Introduction to Software Engineering

Design patterns

Philippe Lalanda

Philippe.lalanda@imag.fr

http://membres-liglab.imag.fr/lalanda/

Outline

- Definition
- Creation patterns
- Decoupling patterns (composition vs. inheritance)
- Adaptation patterns
- Miscellaneous
- Conclusion

Design - reminder

- Design is an issue
- Processes define activities and organization
  - But nothing on how they should be realized
- Methods define notations, diagram types
  - But nothing on how diagrams should be built
- Can we only rely on experience?

Design pattern - definition

- A design pattern is the combined description of a problem and a well known solution
  - <problem, solution>
- The solution must have been validated in numerous projects
  - validated by experience

Design pattern - interest

- Design pattern = design reuse
  - It is easier to reuse a design solution than a piece of code
  - It is a way to describe good design practices and to transmit knowledge gained through experience

- Patterns are known by designers
  - A language is created and shared

Design pattern - description

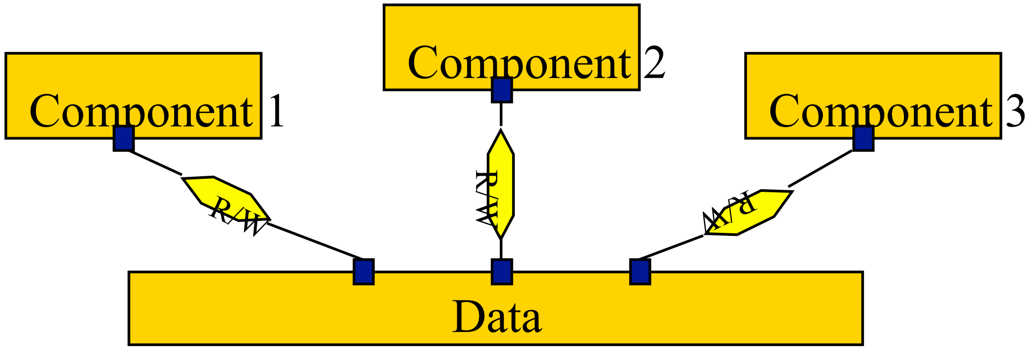
- Name of the pattern
- Global description – what is its purpose
- Problem – what is the problem solved by the pattern
- Solution – high level description possibly with diagrams
- Implementation hints – in appropriate language
- Advantages and limits – nothing is perfect!
- Example – from a real situation

Design pattern - scope

- Many kinds of patterns
  - Architectural patterns (style)
  - Design patterns (object)
  - Language patterns (idioms)
  - Object Analysis patterns (object models)
- Many domains
  - Enterprise Integration Patterns (EIP)
  - Distributed System Patterns
  - SOC patterns

Architectural patterns

- Pattern-Oriented Software Architecture: On Patterns and Pattern Languages (Relié)
  Frank Buschmann, Kevlin Henney, Douglas C. Schmidt
  John Wiley & Sons (13 avril 2007)

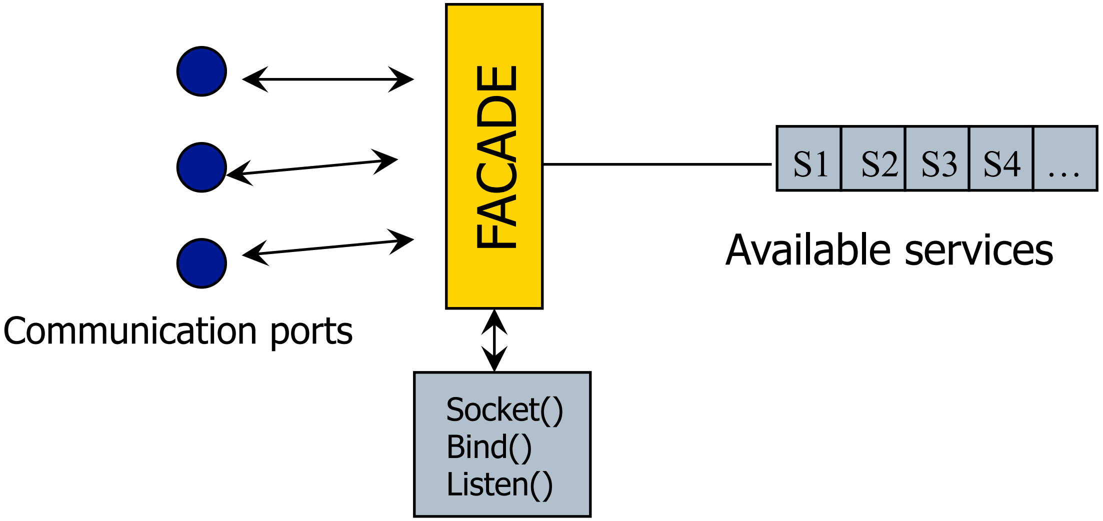


PATTERN-ORIENTED SOFTWARE ARCHITECTURE

On Patterns and Pattern Languages

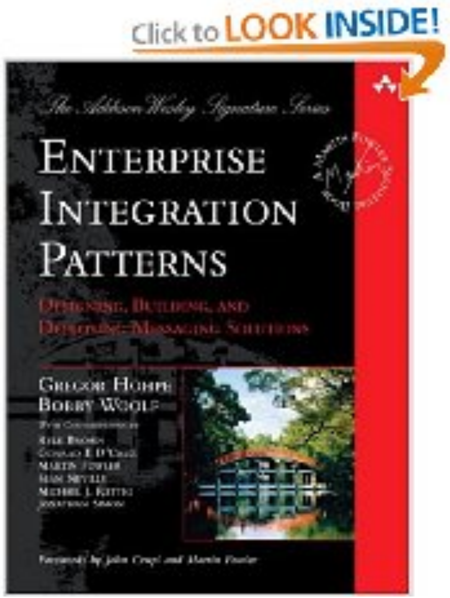
Distributed system patterns

- Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing
  Frank Buschmann, Kevlin Henney, Douglas C. Schmidt
  John Wiley & Sons (16 mars 2007)



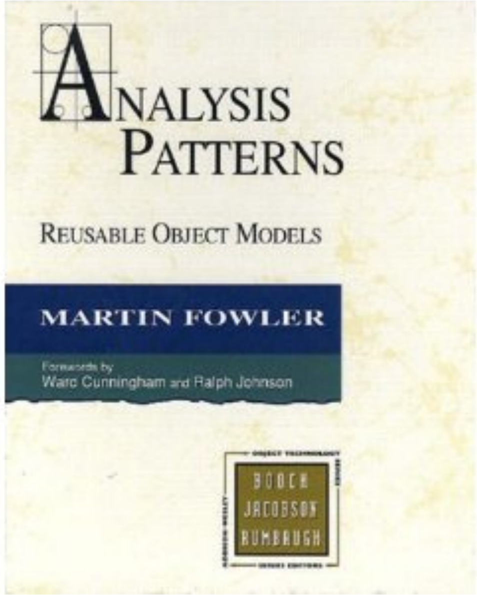
Integration patterns

- Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions
  Gregor Hohpe, Bobby Woolf
  Addison-Wesley Signature Series



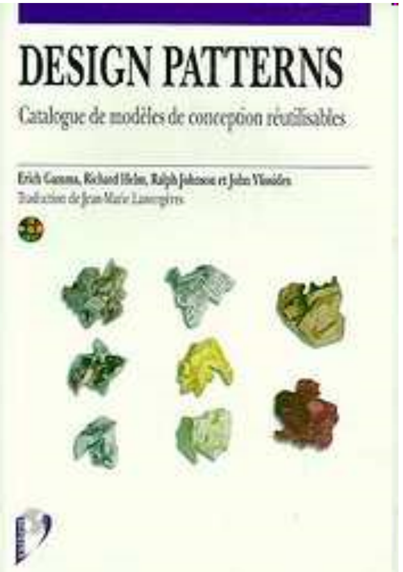
Analysis patterns

- Analysis Patterns: Reusable Object Models
  Martin Fowler
  Addison-Wesley



Design pattern – main reference of this lecture

- Elements of Reusable Object-Oriented Software”
  E. Gamma, R. Helm, R. Johnson, J. Vlissides
  Addison-Wesley, 1995


DESIGN PATTERNS
Catalogue de modèles de conception réutilisables
Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides
Traduction de Jean-Marie Lasvergères

Outline

- ❑ Definition
- ❑ Creation patterns
- ❑ Decoupling patterns (composition vs. inheritance)
- ❑ Adaptation patterns
- ❑ Miscellaneous
- ❑ Conclusion

Creation patterns

- How instances can be created in OO systems

- The main idea behind these patterns is to create objects through high level interfaces
  - Without knowing what is created beyond the interfaces
  - A form of information hiding
  - Improve quality and evolution

Factory method

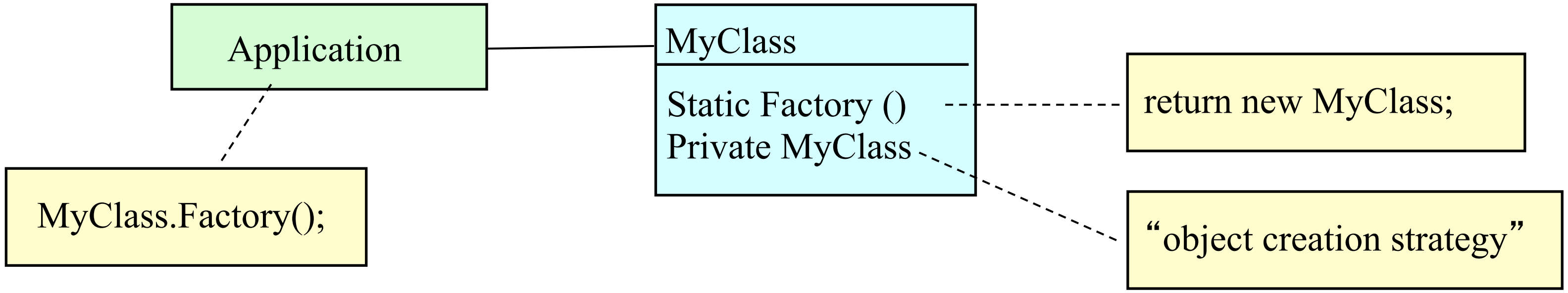
Hide creation complexity

Name Factory method

Problem Object creation can be technically complex (threads, attribute assignment, etc.). This should be hidden to clients.

Solution Define a static method called factory which calls the effective creator which is kept private. The static method returns an instance.

Only the creation interface (the factory) is known.



Consequences Complexity of object creation is hidden to the client.
The objects creation is well mastered (safe and easier to maintain) .
Can be extended to all lifecycle related operations (reset, destruction, etc.)

Factory class

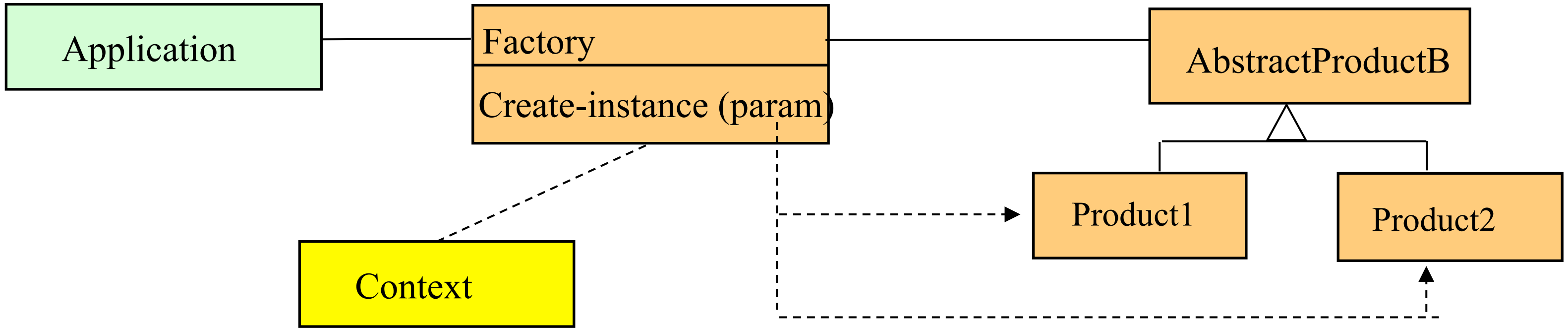
Hide complexity + delay decision about instance to create

Name Factory class

Problem The class of the object to be created is not known by the client application

Solution Create a factory class defining a *create-instance* method and let this method decide which class to instantiate.

Depending on the context (including parameters), the right instance will be created.



Consequences Complexity of object creation is hidden to the client (like in usual factories)
The context-dependence is hidden to the client.
It is easy to add new types of objects (also context-dependent) : only the create-instance method must be updated (the application is unaware of that).

Abstract factory

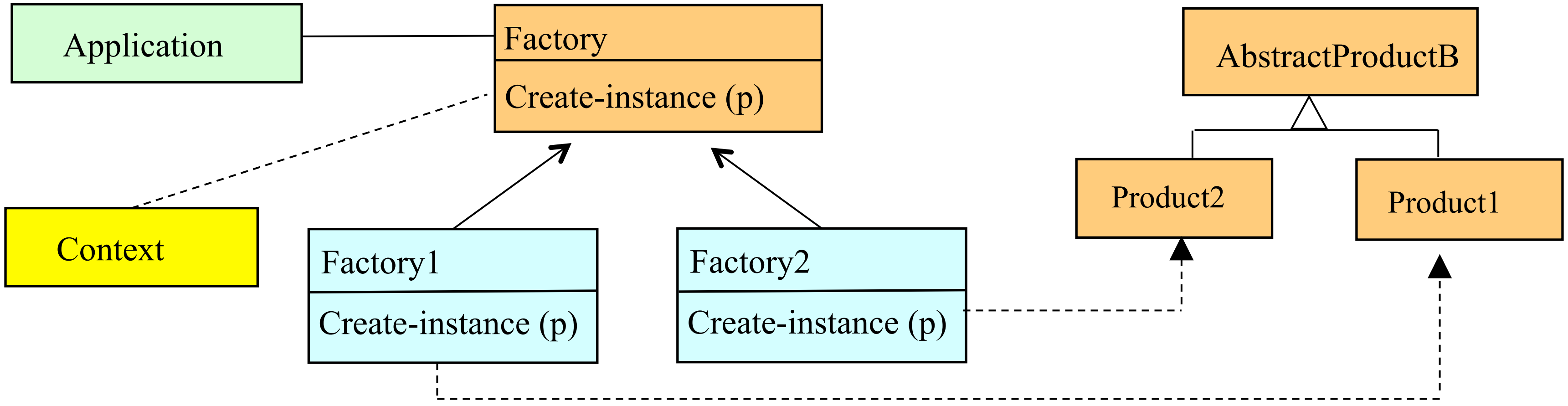
Several creation strategy

Name Abstract factory

Problem The class of the object to be created is not known by the client application

Solution Create an abstract factory class defining the factories and let subclasses decide which class to instantiate.

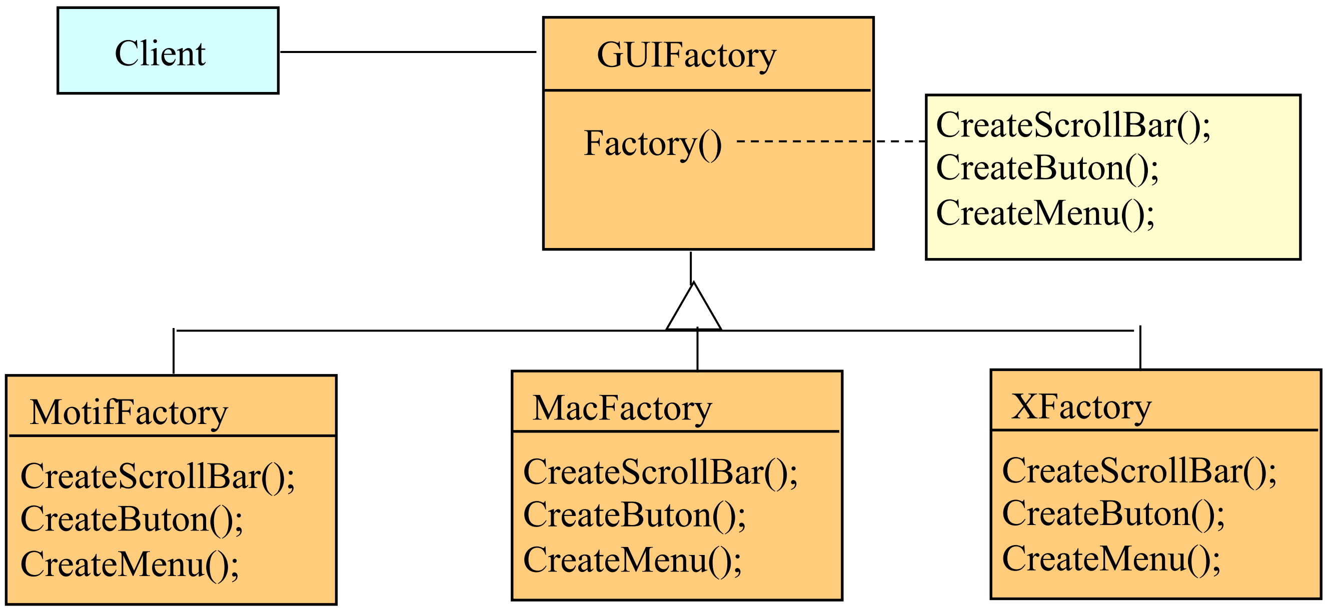
Depending on the context (including parameters), the right instance will be created.



Consequences Complexity of object creation is hidden to the client (like in usual factories)
The context-dependence is hidden to the client.
It is easy to add new types of objects (also context-dependent) : only the concrete factories must be updated (the application is unaware of that).

Abstract Factory: example

- Consider a user interface toolkit that supports multiple look-and-feel standards
  - To be portable across look-and-feel standards an application cannot hard code its widgets for a particular look and feel
  - Depending on the context, the client will call the right factory



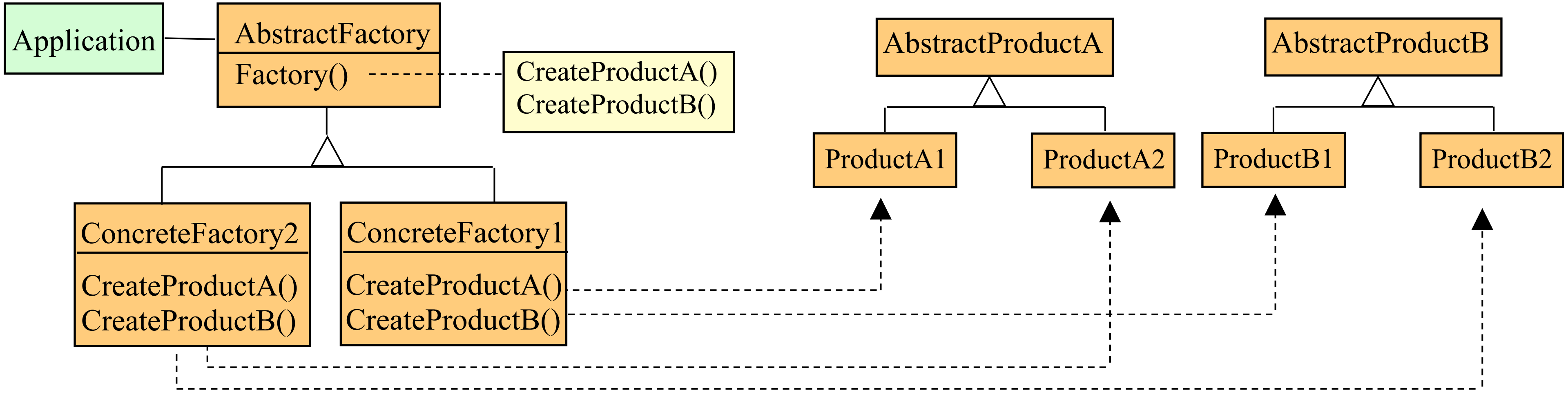
Abstract Family Factory

Create a family of objects

Name Abstract Factory

Problem Some objects are related and must be created coherently (objects family)

Solution Provide an abstract factory for creating families of related or dependent objects.
This abstract class defines all the object to be created jointly.
The subclasses decide which set of classes to instantiate..

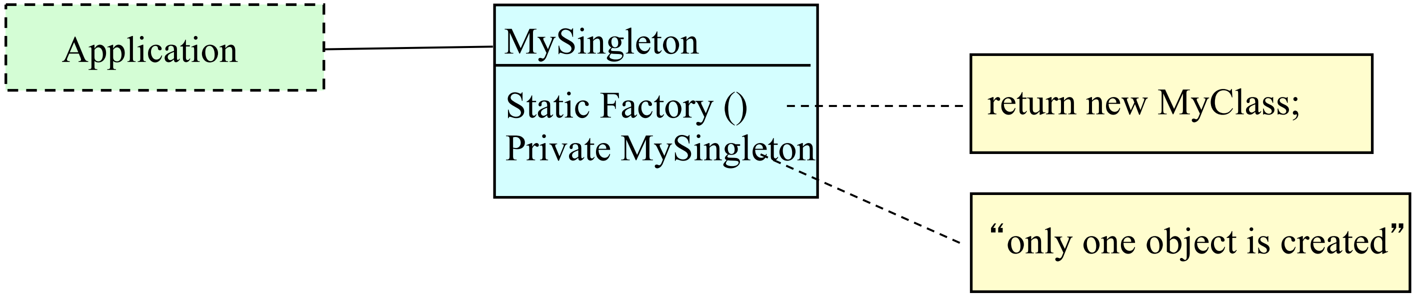


Consequences The programmer has a context to create coherent sets of classes
Extensibility: New product families can be added easily

Singleton

Hide instance uniqueness

Name	Singleton
Problem	Only a single instance of a given object is allowed
Description	Ensure a class has only one instance and provide a global point of access.
Solution	Based on the Factory pattern (here, the object creation strategy is specific)



Consequences Avoid polluting the name space with global variables
The pattern can be changed easily and allow more than one instance
Strict control on how an when a singleton is accessed is possible

Prototype

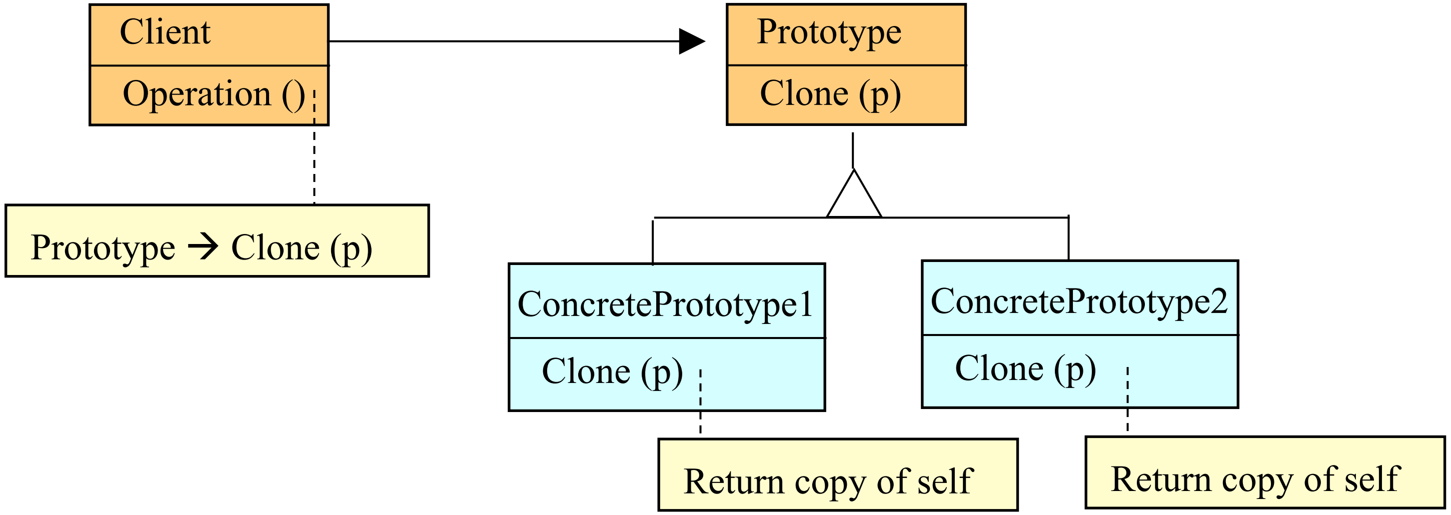
Cloning objects

Name Prototype

Problems Many (almost) similar objects have to be created

Description Specify the kinds of object to create using a prototypical instance, and create new objects by copying this prototype with a specific configuration.

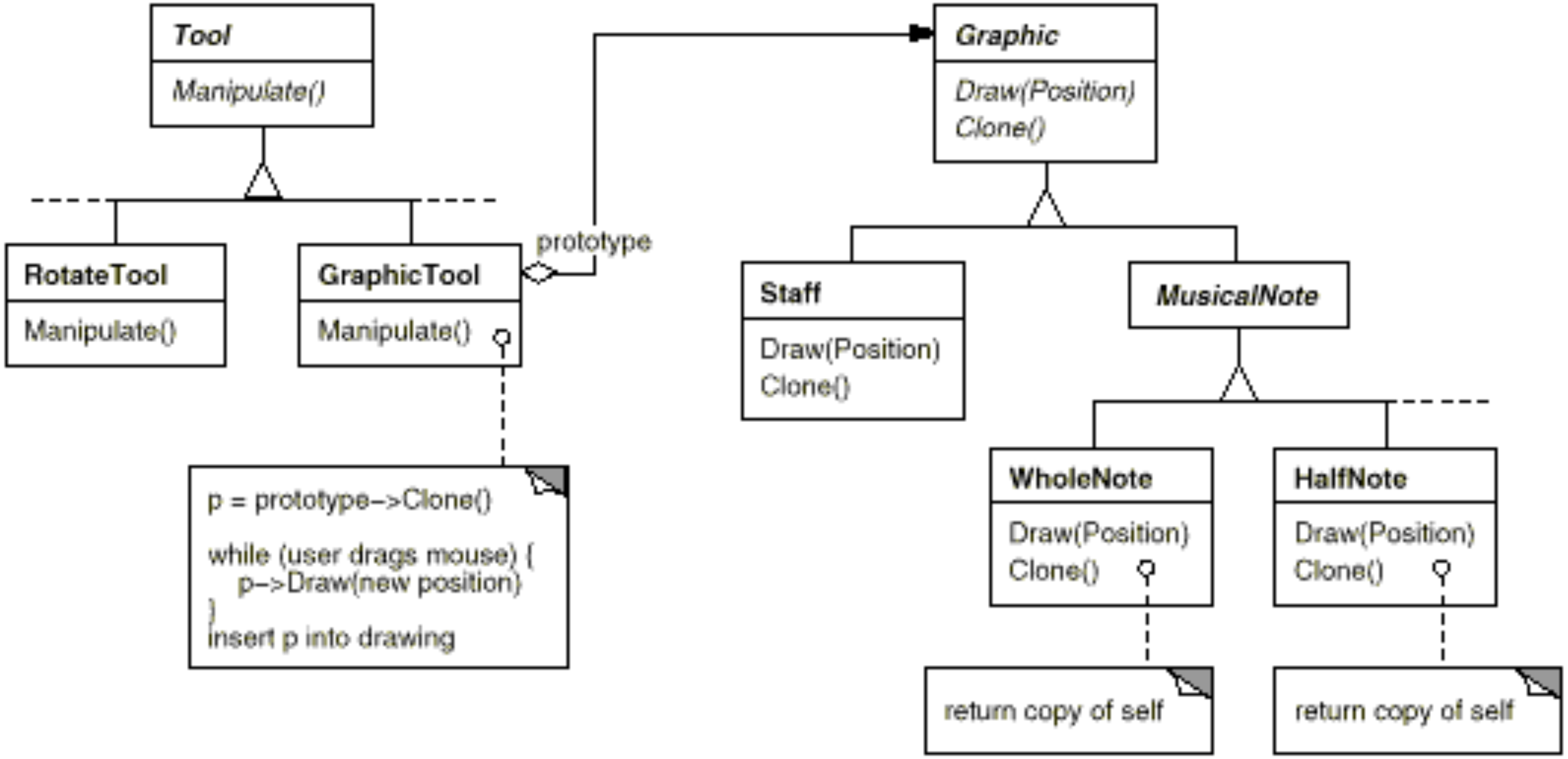
Solution



Consequences It reduces subclassing (Factory creates a hierarchy of creators)
It allows the creation of new objects by varying values or structures

Prototype: example

- Consider building an editor for music stores customizing a general framework for graphical editors



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

Builder

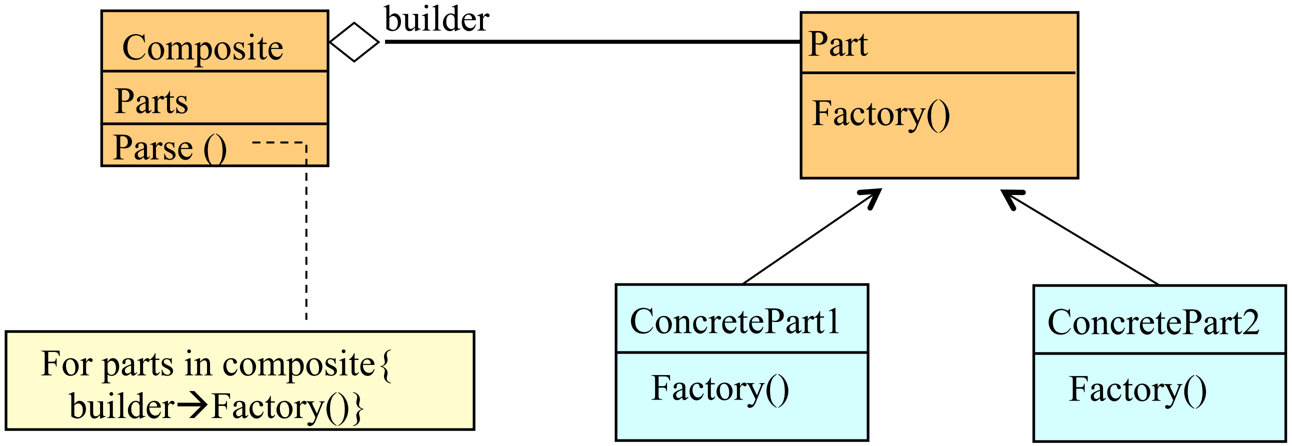
Building complex objects

Name Builder

Problem A complex object is made of heterogeneous parts

Description Separate the construction of a complex object from the creation of its parts so that the same construction process can lead to different composites.

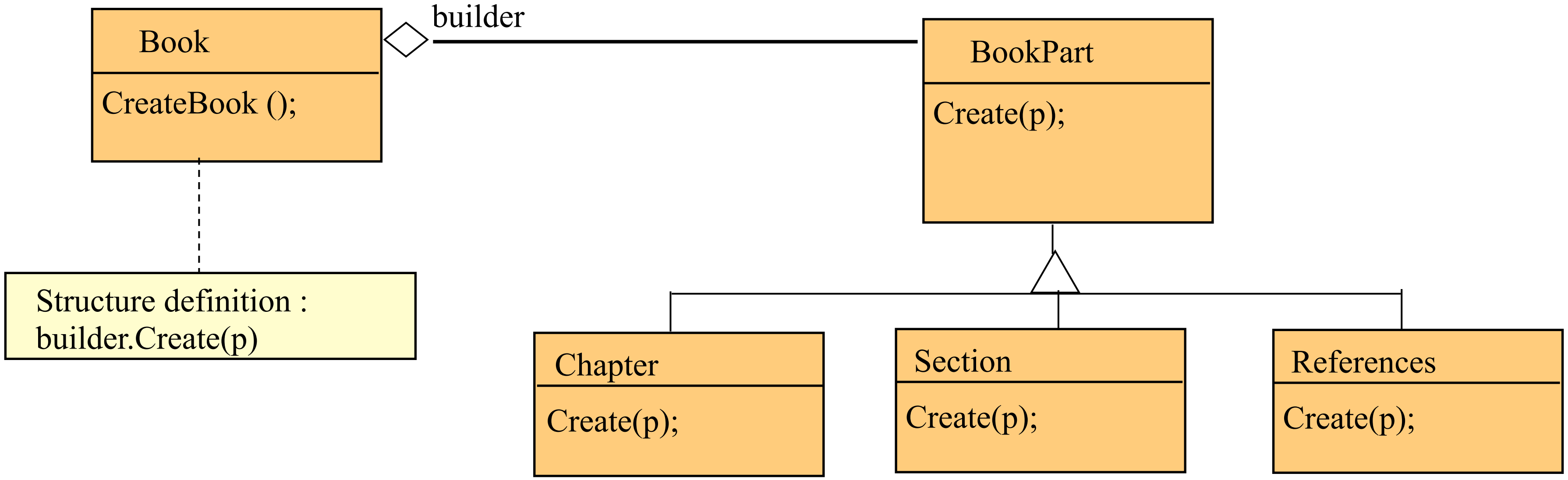
Solution



Consequences The construction process is expressed in the composite
Extensibility: new parts can be added easily.
Creation complexity of parts is hidden

Builder: example

- Building a representation of a book



Outline

- Definition
- Creation patterns
- Decoupling patterns (composition vs. inheritance)
- Adaptation patterns
- Miscellaneous
- Conclusion

Decoupling patterns

- How related classes should be structured

- The main idea behind these patterns is to create multiple hierarchies to deal with different aspects
  - Use of composition combined with subclassing

Bridge

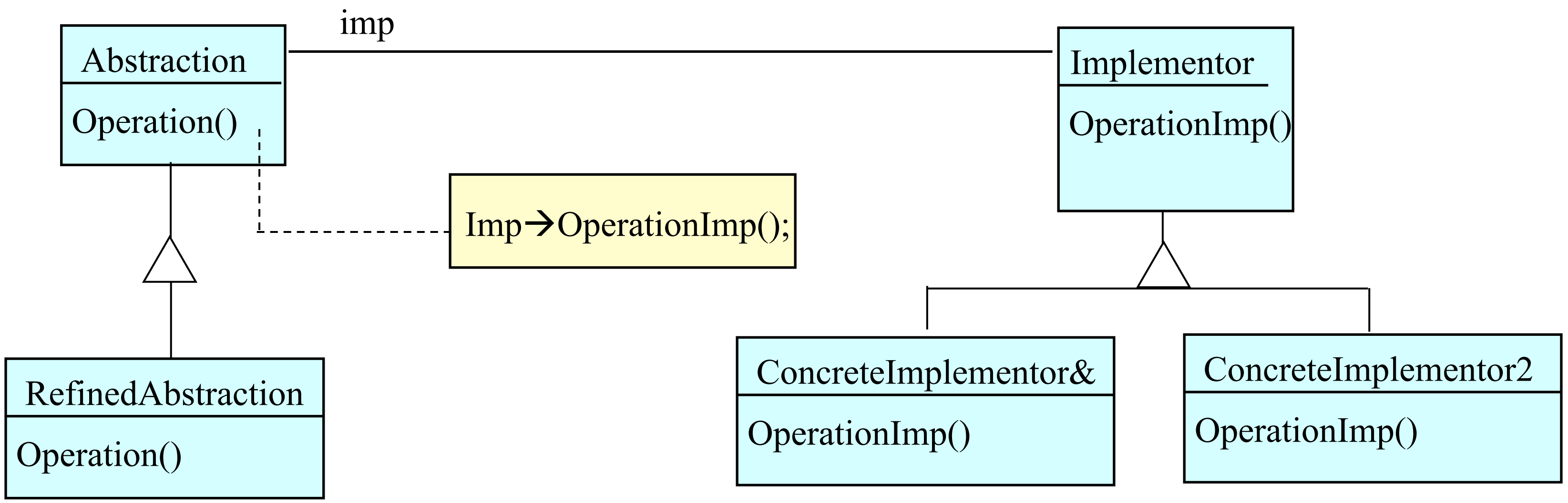
When subclassing is not enough!

Name Bridge

Problem Not always possible to use inheritance to define several implementations for an abstraction

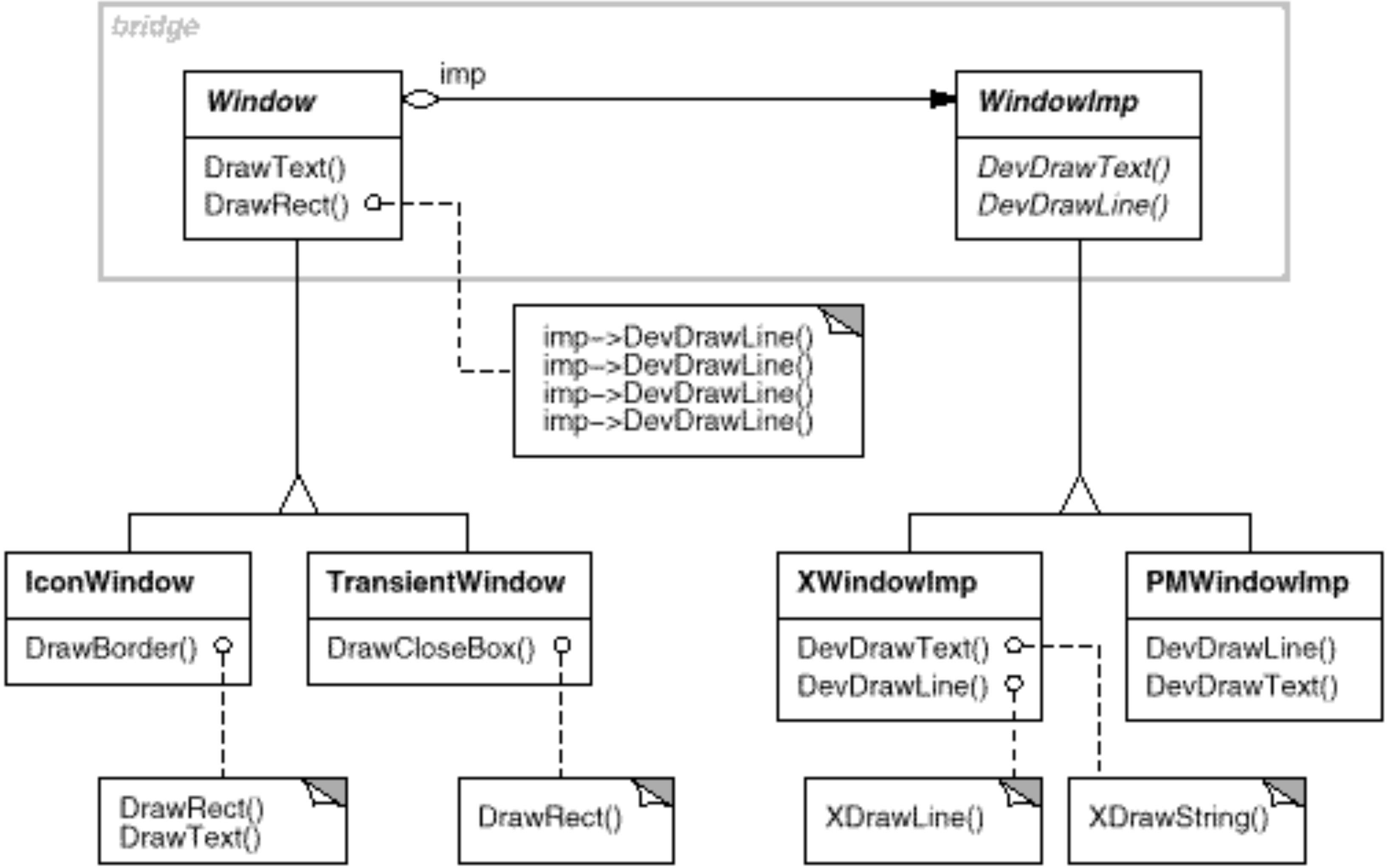
Description Use composition to decouple an abstraction from its implementation

Solution



Consequences Both abstraction and implementations are extensible by subclassing
Abstraction and implementation can be modified independently

Bridge: example



Strategy

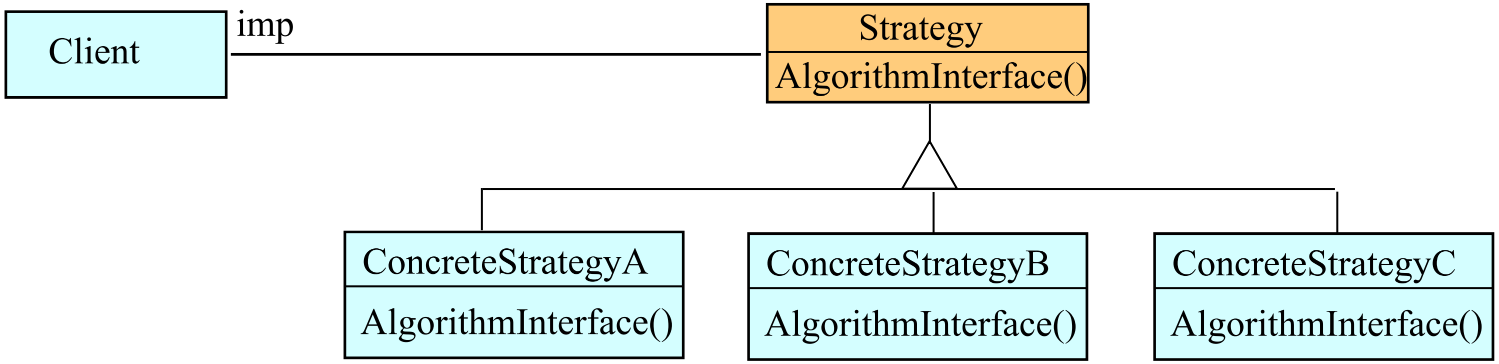
Several implementation for an algorithm

Name Strategy

Problem A class defines many behaviors and this appears as multiple conditional statements in its operations.

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

Solution



Consequences An alternative to subclassing.
With inheritance, an algorithm can't vary dynamically
Strategies eliminate conditional statements
Pb: clients must be aware of different strategies

State

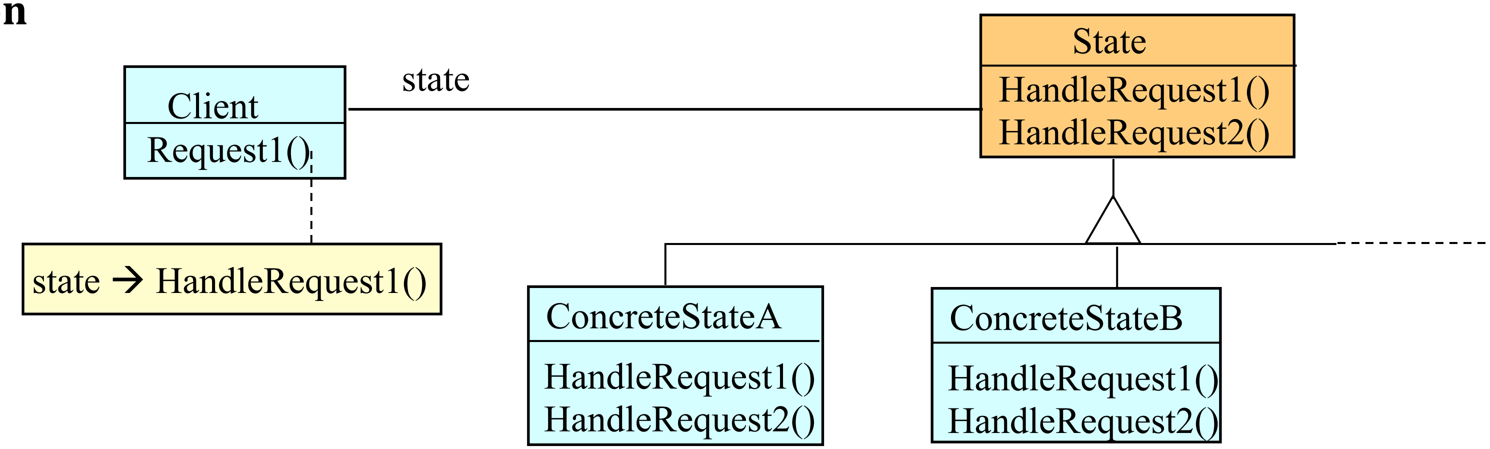
State dependant behaviours

Name State

Problem Allow an object to change its behavior when its internal state changes without too many conditional statements

Description Introduce an abstract class to represent the states of an object. Redefine the state-dependent behaviors in the sub classes.
Change the state object used when the state changes

Solution



Consequences It localizes state-specific behavior. It puts all behavior associated with a particular state into one object that is changed dynamically

New states can be added easily

It makes state transition explicit (otherwise state are represented by internal data values and transitions are not explicit)

State objects can be shared

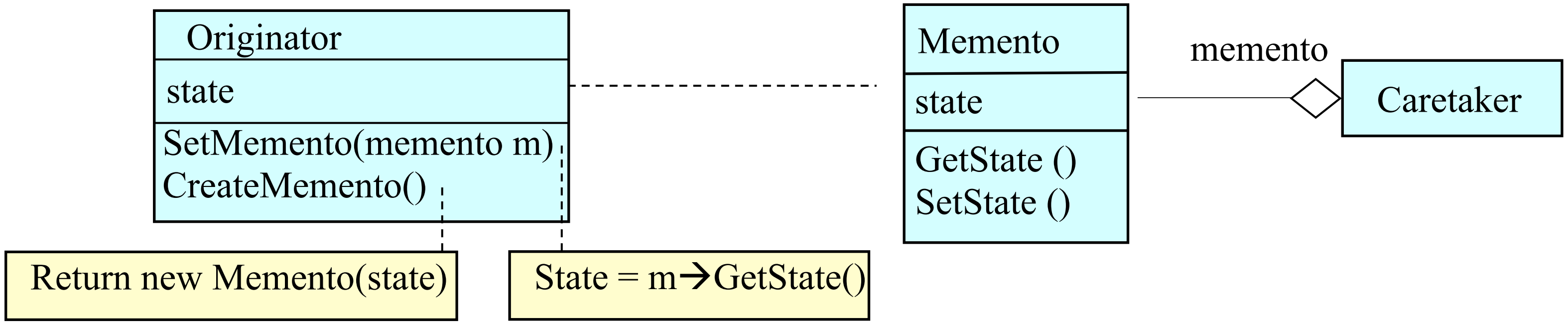
Memento

Name Memento

Problem It is sometimes necessary to record the internal state of an object for undo operations for instance. A direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Description Capture and externalize an object's internal state without violating encapsulation so that the object can be restored later.

Solution



Consequences It preserves encapsulation boundaries

It simplifies originator. In other designs, originators keep the versions of internal state that clients have requested. Having clients manage the state they ask for simplifies originator and keeps clients from having to notify originators when they are done

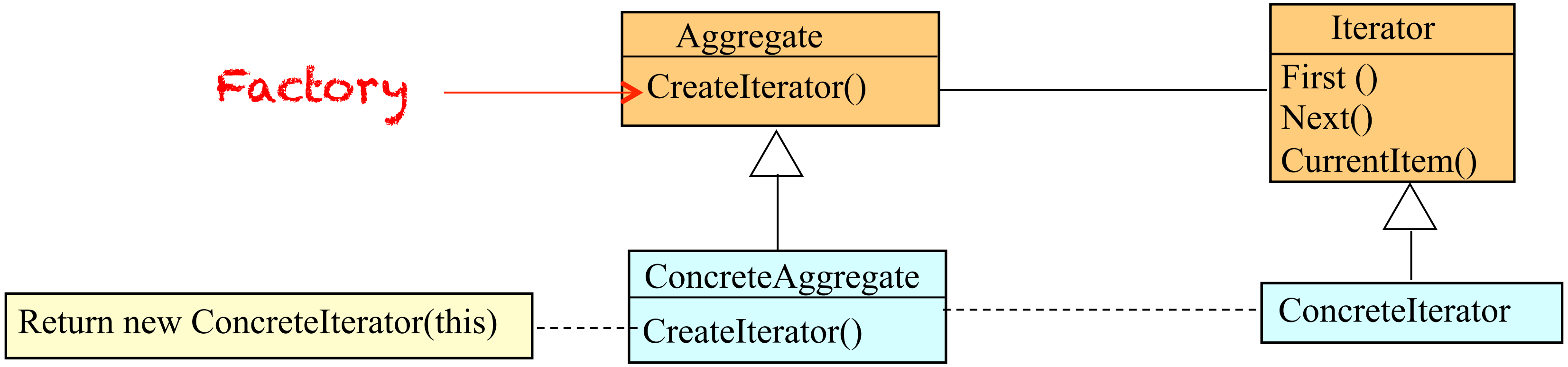
Using mementos can be expensive (copy of large amount of information)

Iterator

Decouple aggregates and their traversal

Name	Iterator
Problem	An aggregate object (like a list) should give a way to access its elements without exposing its internal structure. Also, the client might want to traverse the aggregate in different ways.
Description	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Solution



Consequences

It supports variation in the traversal of aggregate (replace the iterator instance)

Iterators simplify the aggregate interface (the iterator's interface not needed here

More than one traversal can be pending on an aggregate (an iterator keeps track of its own traversal state).

Outline

- ❑ Definition
- ❑ Creation patterns
- ❑ Decoupling patterns (composition vs. inheritance)
- ❑ Adaptation patterns
- ❑ Miscellaneous
- ❑ Conclusion

Adaptation patterns

- How changes can be incorporated smoothly
- The main idea behind these patterns is to define stable parts that can be extended
  - Abstract classes

Object Adaptor

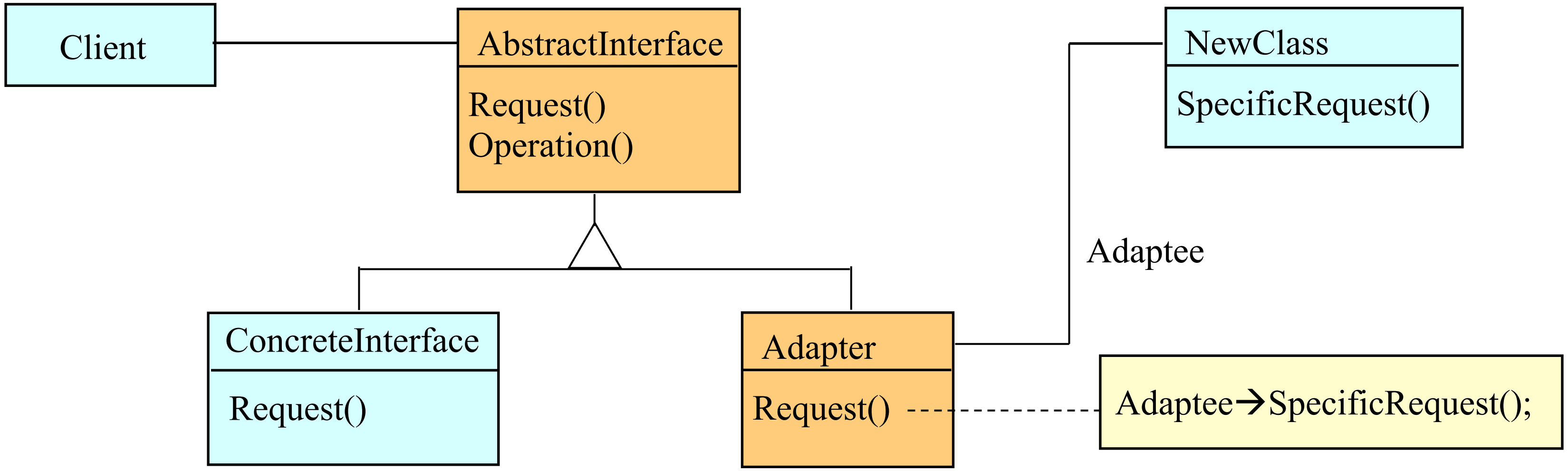
Create a slightly different Interface (with an object)

Name Object Adaptor (or wrapper)

Problem A client needs an interface different from the available ones

Description Create an abstract class for the all the classes providing the expected interfaces and use an adapter to integrate different interfaces

Solution



Consequences Transparent for client.

Avoid code duplication and class multiplication (operation not redefined)

Flexibility: a different adaptation can be defined easily

Class Adaptor

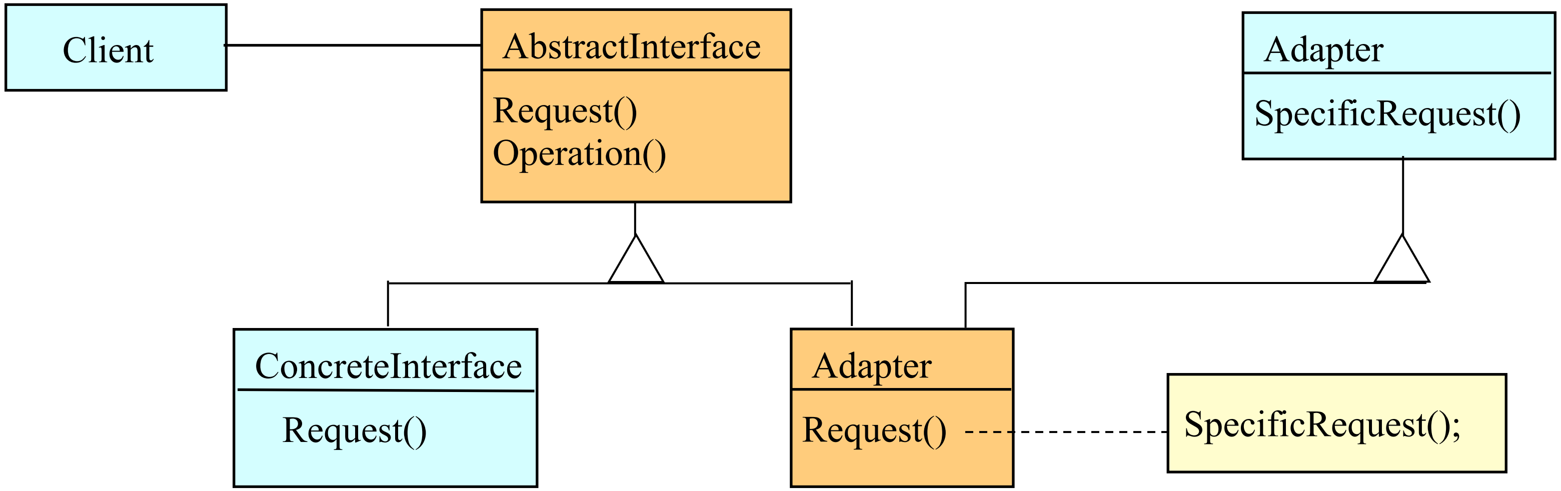
Create a slightly different Interface (with a class)

Name Class Adaptor (or wrapper)

Problem A client needs an interface different from the available ones

Description Create an abstract class for the all the classes providing the expected interfaces and use an adapter to integrate different interfaces

Solution



Consequences Transparent for client.

Avoid code duplication and class multiplication (operation not redefined)

Flexibility: a different adaptation can be defined easily

Decorator

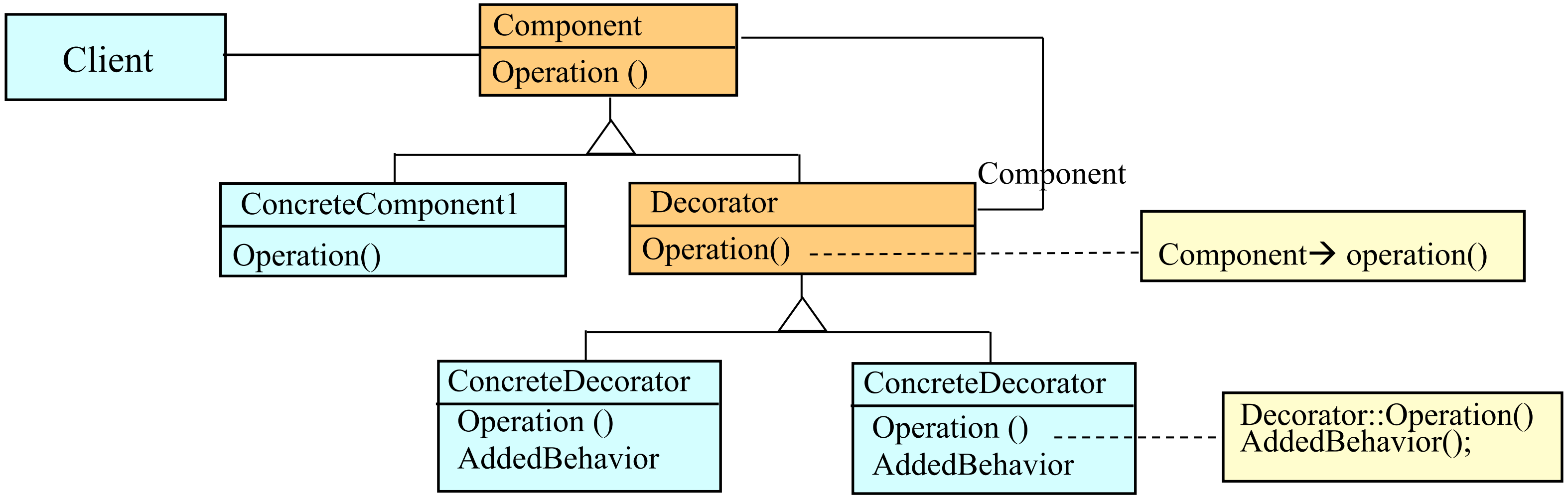
Add new features to an object

Name Decorator

Problem Dynamically provide additional functionalities to an object when subclassing is impractical (explosion of subclasses to support every combination)

Description Decorator subclasses are free to add operation with specific functionalities

Solution



Consequences Add/remove responsibilities to individual objects dynamically and transparently
Avoid trying to support all foreseeable features in complex hierarchy
No code duplication

Outline

- Definition
- Creation patterns
- Decoupling patterns (composition vs. inheritance)
- Adaptation patterns
- Miscellaneous
- Conclusion

Proxy pattern

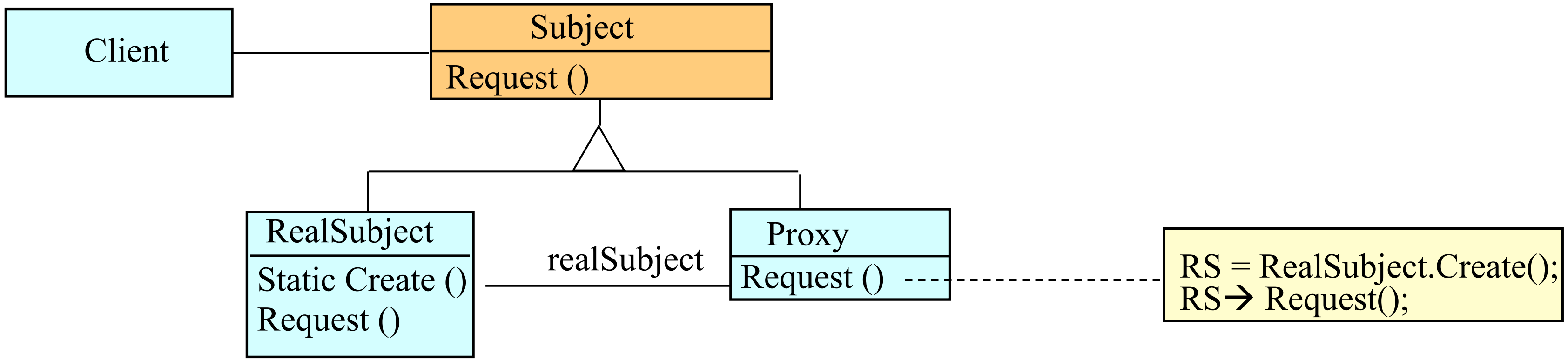
Creating fakes

Name Proxy

Problem Sometimes it is necessary to create expensive objects on demand
How can we delay the objects creation till the right moment?

Description Use another object, called a proxy, as a stand-in for the real object and create the expensive object only when necessary

Solution

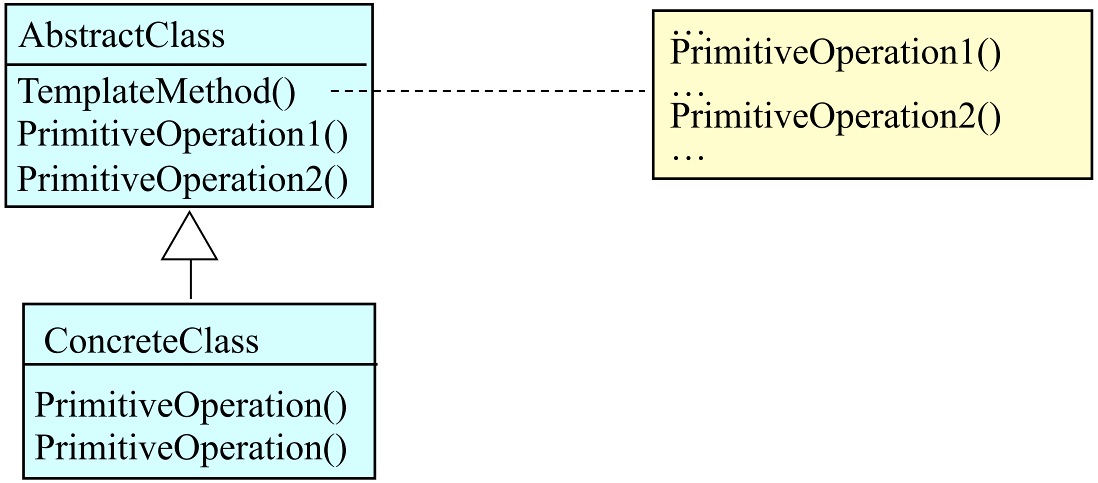


Consequences Introduction of a level of indirection
Can hide the fact that an object resides in a different address space
Add. actions, including optimizations, can be performed when an object is accessed
Can maintain a single copy of an expensive object and duplicate it when modified

Template pattern

Force steps to be done

Name	Template
Problem	Some operations part must be repeated in many subclasses
Description Solution	Define steps of the operations using abstract operations



Consequences	Avoid code duplication: a fundamental technique for code reuse Lead to an inverted control structure: a parent class calls the operations of the subclass (and not the other way around) It is important to specify which operations must be overridden

Façade pattern

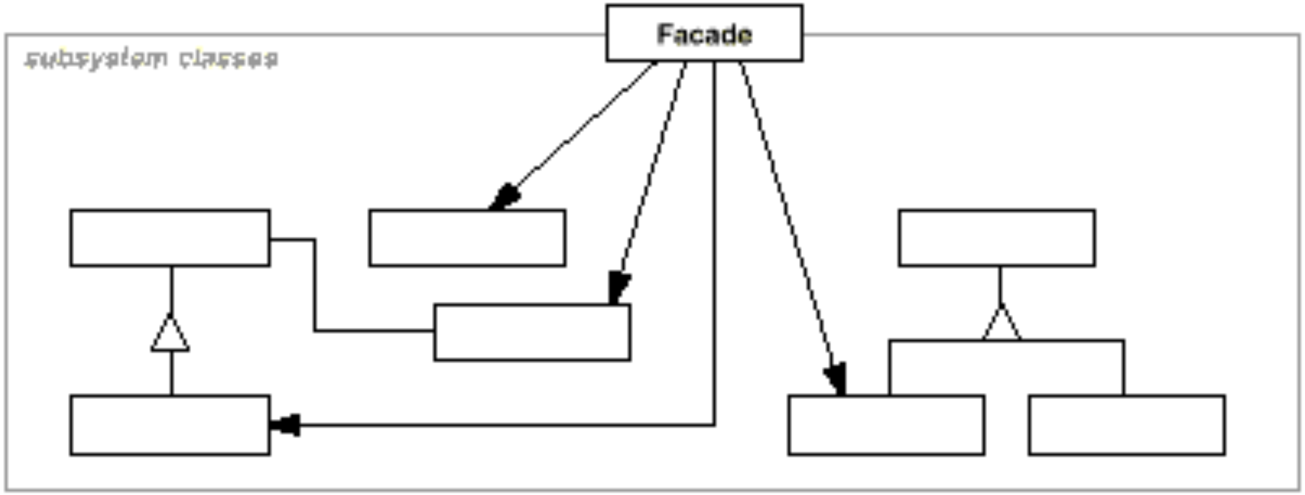
Provide a unified view of many interfaces

Name Facade

Problem Structuring a system into subsystems reduce complexity. It also increases the number of interfaces to deal with.

Description Provide a unified interface to a set of interfaces

Solution



Consequences Promote weak coupling between the subsystem and its clients
Shield clients from subsystems components
It does not prevent client from using subsystem classes.

Outline

- ❑ Definition
- ❑ Creation patterns
- ❑ Decoupling patterns (composition vs. inheritance)
- ❑ Adaptation patterns
- ❑ Miscellaneous
- ❑ Conclusion

Conclusion

- Patterns make room for evolution
- Important mechanisms
  - Combined use of composition and inheritance
  - Interface based programming

Patterns drawbacks

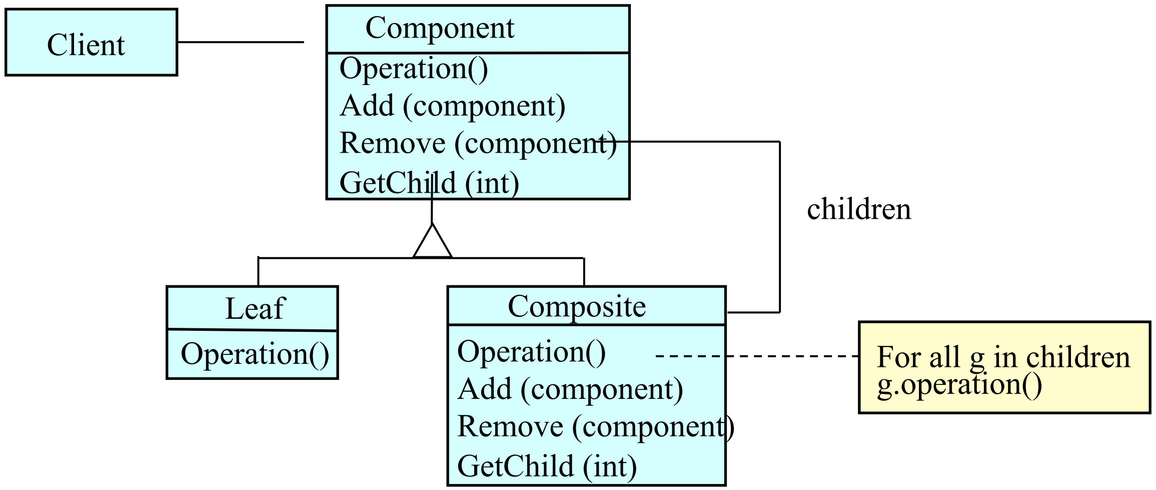
- Deceptively simple
  - Easy to understand and remember patterns
  - But ... Hard to actually use them correctly
- Pattern overload
  - Using pattern is not an end in itself
  - It is a means top be appropriately used
- Labor-intensive
  - No immediate benefits

Annex

- More patterns

Composite

Name	Composite
Problem	Objects and composites are treated differently in most codes
Description	Organize objects into tree structures to represent whole-part hierarchies
Solution	



Consequences

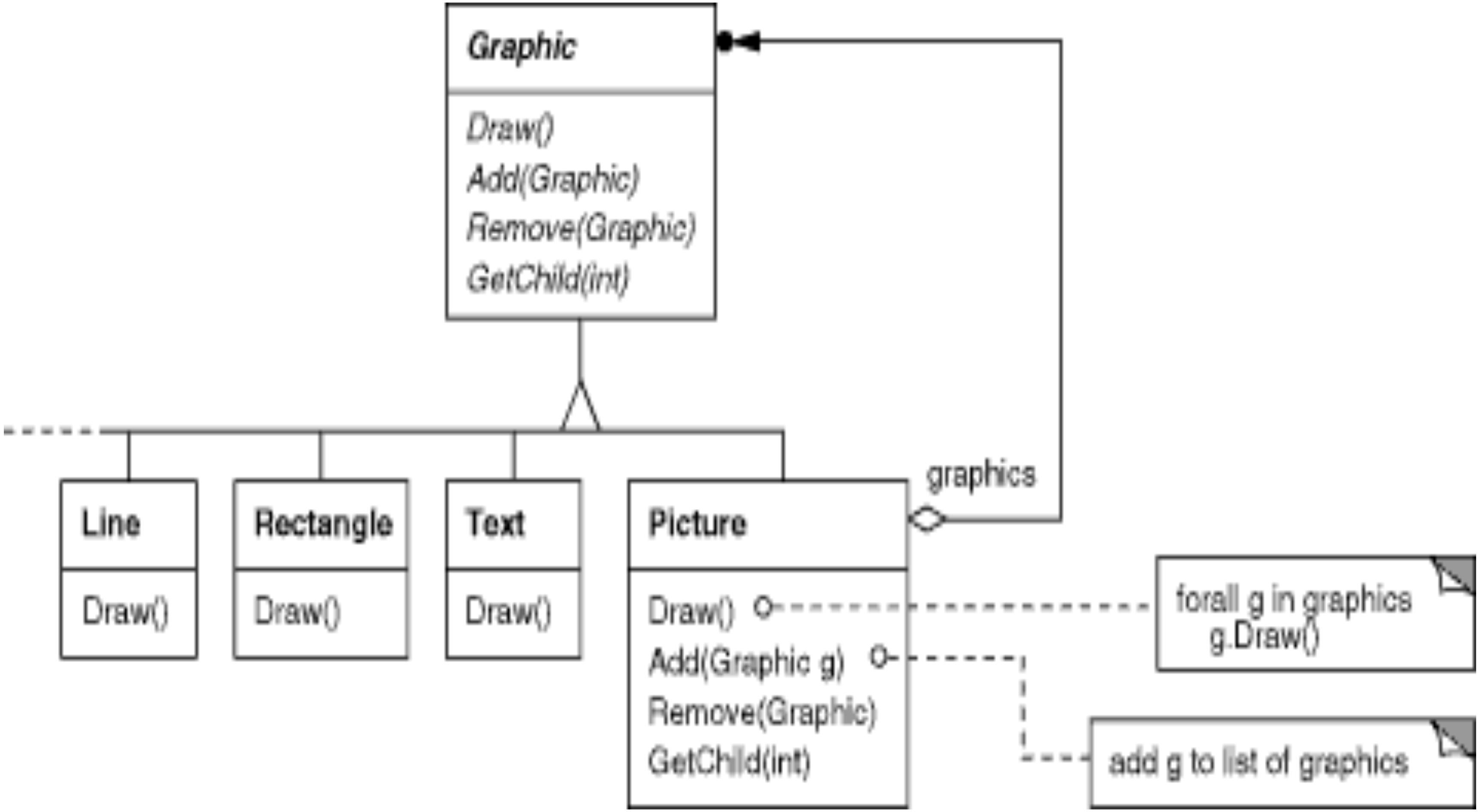
Clients ignore difference between objects compositions and individual objects

Clients treat all objects in the composite structure uniformly

Make it easier to add new kinds of components. Newly defined composite or leaf work automatically with existing clients

Can make the design overly general

Composite: example



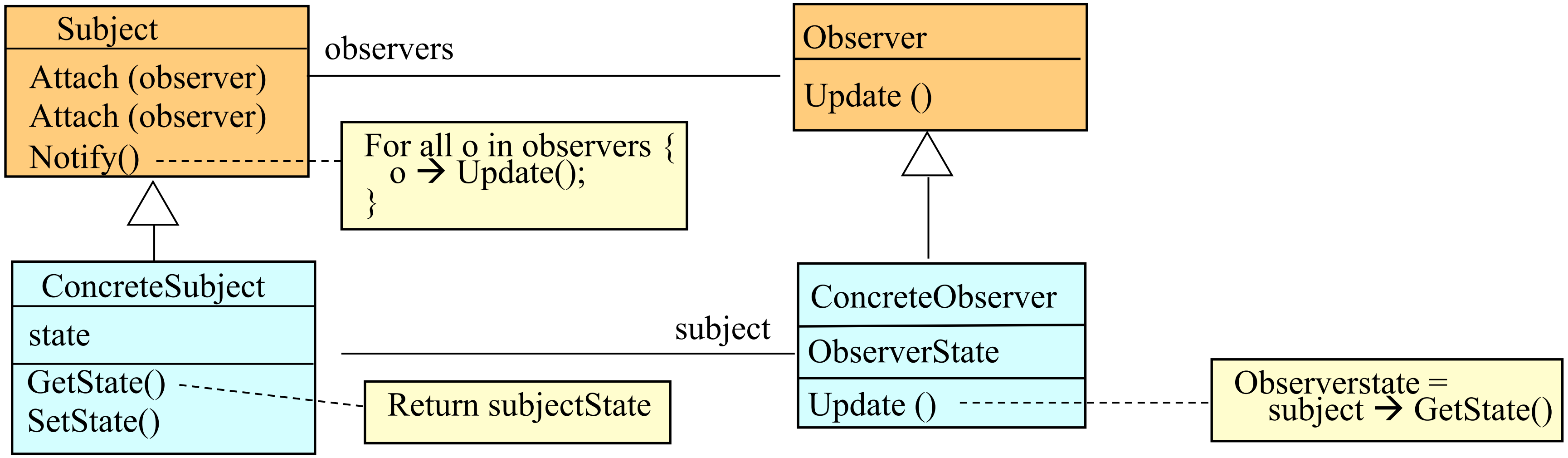
Observer

Name Observer

Problem How to maintain consistency between related objects

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

Solution

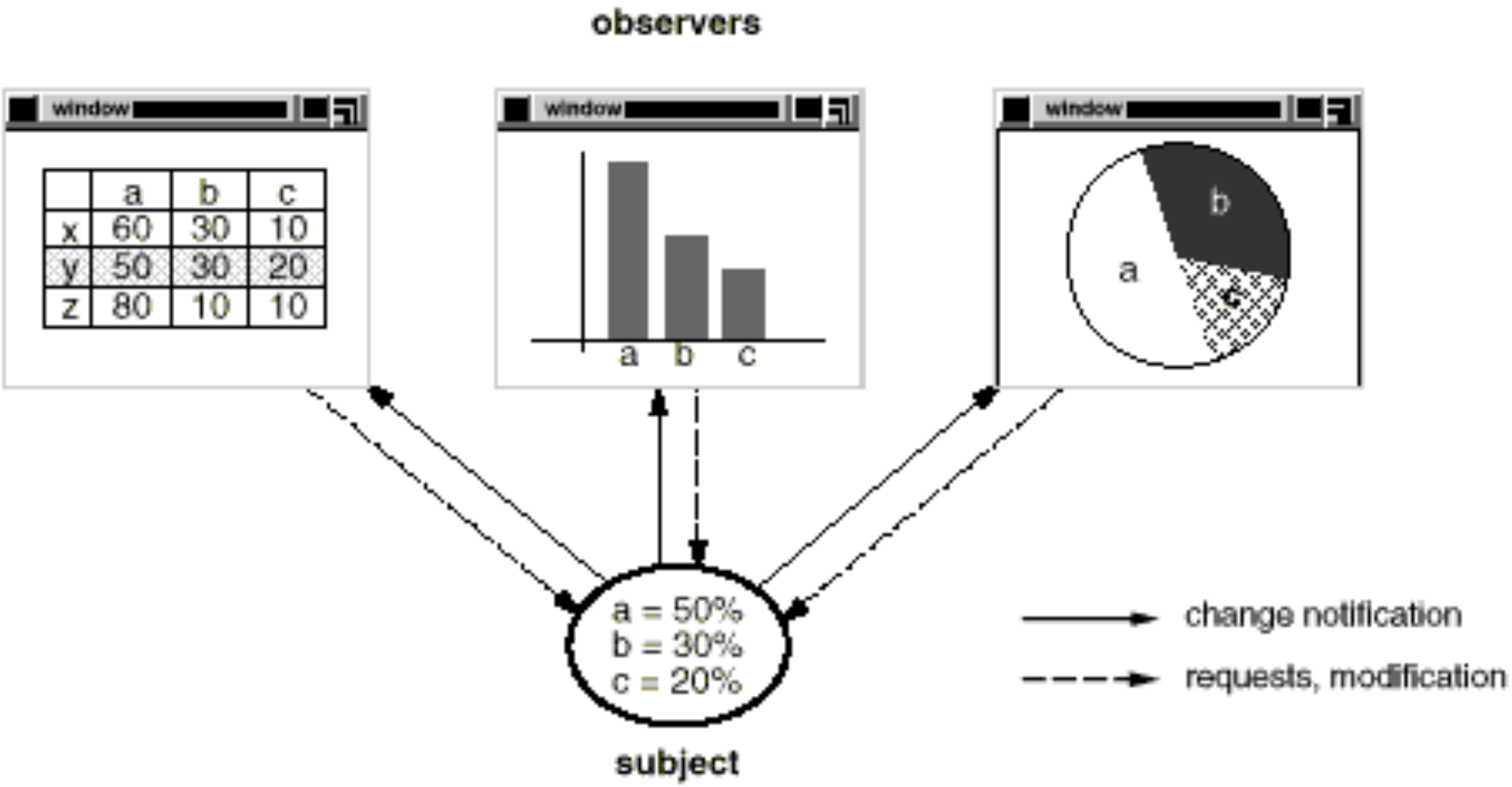


Consequences Abstract coupling between subjects and observers. A subject does not know the concrete class its observers.

Support for broadcast communication. Observers can be removed any tim

Unexpected updates. Beware of cascades of updates!

Observer: example



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10