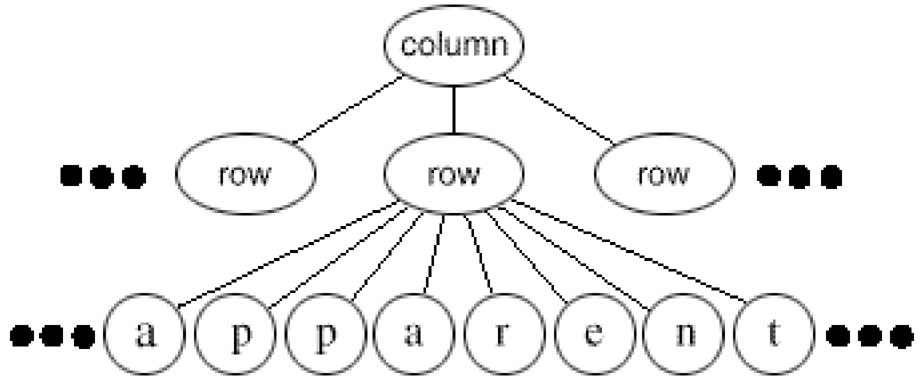
- Use sharing to support large numbers of fine-grained objects efficiently.

Motivation

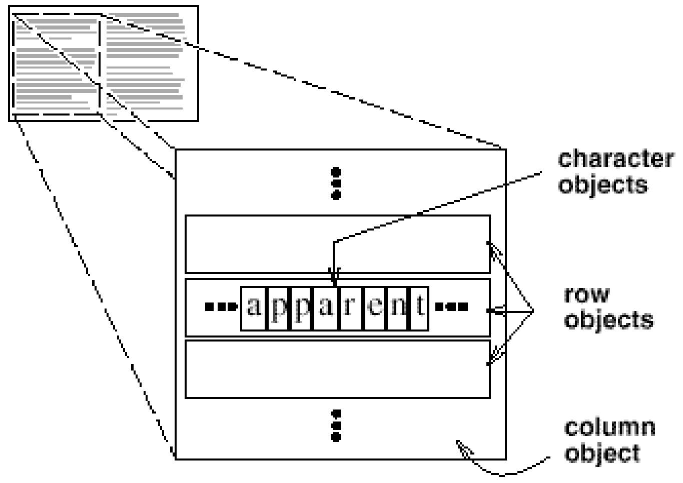
- Some applications benefit from using objects in their design but a naive implementation is prohibitively expensive because of the large number of objects.
- For example a document editor uses an object for each character in the text.

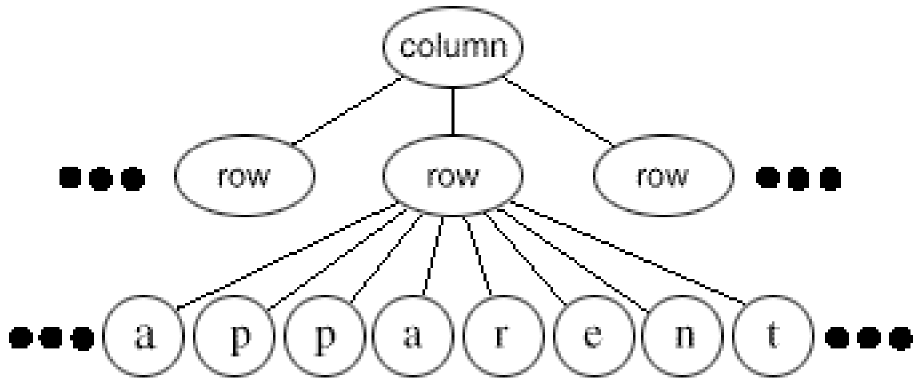
Motivation (cont)

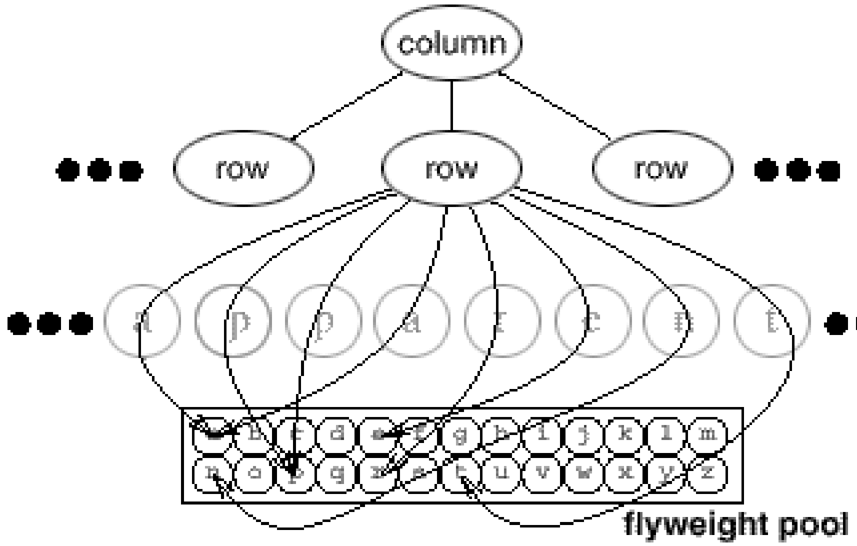




Motivation (cont)





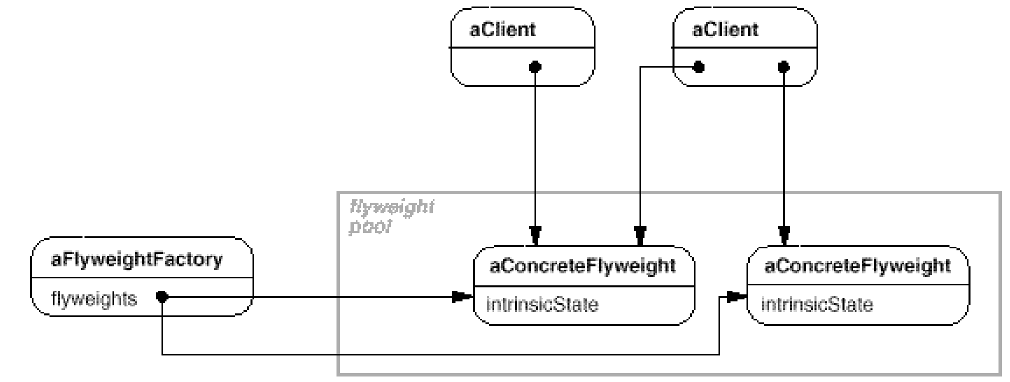


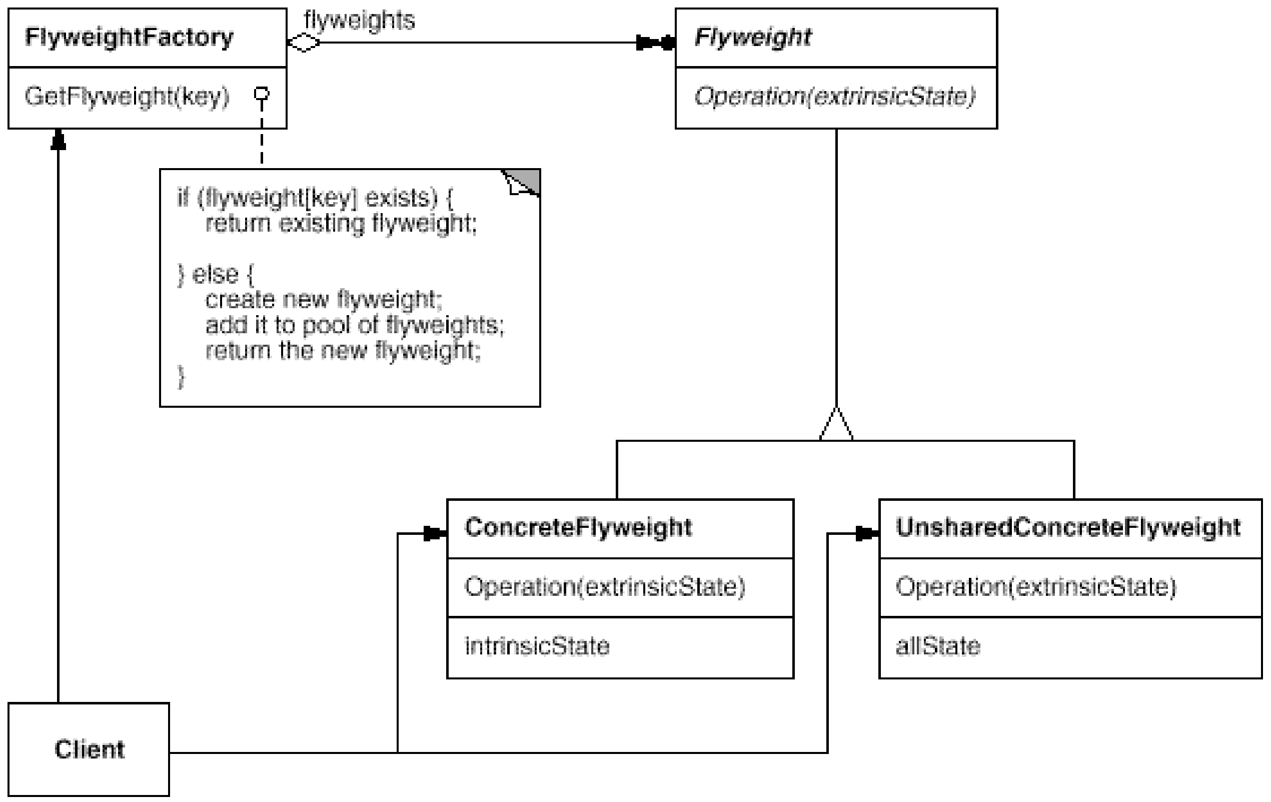
Applicability

Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage cost is high because of the quantity of objects.
- Most objects 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.

Structure




FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return existing flyweight;

} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

Participants

Flyweight

- Declares an interface through which flyweights can receive and act upon extrinsic state.

Concrete Flyweight

- Implements the flyweight interface and adds storage for intrinsic state.
- A concrete flyweight object must be shareable, i.e. state must be intrinsic.

Unshared Concrete Flyweight

- Not all flyweights subclasses need to be shared, unshared concrete flyweight objects have concrete flyweight objects at some level in the flyweight object structure.

Participants (cont)

Flyweight Factory

- Creates and manages flyweight objects.
- Ensures that flyweights are shared properly; when a client requests a flyweight the flyweight factory supplies an existing one from the pool or creates one and adds it to the pool.

Client

- Mainrains a reference to flyweight(s).
- Computes or stores the extrinsic state of flyweight(s).

Collaborations

State that a flyweight needs to function must be characterised as either intrinsic or extrinsic. Intrinsic state is stored in the concrete flyweight object; extrinsic state is stored or computed by client objects. Clients pass this state to the flyweight when invoking operations.

Clients should not instantiate concrete flyweights directly. Clients must obtain concrete flyweight objects exclusively from the flyweight factory object to enshure that they are shared properly.

Consequences

Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state.

The increase in run-time cost are offset by storage savings which increase

- as more flyweights are shared.
- as the amount of intrinsic state is considerable.
- as the amount of extrinsic state is considerable but can be computed.

Consequences (cont)

The flyweight pattern is often combined with the composite pattern to build a graph with shared leaf nodes. Because of the sharing, leaf nodes cannot store their parent which has a major impact on how the objects in the hierarchy communicate.

Known Uses

Has been used in e.g. document editors. When first introduced in such an editor, only the style and and character code of the characters were intrinsic, while the position of the characters was extrinsic. This made the program very fast. In a document containing 180.000 characters, only 480 character objects had to be allocated.

Can also be used to abstract the look and feel of layouts. Only the objects of the flyweight pool have to be replaced to change a complete layout.

Questions

Give a non-GUI example of a flyweight.

What is the minimum configuration for using flyweight? do you need to be working with thousands of objects, hundreds, tens?

Suppose you have to implement a texteditor. The text of the texteditor consists of lines and characters on the lines.

Facade

Facade

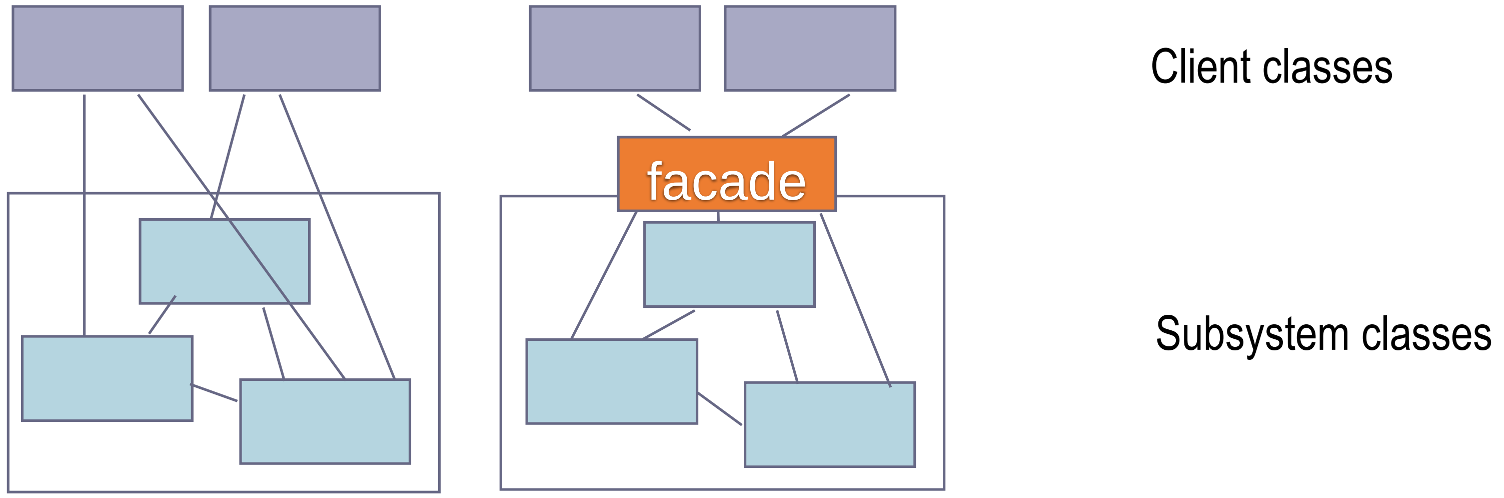
Category

– Structural

Intent

– Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation



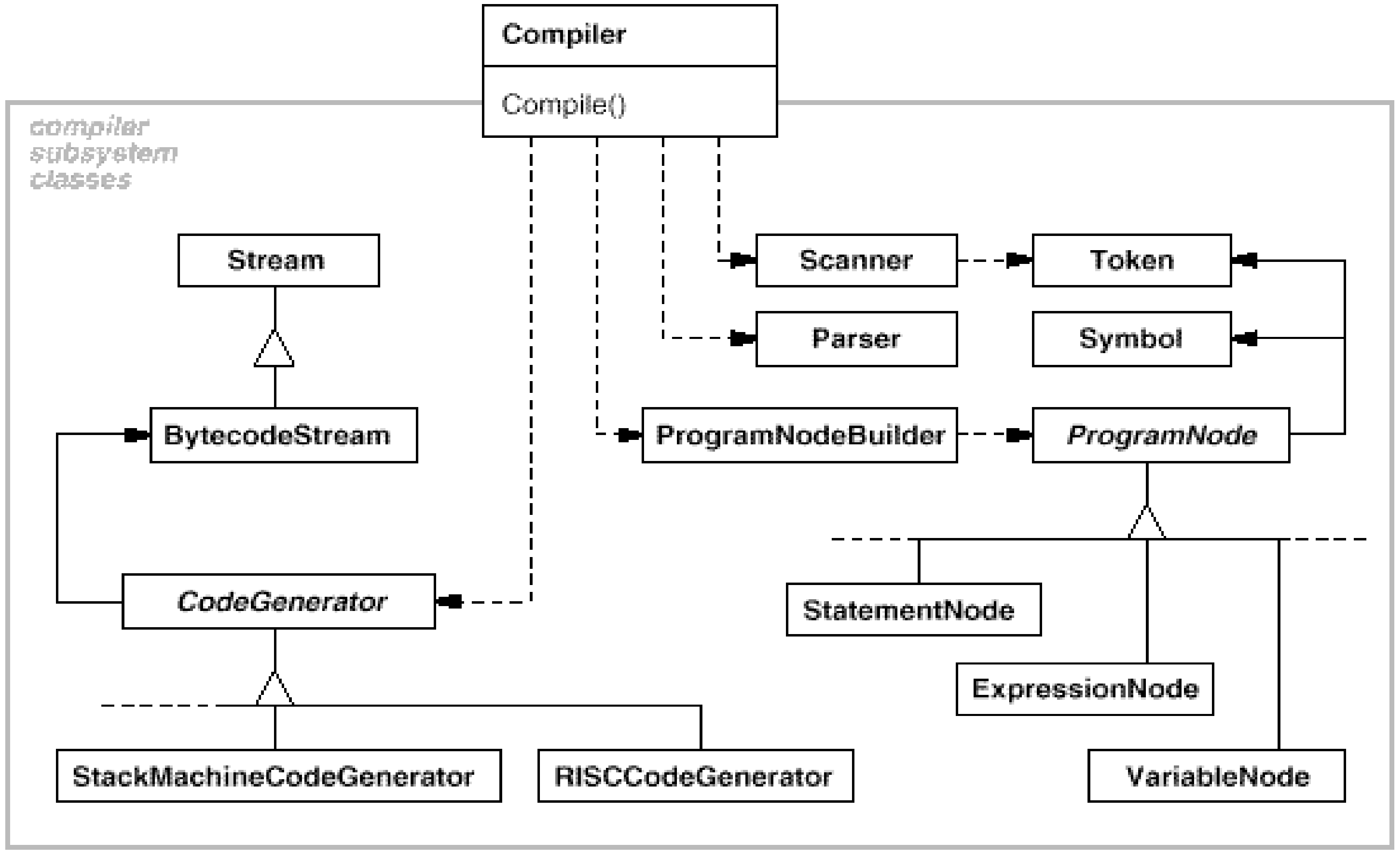
Motivation (cont)

Provide a simple interface to a complex subsystem.

Decouple a subsystem from clients and other subsystems.

Create layered subsystems by providing an interface to each subsystem level.

Example

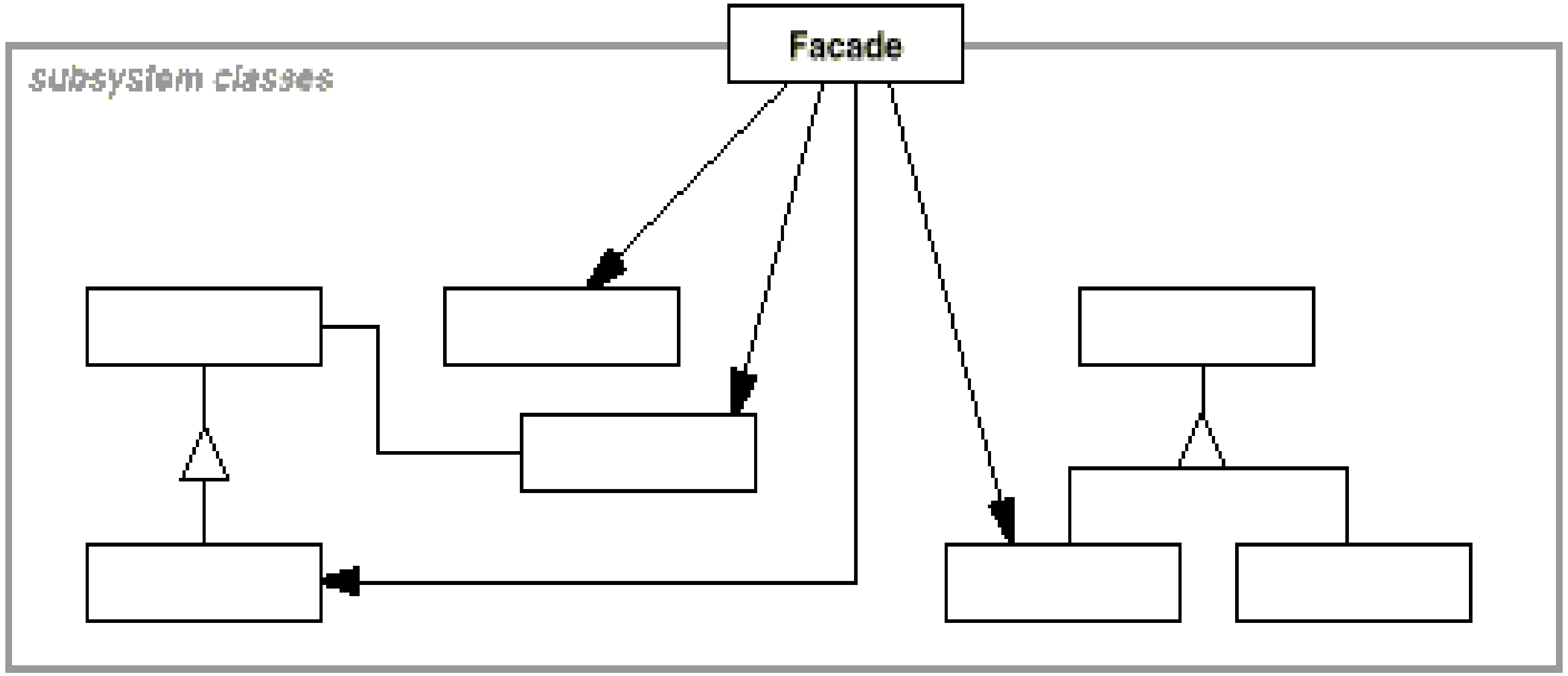

Compiler
Compile()
compiler subsystem classes
Stream
BytecodeStream
CodeGenerator
StackMachineCodeGenerator
RISCCodeGenerator
Scanner
Token
Parser
Symbol
ProgramNodeBuilder
ProgramNode
StatementNode
ExpressionNode
VariableNode

Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure



Participants

Facade

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

Subsystem classes

- implement subsystem functionality.
- handle work assigned by the Facade object.
- have no knowledge of the facade; that is, they keep no references to it.

Collaborations

Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.

Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:

- It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It promotes weak coupling between the subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
- It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Known Uses

We have seen the compiler example, but this pattern can be used for other complicated frameworks as well.

Questions

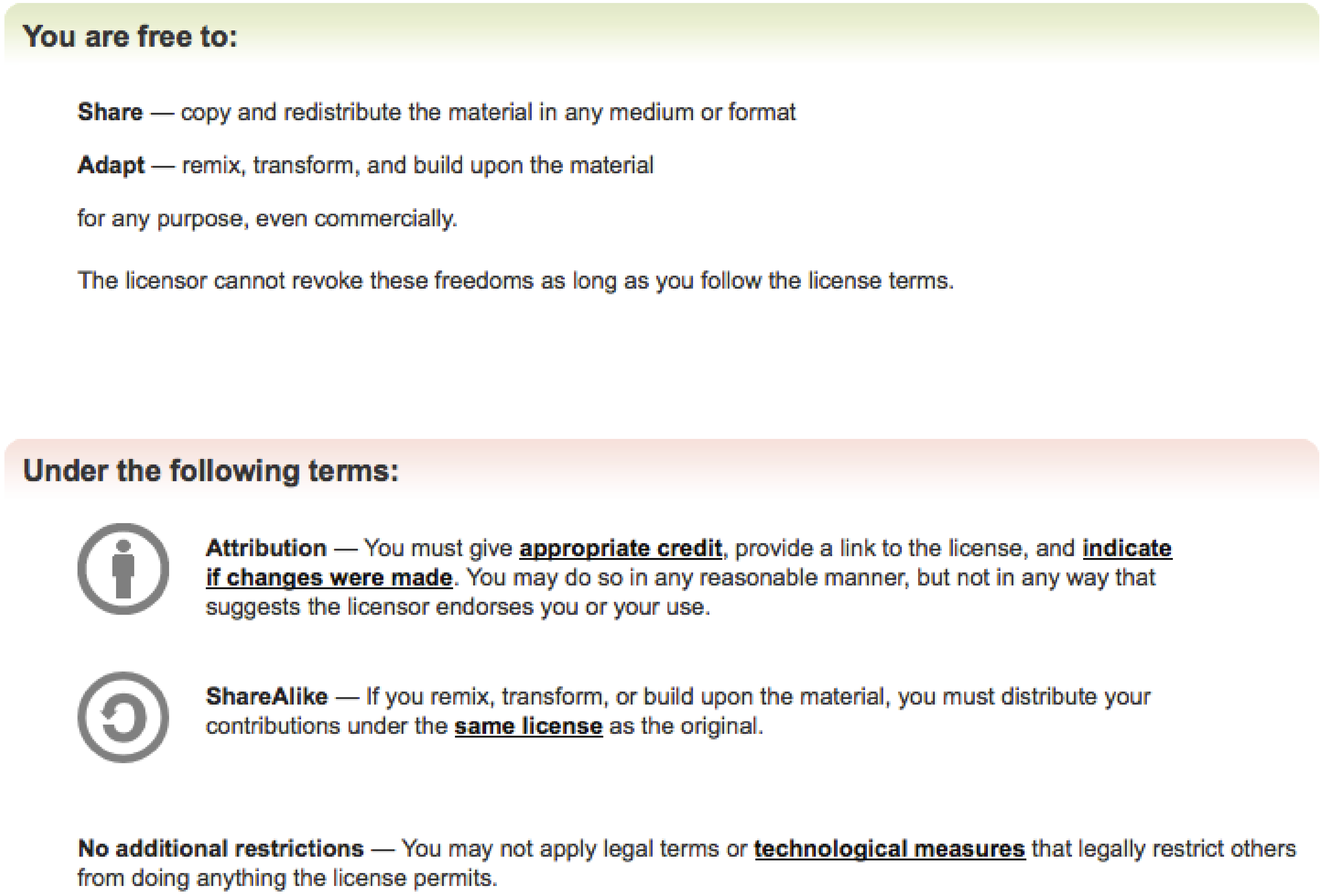
Describe the differences between Facade and Adapter.

How complex must a sub-system be in order to justify using a facade?

What are the additional uses of a facade with respect to an organization of designers and developers with varying abilities? What are the political ramifications?

License: Creative Commons 4.0

http://creativecommons.org/licenses/by-sa/4.0/



You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.