PART 1. DESIGN PATTERNS

Design Patterns

OO design patterns = *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*.

Each design pattern systematically names, explains, and evaluates an important and recurring design problem in object-oriented systems.

1. A design pattern is a frequently used abstraction of how the software should be built.
2. A design pattern is a schema of a solution of a recurring problem.
3. A design pattern identifies and describes the key aspects of a common design structure that make it useful for creating a reusable object-oriented design: participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

- While architectural patterns and design patterns serve different purposes, they are both valuable tools for solving different types of problems in software engineering. Design patterns are more focused on solving specific coding and design challenges within a single application, whereas architectural patterns address larger-scale concerns that span multiple components, services, or systems. They deal with higher-level system and architectural concerns related to fault tolerance, reliability, and resilience in distributed systems.

Three Types of Design Patterns

Creational patterns

- Creational patterns are used to create objects of the right class for a problem, generally when instances of several different classes are available. They're particularly useful when you're taking advantage of polymorphism and need to choose between different classes at runtime rather than compile time.

Structural patterns

- Structural patterns form larger structures from individual parts, generally of different classes. Structural patterns vary a great deal depending on what sort of structure is being created for what purpose.

Behavioral patterns

- Behavioral patterns describe interactions between objects. They focus on how objects communicate with each other. They can reduce complex flow charts to mere interconnections between objects of various classes. Behavioral patterns are also used to make the algorithm that a class uses simply another parameter that's adjustable at runtime.

Design Patterns Taxonomy (GoF)

Fundamental Design Patterns

Role: the most important design patterns to know. Used extensively in other DPs.

Delegation, Interface, Abstract Superclass, Marker Interface, Proxy

Creational Design Patterns

Role: they provide guidance on how to create objects at runtime. They tell us how to structure and encapsulate these decisions.

Factory Method, Abstract Factory, Builder, Prototype, Singleton, Object Pool

Structural Design Patterns

Role: describe common ways that different types of objects can be organized to work with each other.

Layered Initialization, Filter, Composite, Adapter, Iterator, Bridge, Façade, Flyweight, Dynamic Linkage, Virtual Proxy, Decorator, Cache Management

Behavioral Design Patterns

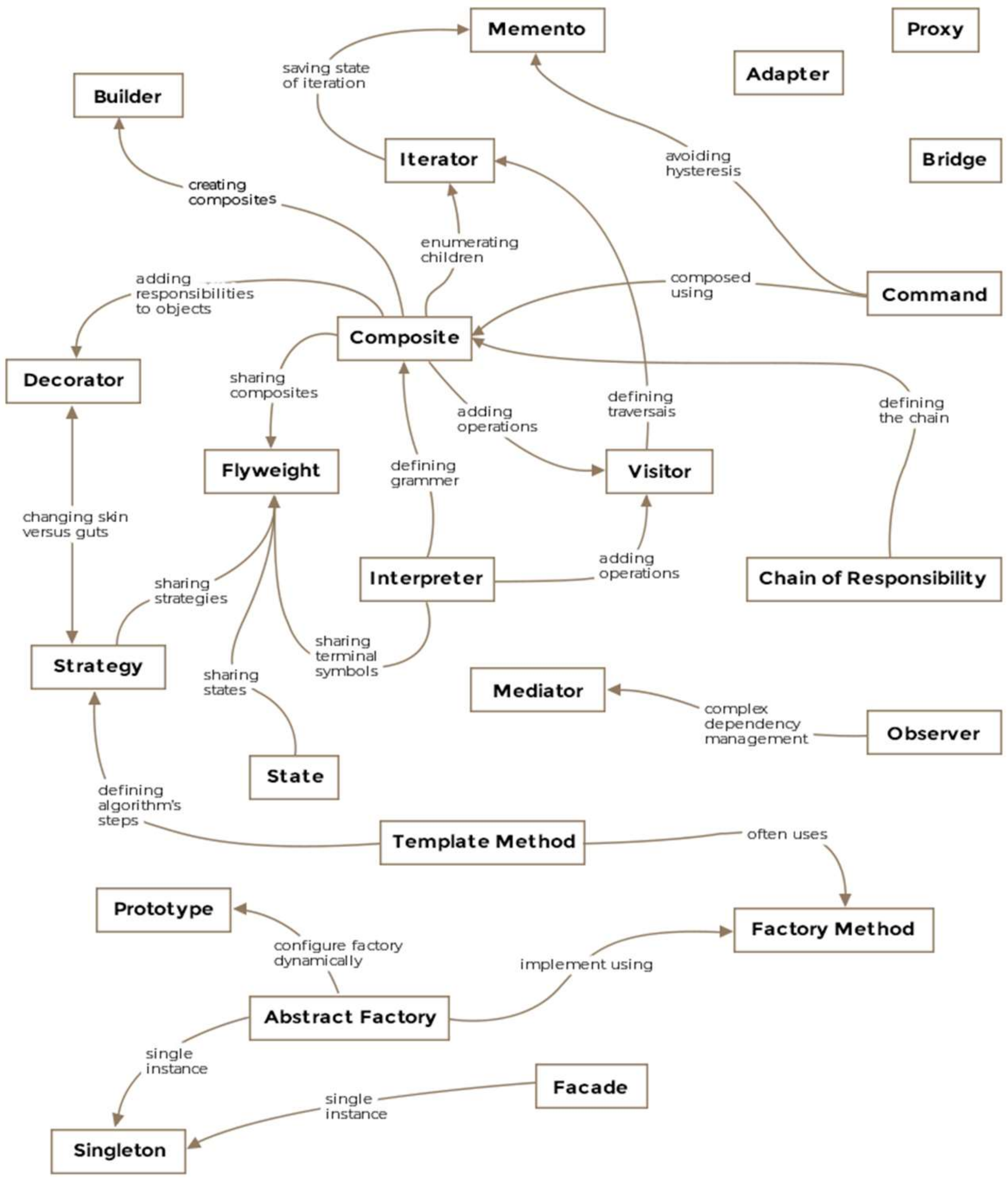
Role: they are used to organize, manage, and combine behavior, i.e. interactions and resposability distribution.

Chain of Responsability, Command, Interpreter, Mediator, Snapshot, Observer, State, Null Object, Strategy, Template Method, Visitor

Design Patterns Space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Vistor

The patterns are inter-related!



Design Pattern Relationships

Design Pattern Complete Description (I)

- **Pattern Name and Classification.** The pattern's name conveys the essence of the pattern succinctly.
- **Intent.** A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
- **Also Known As**. Other well-known names for the pattern, if any.
- **Motivation**. A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
- **Applicability**
  - What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?
- **Structure.** A graphical representation of the pattern using a notation based on UML.
- **Participants.** The classes and/or objects participating in the design pattern and their responsibilities.

Design Pattern Complete Description (II)

- **Collaborations.** How the participants collaborate to carry out their responsibilities.
- **Consequences.** How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?
- **Implementation.** What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- **Sample Code.** Code fragments that illustrate how you might implement the pattern in Java.
- **Known Uses.** Examples of the pattern found in real systems.
- **Related Patterns.** The patterns are inter-related. What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

1.1. Patterns of General Principles in Assigning Responsibilities

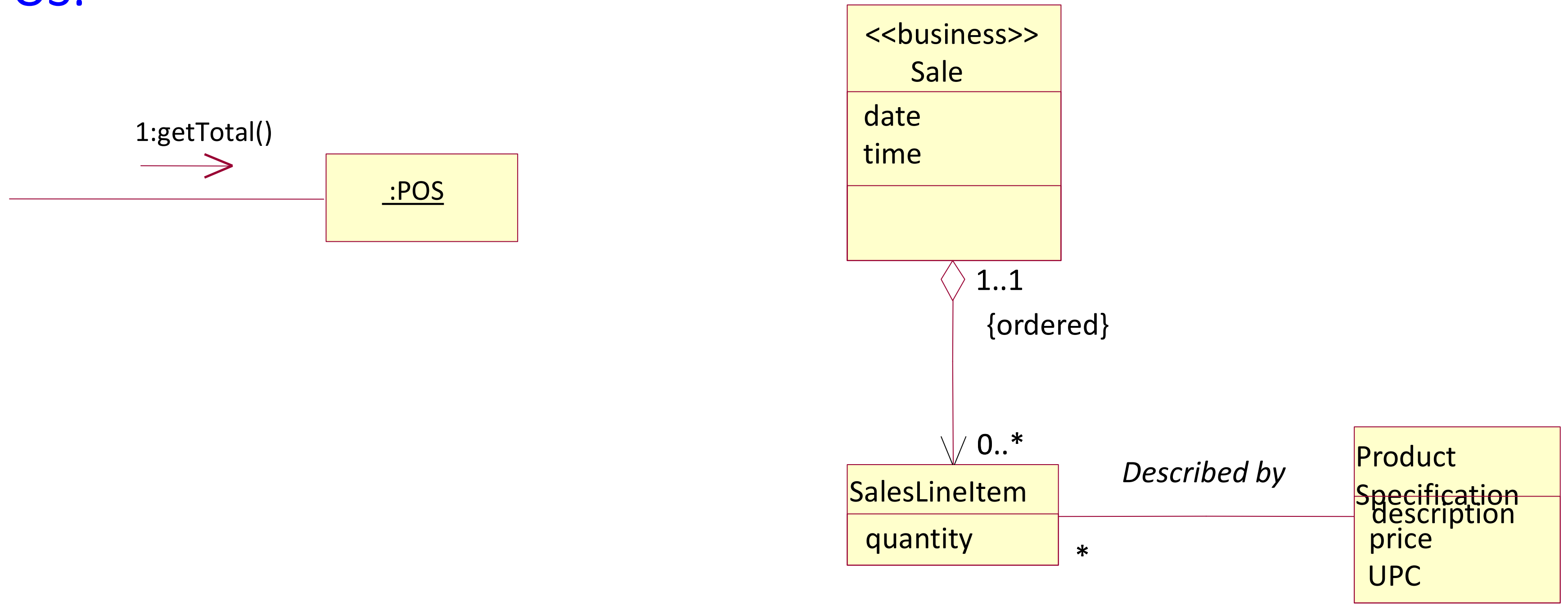
1. Expert
2. Creator
3. Low Coupling
4. High Cohesion
5. Controller

Information Expert

What is a basic principle by which to assign responsibilities to objects?

Assign a responsibility to the information expert - the class that has the *information* necessary to fulfill the responsibility.

POS:



Who should be responsible for knowing the grand total of a sale?

By Information Expert, we should look for that class of objects that has the information needed to determine the total.

Object Creator

Problem:

Who should be responsible for creating a new instance of some class?

Solution:

Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B *aggregates A objects.*
- B *contains A objects.*
- B *records instances of A objects.*
- B *closely uses A objects.*
- B *has the initializing data that will be passed to A when it is created (thus B* is an Expert with respect to creating A).

B is a *creator of A objects.*

If more than one option applies, prefer a class B which *aggregates or contains* class A.

Low Coupling

Couplig in OO: is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements; "too many" is context dependent, but we examine it anyway. These elements include classes, subsystems, systems, and so on.

Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an evaluative principle that you apply while evaluating all design decisions.

Common forms of coupling from TypeX to TypeY include the following:

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- A TypeX object calls on services of a TypeY object.
- TypeX has a method that references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.

Only the classes with low coupling could be easely reused.

High Cohesion

Cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. These elements include classes, subsystems, and so on.

Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly affected by change

Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.

Controller (I)

Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation?

Solution: Assign the responsibility to a class representing one of the following choices:

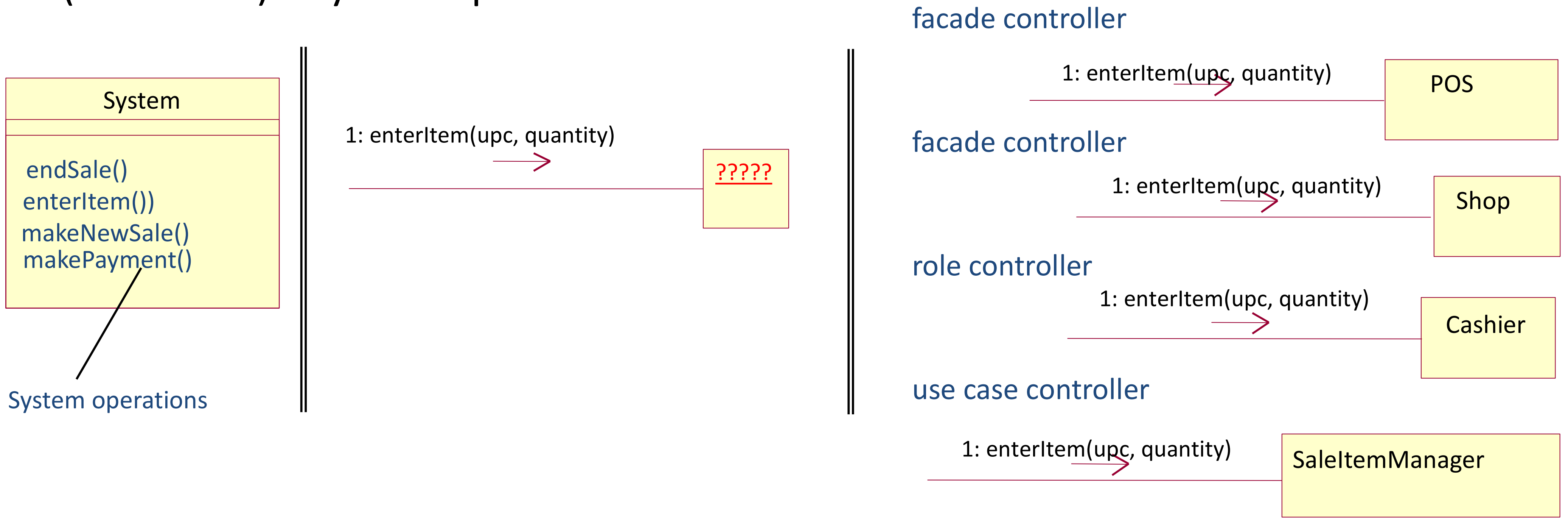
- Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a facade controller).
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session (use case or session controller).
  - Use the same controller class for all system events in the same use case scenario.
  - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).

Corollary: Note that "window," "view," and "document“ should not fulfill the tasks associated with system events; they typically receive these events and delegate them to a controller.

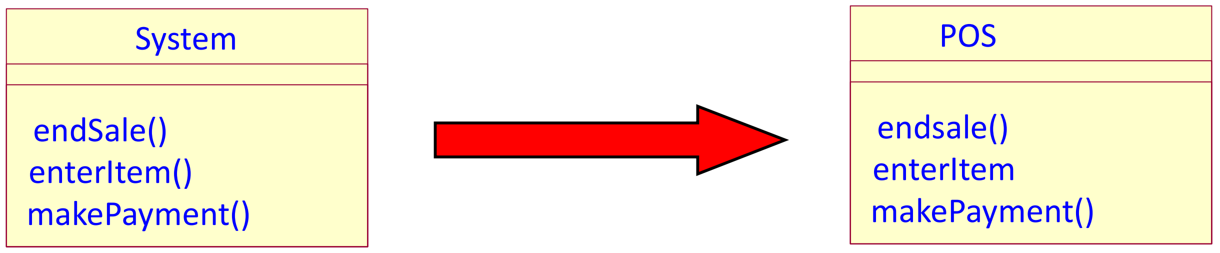
Controller (II)

POS:

Problem: What first object beyond the UI layer receives and coordinates ("controls") a system operation?



Solution:



1.2. Fundamental Design Patterns

1. Delegation
2. Interface
3. Marker Interface
4. Proxy

Class Reuse: Inheritance

- Code reuse = the most important objective for software development.
- **Class reuse** = The use of existing classes for creation of new classes, without modification of the existing classes. Two solutions:
  - Inheritance
  - Composition

Class Inheritance lets you define the implementation of one class in terms of another‘s: with inheritance, the internals of parent classes are often visible to subclasses.

Class inheritance is defined statically at compile-time it is straightforward to use, but has some disadvantages:

- You can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time.
- Parent classes often define at least part of their subclasses' physical representation. Because inheritance exposes a subclass it “breaks encapsulation“ (the superclass and the subclass are tightly coupled and it is difficult to reuse a subclass).

Class reuse: Composition

The new functionality is obtained by assembling or *composing* objects to get more complex functionality.

Object composition requires that the objects being composed have well-defined interfaces. It is defined dynamically at run-time through objects acquiring references to other objects.

- Because objects are accessed solely through their interfaces encapsulation is not broken: any object can be replaced at run-time by another as long as it has the same type.
- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small.

Composition has some disadvantages:

- a design based on object composition will have more objects (if fewer classes),
- the system's behavior will depend on their interrelationships instead of being defined in one class.

When the inheritance should be used?

Principle: Favor object composition over class inheritance.

Use the inheritance :

1. When the relationship between the two classes is “x is a type of y” and never when is “x is a role played by y”.
2. When during the program execution you do not need to change the belonging of an object to a class.
3. When you need to model types of roles, transactions or devices in the problem domain and never for behaviors.

Do not use the inheritance:

1. For overriding or canceling the members of the superclass.
2. For a utility class (i.e. for a functionality to be used in many ways or in many places in the program).

Delegation

Intent: Delegation extends and reuses the functionality of a class in an other class with more functionality than the first one. In delegation, *two* objects are involved in handling a request: a receiving object, delegator, delegates operations to its *delegate*. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the this member variable. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.

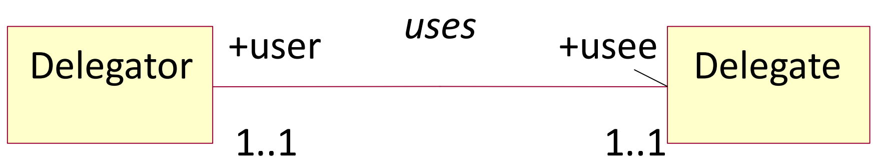
Motivation: is a way of making composition as powerful for reuse as inheritance.

Applicability: when we need to extend an existing class, but we can not use the inheritance because:

– an object belonging to a subclass should become an object of another subclass of the same superclass

– when a class should hide a member inherited from its superclass.

Solution: Let introduce an other class (Delegator) which uses an instance of the uses an instance of the class to be used (Delegate).



Inheritance vs. Delegation

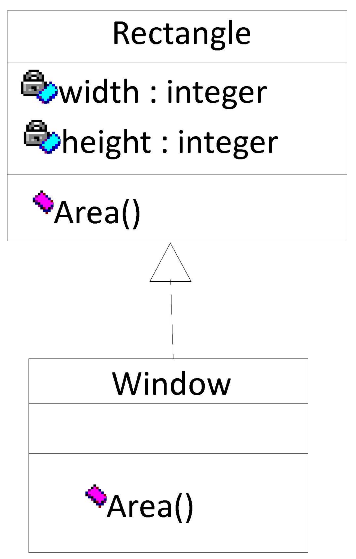
Inheritance

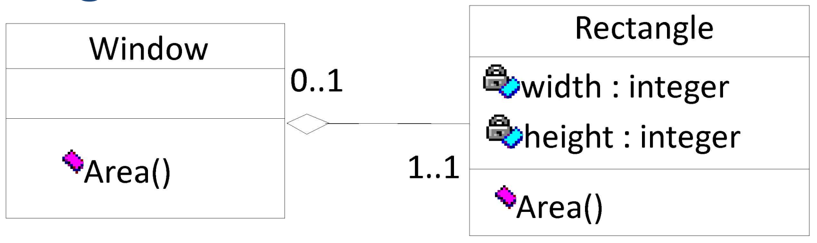
1. Inheritance can not be modified during execution.
2. Often inheritance enters in conflict with incapsulation because allows to a subclass to access the superclass details. Each change in the superclass involves changes of its subclasses. Subclass reuse becomes difficult. In this case is better to inherit from an abstract class.

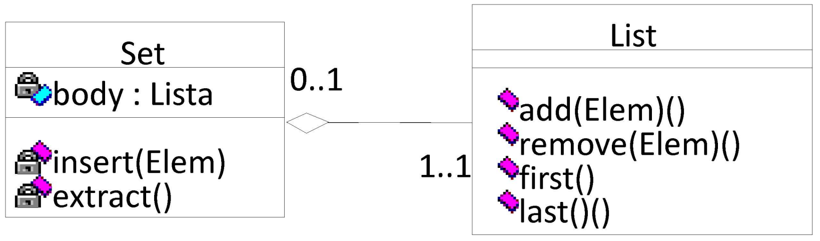
Composition (Aggregation) + delegation

1. Avoid inheritance disadvantages.
2. Allows to build behaviors at runtime.
3. Is less structured than inheritance. It is a technique for describing the relationship between two classes.

Inheritance Delegation

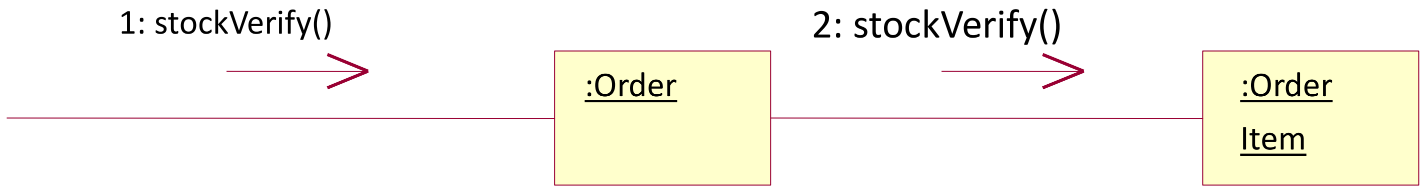






Delegation: Implementation

Example:
Java event model is based on delegation: the event sources do not decide themselves what to do when an event occurs, but delegate the event processing to listeners.



```
class Order {
. . . .
private Vector orderItems = new Vector();
. . . .
void stockVerify() throws StockException {
. . . .
   orderItems.elementAt(i).stockVerify();
. . . .
}
public class StockException extends Exception {
   public StockException(String msg) {
        super(msg);
   }
}
```

```
class OrderItem {
. . . .
private Product product;
private Quantity quantity;
. . . .
synchronized void stockVerify()
        throws StockException {
. . . .
   if (product.getStock() - quantity <
       Stock.minStock(product))
          throw new StockException
               ("Under the minimum stock");
. . . .
}
```

Related Patterns: any model which uses delegation. Particularly, Proxy and Decorator.

Interface (I)

Intent: Instances of a class provide data and services to instances of other classes. You want to keep client classes independent of specific data-and-service-providing classes so you can substitute another data-and-service-providing class with minimal impact on client classes. You accomplish this by having other classes access the data and services through an interface.

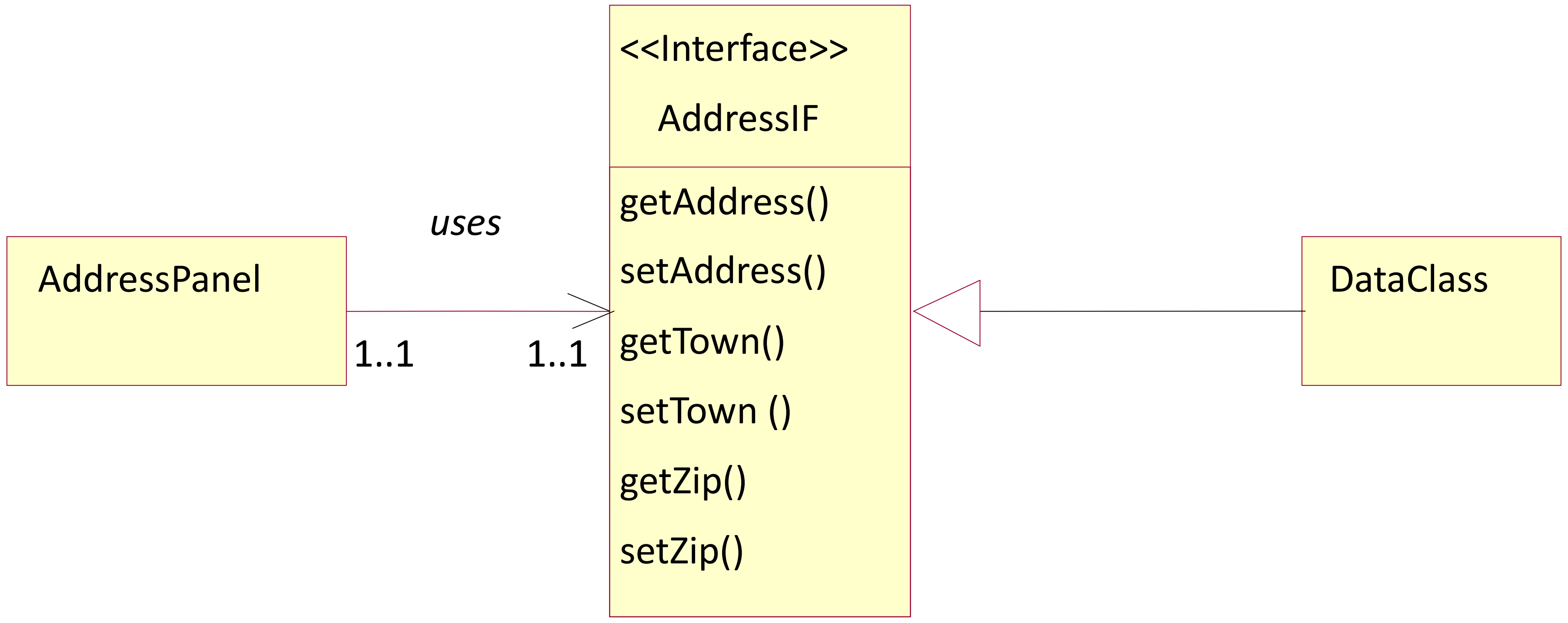
Motivation: An application that manages the purchase of goods for a business. Among the entities your program work with are vendors, freight companies, receiving locations, and billing locations. One common aspect of these entities is that they all have street addresses which appear in different parts of the user graphical interface.

You want to have a class for displaying and editing street addresses so that you can reuse it wherever there is a street address in the user interface. We call this class AddressPanel. AddressPanel objects should get and set address information in a separate data object that may represent a vendor, a freight company, and so on.

Has an AddressPanel to be coupled with all these objects?

Interface (II)

You can solve the problem by creating an address interface. Instances of the AddressPanel class would then simply require the data objects that they work with to implement the address interface. They would call the accessor methods of the interface to get and set the object’s address information.



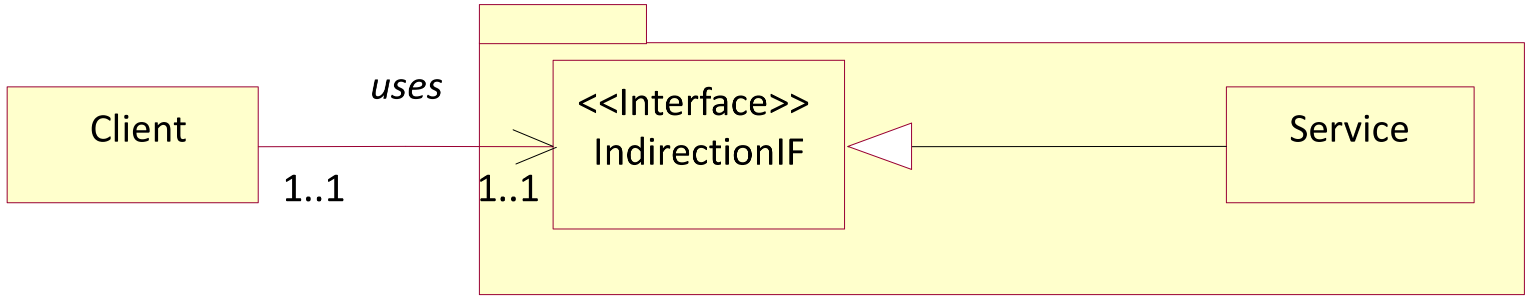
Interface (III)

Applicability:

1. An object relies on another object for data or services. If the object must assume that the other object upon which it relies belongs to a particular class, the reusability of the object's class would be compromised.
2. You want to vary the kind of object used by other objects for a particular purpose without making the object dependent on any class other than its own.
3. A class's constructors cannot be accessed through an interface, because Java's interfaces cannot have constructors.

Solution:

To avoid classes having to depend on other classes because of a uses/used-by relationship, make the usage indirect through an interface.
Client uses classes that implement the IndirectionIF interface.
IndirectionIF provides indirection that keeps the Client class independent of the class that is playing the Service role. Interfaces in this role are generally public.
Service. Classes in this role provide a service to classes in the Client role. Classes in this role are ideally private to their package.



Interface (IV)

Consequences:

1. Applying the Interface pattern keeps a class that needs a service from another class from being coupled to any specific class.
2. Like any other indirection, the Interface pattern can make a program more difficult to understand.

Implementation:

Implementation of the Interface pattern is straightforward:

1. define an interface to provide a service,
2. write client classes to access the service through the interface, and
3. write service-providing classes that implement the interface.

Java interfaces cannot have constructors. For this reason, interfaces are not helpful in keeping a class responsible for creating objects independent of the class of objects that it creates.

Interface (IV)

```
import java.awt.*;
import java.awt.event.*;
class AddressPanel extends Panel {
  private addressIF data;
  TextField addressField = new TextField("", 35);
  TextField townField  = new TextField("", 16);
  TextField ZIPField = new TextField("", 10);
  public AddressPanel() {
     .......
  }
  public void setData(AddressIF address) {
    data = address;
    addressField.setText(address.getAddress());
    townField.setText(address.getTown());
    ZIPField.setText(address.getZIP());
  }
  public void save() {
    if (data != null) {
      data.setAddress(addressField.getText());
      data.setTown(townField.getText());
      data.setZIP(ZIPField.getText());
    }
  }
}
public interface AddressIF {
    public String getAddress();
    public void setAddress(String address1);
    public String getTown();
    public void setTown(String town);
    public String getZIP() ;
    public void setZIP(String ZIP);
}
```

```
class Address implements AddressIF{
  private String address;
  private String town;
  private String ZIP;
    //...
  public String getAddress(){return address; }
  public void setAddress(String address) {
      this.address = address; }
  public String getTown() { return town; }
  public void setTown(String town) {
      this.town = town; }
  public String getZIP() { return ZIP; }
  public void setZIP(String ZIP) {
        this.ZIP = ZIP;
  }
}
```

```
import java.awt.*;
public class AddressTest extends Frame {
  public static void main(String[] argv) {
        new TestAddress().show();
  }
AddressTest () {
   super("Test the AddressPanel");
   add(new
    AddressPanel(),BorderLayout.CENTER);
   pack();
   setWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent evt)
         {}
   } );
  }
}
```

Marker Interface (I)

Intent:

The Marker Interface pattern uses the fact that a class implements an interface to indicate semantic Boolean attributes of the class. It works particularly well with utility classes that must determine something about objects without assuming that they are an instance of any particular class.

Motivation:

Java's Object class defines a method called `equals`. The argument to `equals` can be a reference to any object. All Java classes inherit the `equals` method from the Object class. The implementation of `equals` provided by the Object class is equivalent to the `==` operator. However, classes that want their instances to be considered equal if they contain the same values override the `equals` method appropriately.

Container objects, such as `java.util.ArrayList`, call an object's `equals` method when performing a search of their contents to find an object that is equal to a given object. Such searches might call an object's `equals` method for each object in the container objects. This is wasteful in those cases where the object being searched for belongs to a class that does not override the `equals` method, because it is faster to use the `==` operator to determine whether two objects are the same object. If the container class were able to determine that the object being searched for belongs to a class that does not override the equals method, then it could use the `==` operator instead of calling `equals`.

The solution is to introduce an interface without member methods called `EqualsByIdentity` which is implemented by classes which do not override `equals()`, thus it is possible to use `==`. The interface provides a marking of the classes.

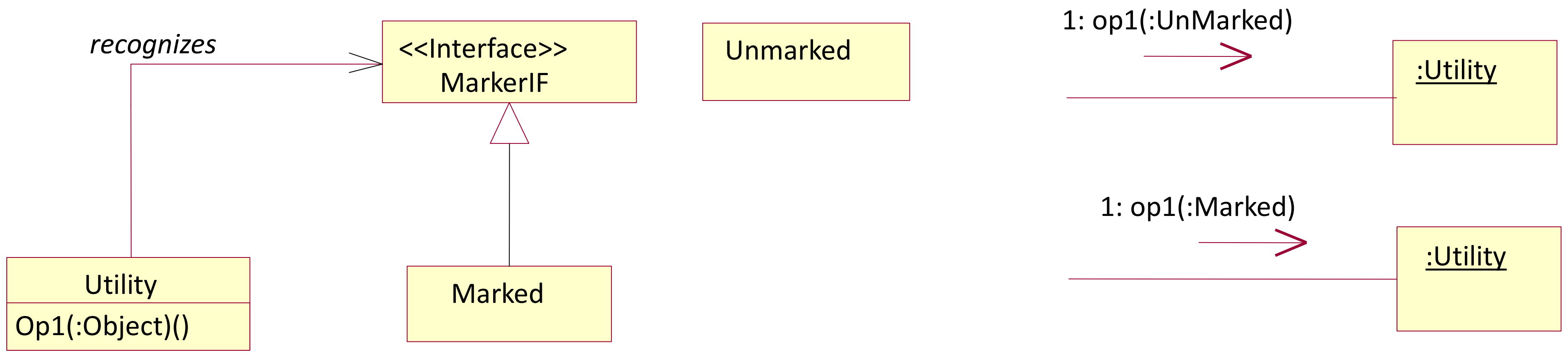
Marker Interface (II)

Applicability:

1. Utility classes may need to know something about the intended use of an object’s class that is either true or false without relying on objects being an instance of a particular class.
2. Classes can implement any number of interfaces.
3. It is possible to determine whether an object’s class implements a known interface without relying on the object being an instance of any particular class.
4. Some attributes about the intended use of a class may change during the class’s lifetime.

Solution:

For instances of a utility class to determine whether another class’s instances are included in a classification without the utility class having knowledge of other classes, the utility class can determine whether other classes implement a marker interface. A marker interface is an interface that does not declare any methods or variables.



Marker Interface (III)

Consequences:

Instances of utility classes are able to make inferences about objects passed to their methods without depending on the objects to be instances of any particular class. The relationship between the utility class and the marker interface is transparent to all other classes except those that implement the interface.

Implementation:

The formal parameter that corresponds to the object is typically declared as Object. It is also possible to use an interface that declares methods in the Marker Interface pattern. In such cases, the interface used as a marker interface usually extends a pure marker interface.

Declaring that a class implements a marker interface implies that the class is included in the classification implied by the interface. It also implies that all subclasses of that class are included in the classification.

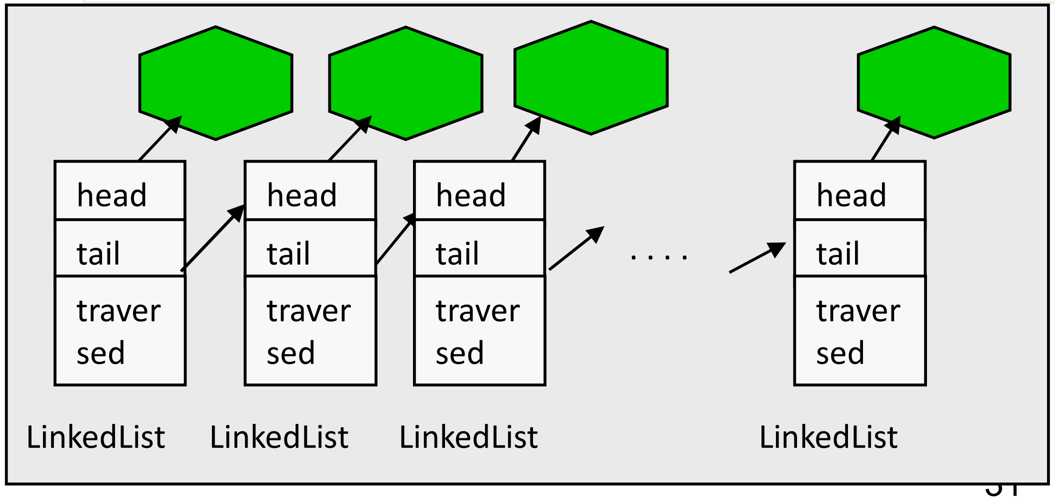
Java exemple:

The Serializable interface is a Marker Interface. Instances of the ObjectOutputStream write as a stream of bytes only objects that implements Serializable. The conversion of an object to a stream of bytes is called *serialization.* An instance of the ObjectInputStream class can read the stream of bytes and turn it back into an object.

Marker Interface (IV)

```
import java.util.Enumeration;
import java.util.NoSuchElementException;
public class LinkedList implements Cloneable, java.io.Serializable {
    private Object head;
    private LinkedList tail;
    private boolean traversed = false;
    public LinkedList() { this(null, null); }
    public LinkedList(Object head) { this(head, null); }
    public LinkedList(Object head, LinkedList tail) { this.head = head; this.tail = tail; }
    public Object getHead() { return head; }
    public LinkedList getTail() { return tail; }
    synchronized public int size() {
        if (tail == null) return 1;
        try {
            traversed = true;
            if (tail.traversed) return 1;
            return 1 + tail.size();
        } finally { traversed = false; }
    }
    public Enumeration elements() { return new ListEnumeration(); }
    private class ListEnumeration implements Enumeration {
        private LinkedList thisNode = LinkedList.this;
        public boolean hasMoreElements() { return thisNode != null;}
        public Object nextElement() {
          if (thisNode == null) throw new NoSuchElementException();
          Object next = thisNode.head;
          thisNode = thisNode.tail;
          return next;
        }
    }
    public LinkedList find(Object target) {
        if (target == null || target instanceof EqualByIdentity) return findEq(target);
        else return findEquals(target);
    }
    private synchronized LinkedList findEq(Object target) {
        if (head == target) return this;
        if (tail == null) return null;
        try {
            traversed = true;
            if (tail.traversed) return null;
            return tail.findEq(target);
        } finally { traversed = false; }
    }
    private synchronized LinkedList findEquals(Object target) {
        if (head.equals(target)) return this;
        if (tail == null) return null;
        try {
            traversed = true;
            if (tail.traversed) return null;
            return tail.findEquals(target);
        } finally { traversed = false; }
    }
}
public interface EqualByIdentity { }
```

The LinkedList class implements a linked-list data structure. The purpose of the methods find, findEq, and findEquals is to find a LinkedList node that refers to a specified object. The find method is the only one of the three that is public. The findEq method performs the necessary equality tests by using the == operator and the findEquals method performs the necessary equality tests by using the equals method of the object being searched for. The find method decides whether to call the findEq method or the findEquals method by determining whether the object to search for implements the marker interface EqualByIdentity.



Proxy (I)

Intent:

Provide a surrogate or placeholder for another object to control access to it. Proxy is a very general pattern that occurs in many other patterns but never by itself in its pure form. The Proxy pattern forces method calls to an object to occur indirectly through a proxy object that acts as a surrogate for the other object, delegating method calls to that object. Classes for proxy objects are declared in a way that usually eliminates the client object's awareness that it is dealing with a proxy.

Motivation:

The proxy object's methods do not directly provide the service that its clients expect; instead, they call the methods of the object that provides the actual service. Although a proxy object's methods do not directly provide the service its clients expect, the proxy object provides some management of those services. Proxy objects share a common interface with the service-providing object. Whether client objects directly access a service-providing object or a proxy object, they access it through the common interface rather than an instance of a particular class. Doing so allows client objects to be unaware that they call the methods of a proxy object rather than the methods of the actual service-providing object. Transparent management of another object's services is the basic reason for using a proxy object.

Proxy versions

- Remote Proxy – Represents an object locally which belongs to a different address space. Think of an ATM implementation, it will hold proxy objects for bank information that exists in the remote server. RMI is an example of proxy implmenetation for this type in java.
- Virtual Proxy – In place of a complex or heavy object use a skeleton representation. It creates the illusion that a service object exists before it actually does. Doing so can be useful if a service object is expensive to create and its services may not be needed. When an underlying image is huge in size, just represent it using a virtual proxy object and on demand load the real object. You feel that the real object is expensive in terms of instantiation and so without the real need we are not going to use the real object. Until the need arises we will use the virtual proxy.
- Protection Proxy – controls access to a service-providing object based on a security policy. Working on a MNC we are well aware of the proxy server that provides us internet, in fact it censures internet: it provides us only work related web pages. If we search for something critical in Google and click the result we get this page is blocked by proxy server. You never know why this page is blocked and you feel this is genuine.
- Smart Reference – is a substitute of a simple pointer for accessing an object. It adds new operations as: counts accesses, loads a persistent object in memory when it is referred for the first time or verifies if a real object is blocked when is accessed for the first time.

Proxy Applicability

The Proxy pattern may be used when:

- It is not possible for a service-providing object to provide a service at a convenient time or place.
- Gaining visibility to an object is complex and you want to hide that complexity.
- Access to a service-providing object must be controlled without adding complexity to the service-providing object or coupling the service to the access control policy.
- The management of a service should be provided in a way that is transparent to the clients of that service.
- The clients of a service-providing object do not care about the identity of the object's class or which instance of its class they are working with.

Proxy - Solution

Solution

ServiceProxy

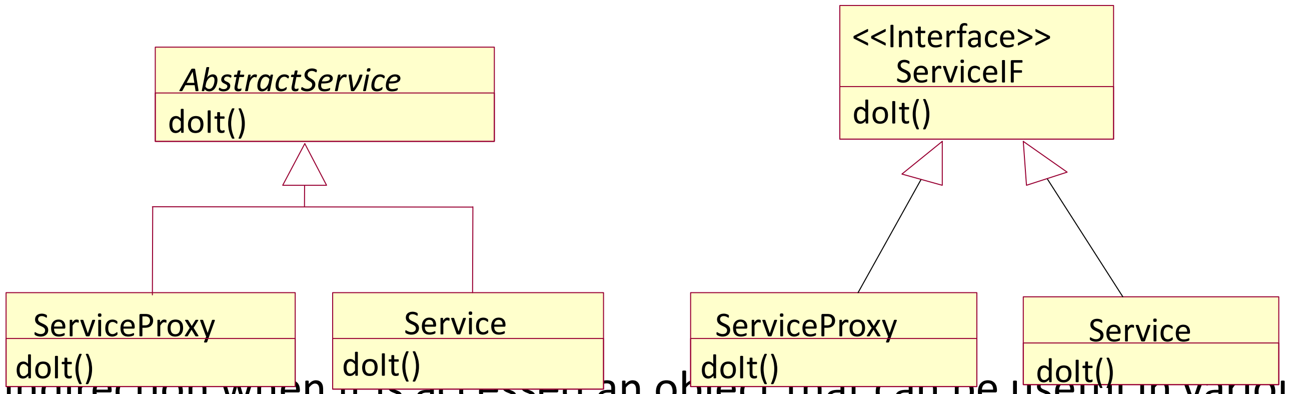
- keeps a reference to **Service.**
- Provides an interface identical to that of the object and thus can replace AbstractService **Service**.
- Controls access to **Service** and may be responsible for its creation and destruction.
  - *remote*: encodes a request and its arguments and sends it to **Service**,
  - *virtual*: can store in a cache information about **Service** and therefore should not be accessed the same object every time a request is received,
  - *protection*: check if the caller has the right to make a request.

AbstractService sau ServiceIF defines a common interface for **ServiceProxy** and **Service**. As a consequence **ServiceProxy** may be used anywhere **Service** is valid.

Service defines the real object that is represented by **ServiceProxy**.

Colaborations:

ServiceProxy sends a request to **Service** when needed, according to the type of proxy.



Conclusions:

- Introduces a level of indirection when it is accessed an object that can be useful in various cases.

Implementation

To be implemented, the model requires only creating a class that share an interface or a superclass of the class that provides the service and delegates operations to the class instance that provides the service.

1.3. Creational Patterns

1. Factory Method (class)
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton
6. Object Pool

- These patterns are often used in place of direct instantiation with constructors (object creation without *new*!).
- They make the creation process more adaptable and dynamic.
- In particular, they can provide a great deal of flexibility to specify what objects are to be created, how those objects are created, and how they are initialized.

Creational Patterns

- Creational patterns involve the construction of new objects. However, they rarely use constructors directly. Rather, a creational pattern often hides the constructors in the classes being created, and provides alternate methods to return instances of the desired class.
- The most common reason for using a creational pattern is that you need to vary the class that's instantiated in order to fit the concrete situation. Clearly, a constructor in a single class is an inadequate method to return instances of possibly different classes. Almost always the objects returned are instances of some common super-class.
- *Class creational patterns* use inheritance to vary the object being created. *Object creational patterns* generally delegate the actual construction to a different object that is responsible for deciding which class is required and invoking the necessary constructor.

Example

```
public URLConnection openConnection() throws IOException
```

- URLConnection is an *abstract class.* Concrete subclasses represent connections to many different kinds of servers including HTTP, FTP, SMTP, NNTP, and more. Each of these concrete subclasses has to speak a different protocol and behave a little differently.

Abstract Factory (I)

Known also as Kit or Toolkit.

Intent

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

2. Given a set of related interfaces, provide a way to create objects that implement those interfaces from a matched set of concrete classes.

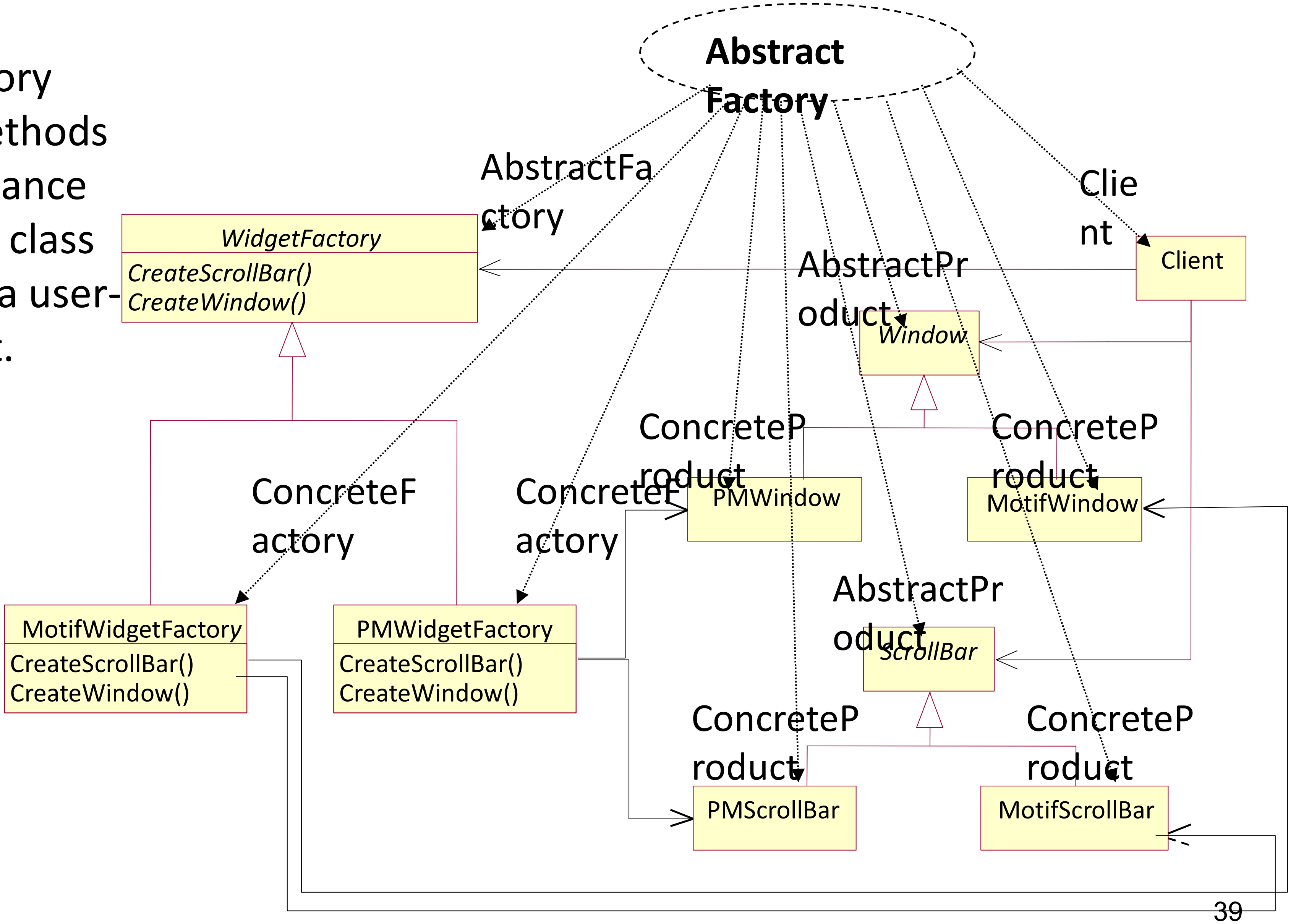
Motivation

Build a user-interface framework that works on top of multiple windowing systems, such as Windows, Motif, or MacOS with the platform's native look and feel. You organize it by creating an abstract class for each type of widget (text field, pushbutton, list box, etc.) and then writing a concrete subclass of each of those classes for each supported platform.

Abstract Factory (II)

To make this structure robust, you need to ensure that all the widget objects created are for the desired platform.

An abstract factory class defines methods to create an instance of each abstract class that represents a user-interface widget.



Abstract Factory (III)

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)

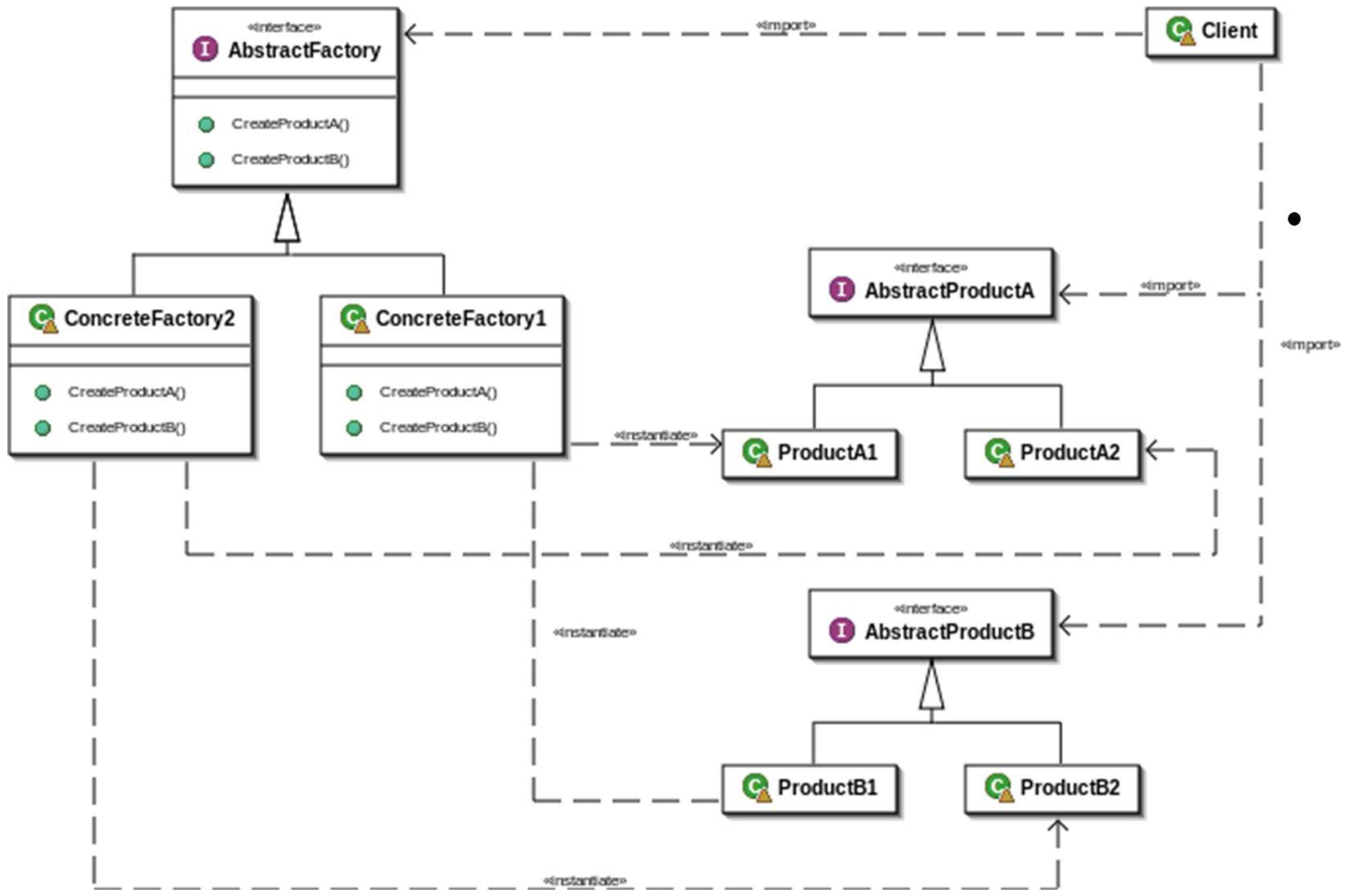
- defines a product object to be created by the corresponding concrete factory.
- implements the AbstractProduct interface.

Client uses only interfaces declared by AbstractFactory and AbstractProduct classes.

In a more general context, an abstract factory class and its concrete subclasses organize sets of concrete classes that work with different but related products. For a broader perspective, consider another situation.

Abstract Factory (IV)

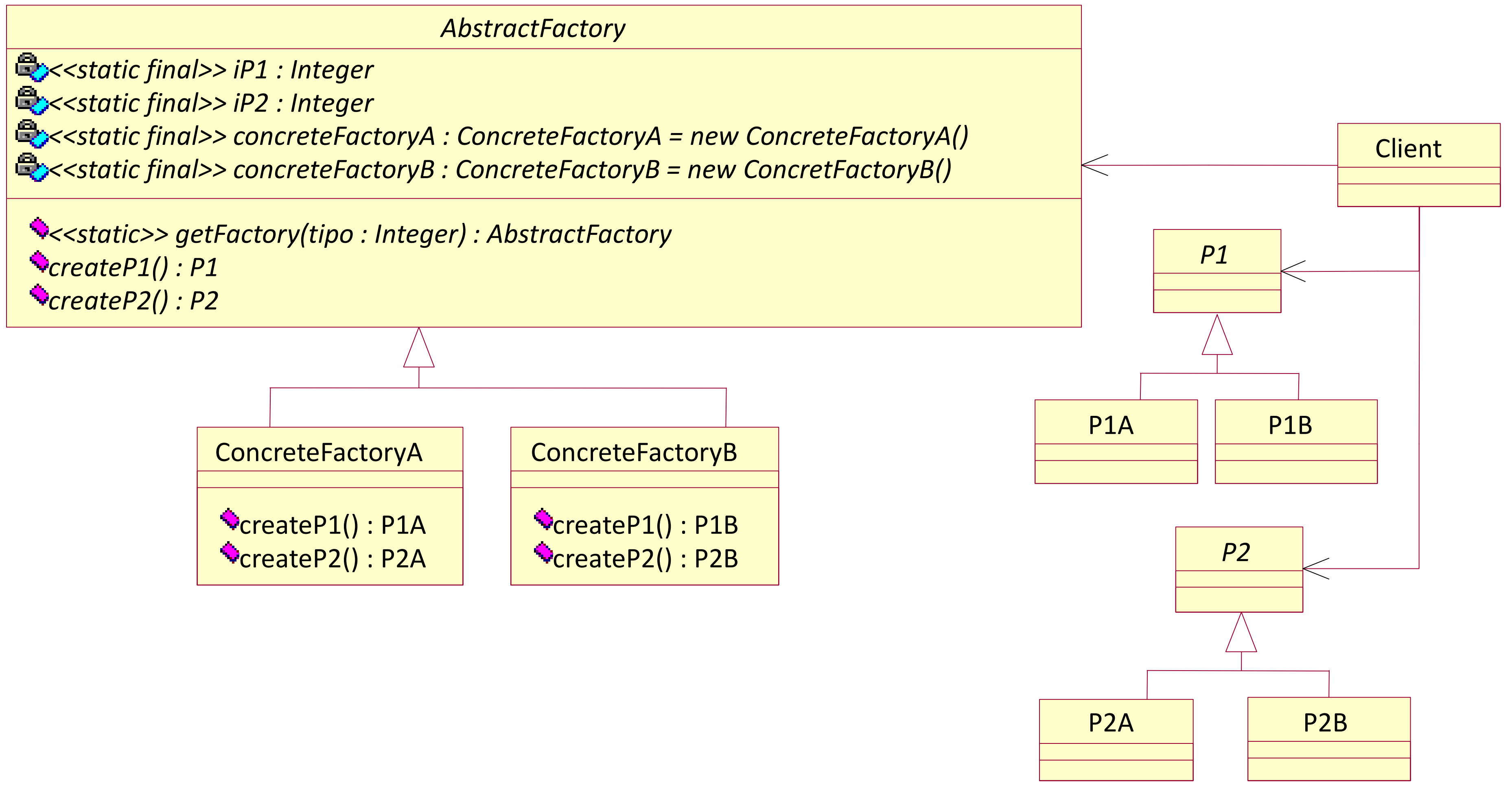
The **abstract factory pattern** provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.



- The *factory* determines the actual *concrete* type of object to be created, and it is here that the object is actually created . However, the factory only returns a reference to the created concrete object.
- This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return a reference to the object.

Abstract Factory (V)

A broader perspective:



AbstractFactory
<<static final>> iP1 : Integer
<<static final>> iP2 : Integer
<<static final>> concreteFactoryA : ConcreteFactoryA = new ConcreteFactoryA()
<<static final>> concreteFactoryB : ConcreteFactoryB = new ConcretFactoryB()
<<static>> getFactory(tipo : Integer) : AbstractFactory
createP1() : P1
createP2() : P2
ConcreteFactoryA
createP1() : P1A
createP2() : P2A
ConcreteFactoryB
createP1() : P1B
createP2() : P2B
Client
P1
P1A
P1B
P2
P2A
P2B

Abstract Factory (VI)

A simple example:

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.

Abstract Factory (VII)

Implementation

Useful techniques for implementing the Abstract Factory pattern:

Factories as singletons. An application typically needs only one instance of a `ConcreteFactory` per product family.

Creating the products. `AbstractFactory` only declares an *interface* for creating products. It's up to `ConcreteProduct` subclasses to actually create them. The most common way to do this is to define a factory method for each product. A concrete factory will specify its products by overriding the factory method for each. While this implementation is simple, it requires a new concrete factory subclass for each product family, even if the product families differ only slightly.

Defining extensible factories. `AbstractFactory` usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the `AbstractFactory` interface and all the classes that depend on it.

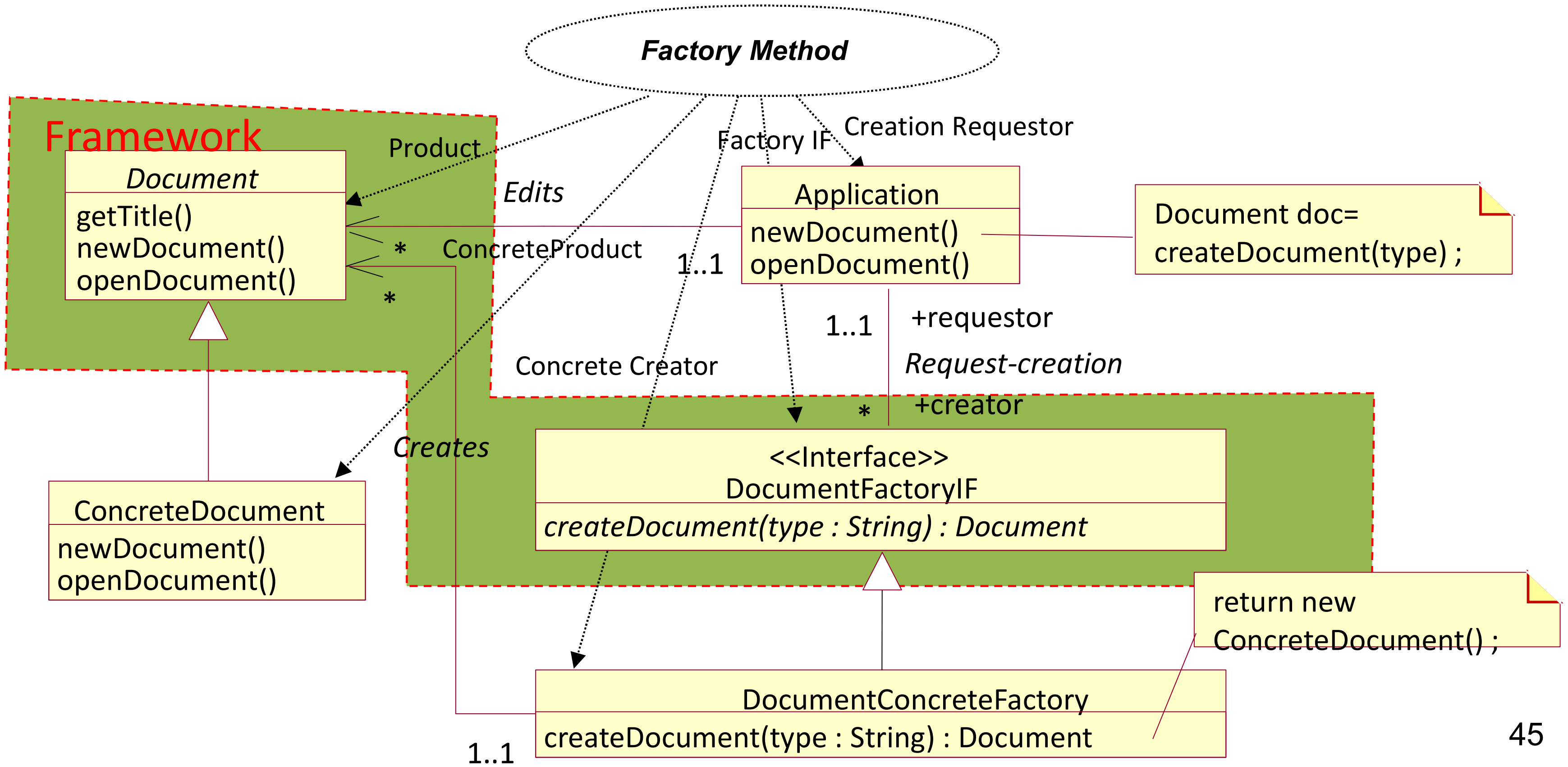
Factory Method (I)

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:

Consider the problem of writing a framework for desktop applications. Such applications are typically organized around documents or files. Their operation usually begins with a command to create or edit a word processing document, spreadsheet, time line, or other type of document the application is intended to work with.



Factory Method (cont.d)

A framework to support this type of application will include high-level support for common operations such as creating, opening, or saving documents. Such support will generally include a consistent set of methods to call when the user issues a command. For the purpose of this discussion, we will call the class providing the methods of the Application class.

Because the logic to implement most of these commands varies with the type of document, the Application class usually delegates most commands to some sort of document object. The logic in document objects for implementing these commands varies with the type of document. However, some operations, such as displaying the title of a document, will be common to all document objects. This suggests an organization that includes:

- An application-independent document interface
- An abstract class that provides application-independency.

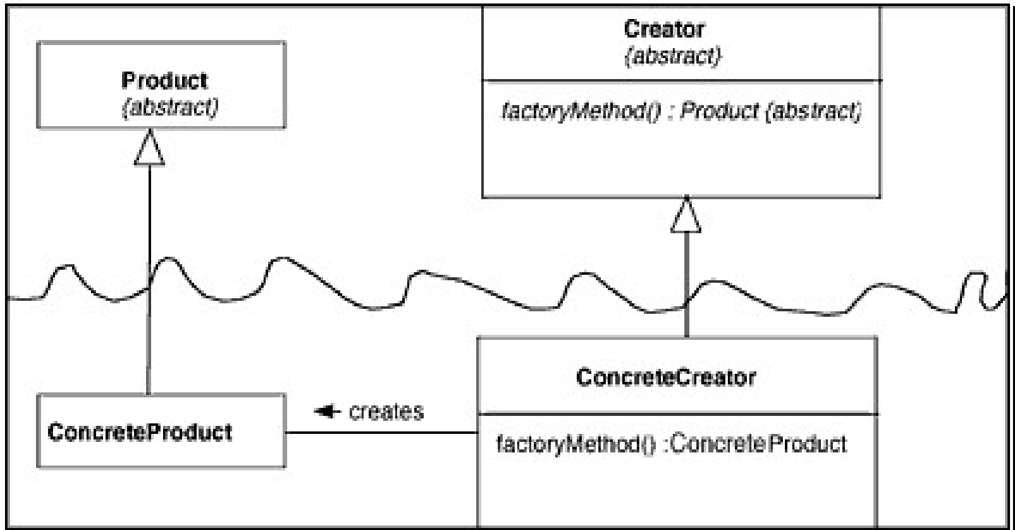
One way to accomplish this is for the programmer using the framework to provide a class that encapsulates logic for selecting and instantiating application-specific classes. For the Application class to be able to call the programmer-provided class without having any dependencies on it, the framework should provide an interface that the programmer-provided class must implement. Such an interface would declare a method that the programmer-provided class would implement to select and instantiate a class. The Application class works through the framework-provided interface and not with the programmer-provided class.

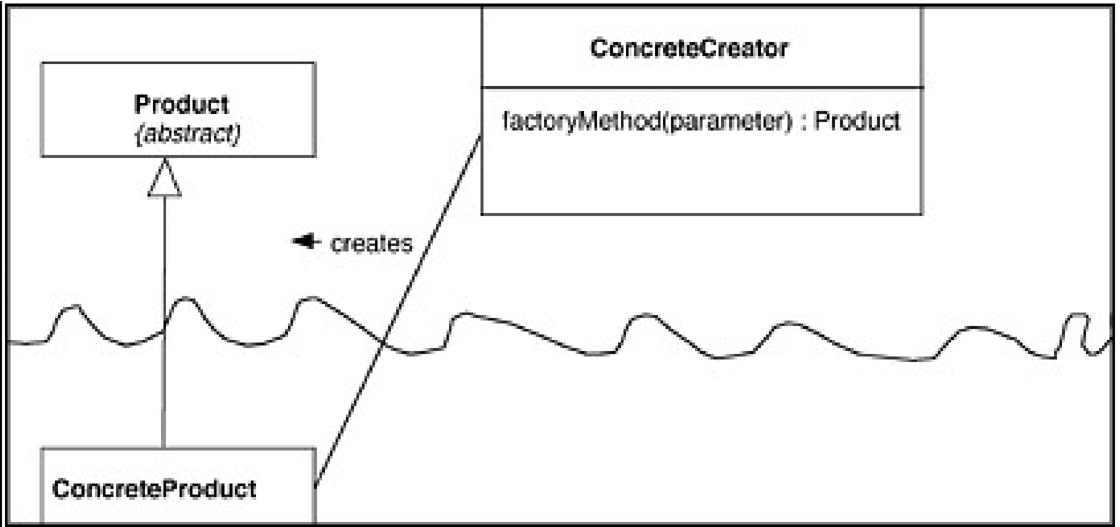
Factory Method (II)

- In particular, you should think of the Factory Method when any of the following conditions hold:
  - You do not know at compile time which specific subclasses need to be instantiated.
  - You want to defer the choice of which objects to create to a subclass.
  - A class delegates its work to a helper class, and you want to remove explicit information about which class the work is delegated to.

Use of an abstract Creator class
It creates instances of an abstract `Product` class. Concrete subclasses of the `Creator` class create particular concrete instances of the Product class.

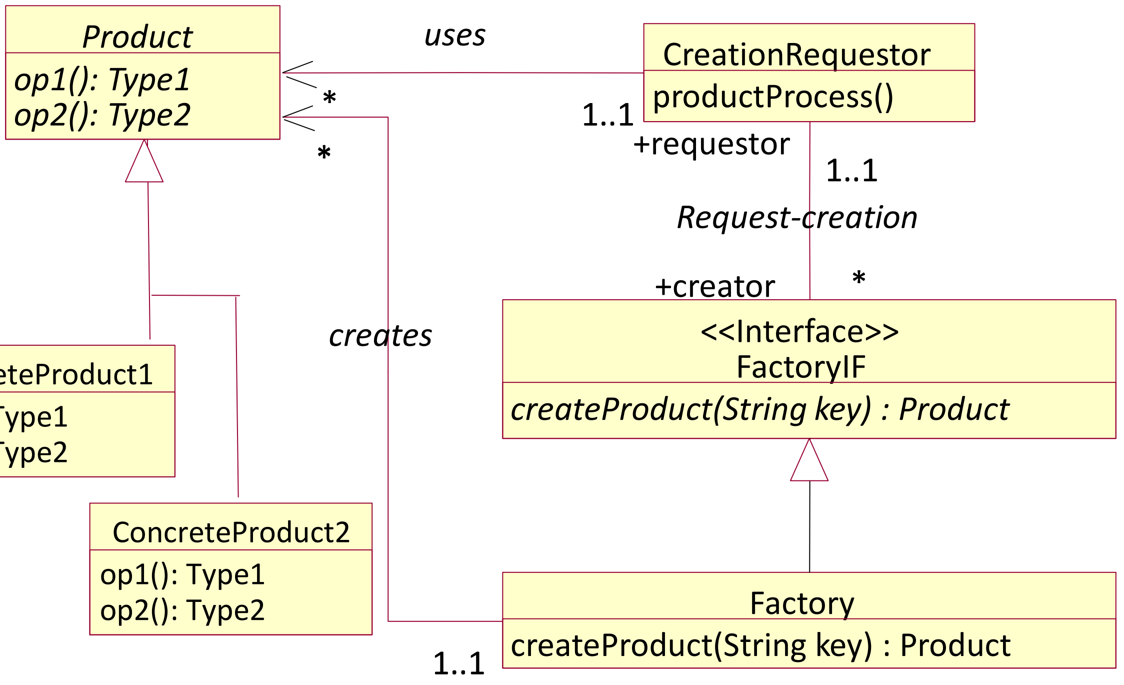
Parameterized Factory Method.
This uses a single, generally concrete, `Creator` class with a single Factory Method to create instances of different subclasses. The subclass instantiated is chosen based on the *arguments* passed to the Factory Method.





Factory Method (III)

```
public interface FactoryIF {
    public Product createProduct (String key)
           throws NoProductException;
} // interface FactoryIF

public class Factory implements FactoryIF {
    public Product createProduct(String key)
           throws NoProductException {
      if ("KEY1".equals(key))
           return new ConcreteProduct1();
      if ("KEY2".equals(key))
           return new ConcreteProduct2();
      throw new NoProductException(key);
    } // createProduct (String)
} // class Factory
. . . .
public class CreationRequestor {
    private static Product product;
    private String key;
    public CreationRequestor(String key, FactoryIF factory) throws NoProductException {
        this.key = key;
        product = factory.createProduct(key);
    } // Constructor(String, FactoryIF)
    public void productProcess() {
    . . . product.op1();
    . . . product.op2();
}
. . . .
abstract public class Product {
    private String key;
    abstract Type1 op1() ;
    abstract Type1 op1() ;
} // class Product
```



Solution:

Provide application-independent objects with an application-specific object to which they delegate the creation of other application-specific objects. Require the application-independent objects that initiate the creation of application-specific objects to assume that the objects implement a common interface.

Factory Method (IV)

Structure:

ProductIF. The objects created using this pattern must implement an interface in this role.

ConcreteProduct1, ConcreteProduct2, and so on. Classes in this role are instantiated by a Factory object. Classes in this role must implement the ProductIF interface.

CreationRequester. A class in this role is an application-independent class that needs to create application-specific classes. It does so indirectly through an instance of a class that implements the FactoryIF interface.

FactoryIF. This is an application-independent interface. Objects that create ProductIF objects on behalf of CreationRequester objects must implement this interface. Interfaces of this sort declare a method that can be called by a CreationRequester object to create concrete product objects.
Interfaces filling this role will typically have a name that includes the word *Factory,* such as DocumentFactoryIF or ImageFactoryIF.

Factory. This is an application-specific class that implements the appropriate FactoryIF interface and has a method to create ConcreteProduct objects. Classes filling this role will typically have a name, such as DocumentFactory or ImageFactory, that contains the word *Factory.*

Factory Method (V)

Consequences:

The `CreationRequester` class is independent of the class of `ConcreteProduct` objects actually created.

The set of `Product` classes that can be instantiated may be changed dynamically.

The indirection between the initiation of object creation and the determination of which class to instantiate can make it more difficult for maintenance programmers to understand.

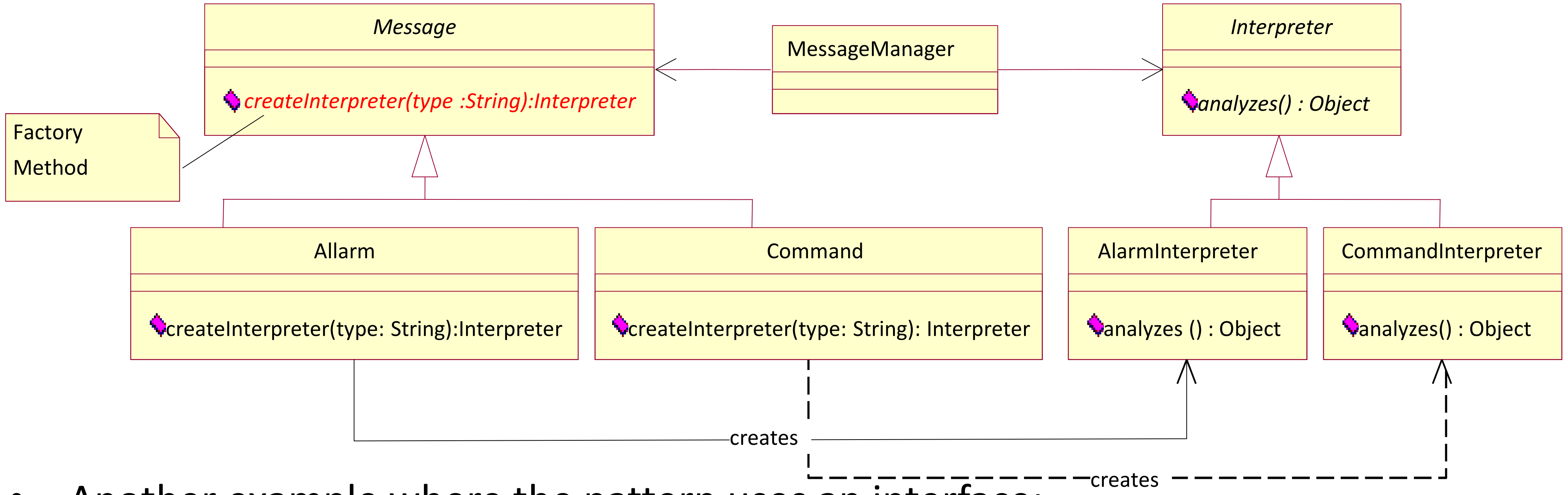
Exemple

Consider an application which manages messages belonging to several types.

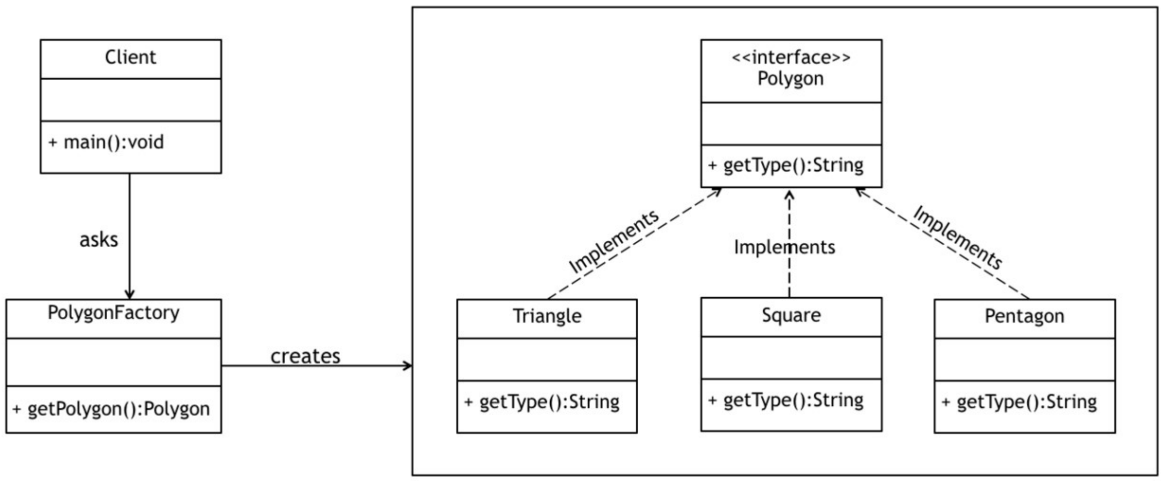
The messages need an accurate analysis for identification the message semantics. This analysis is due to an interpreter which is created when the analysis is needed and then deleted when the semantics is obtained. The interpreters are in a hierarchy similar to that of messages. The abstract class `Message` supplies a method called *`createsInterpreter`* (a Factory Method) which allows an object `MessageManager` to create appropriate interpreters for several messages. The subclasses of the class `Message` redefine this method in order to deliver the interpreter corresponding to the message type.

Factory Method (VI)

- Because the subclasses are each other relatives, the example uses an abstract class rather an interface.



- Another example where the pattern uses an interface:



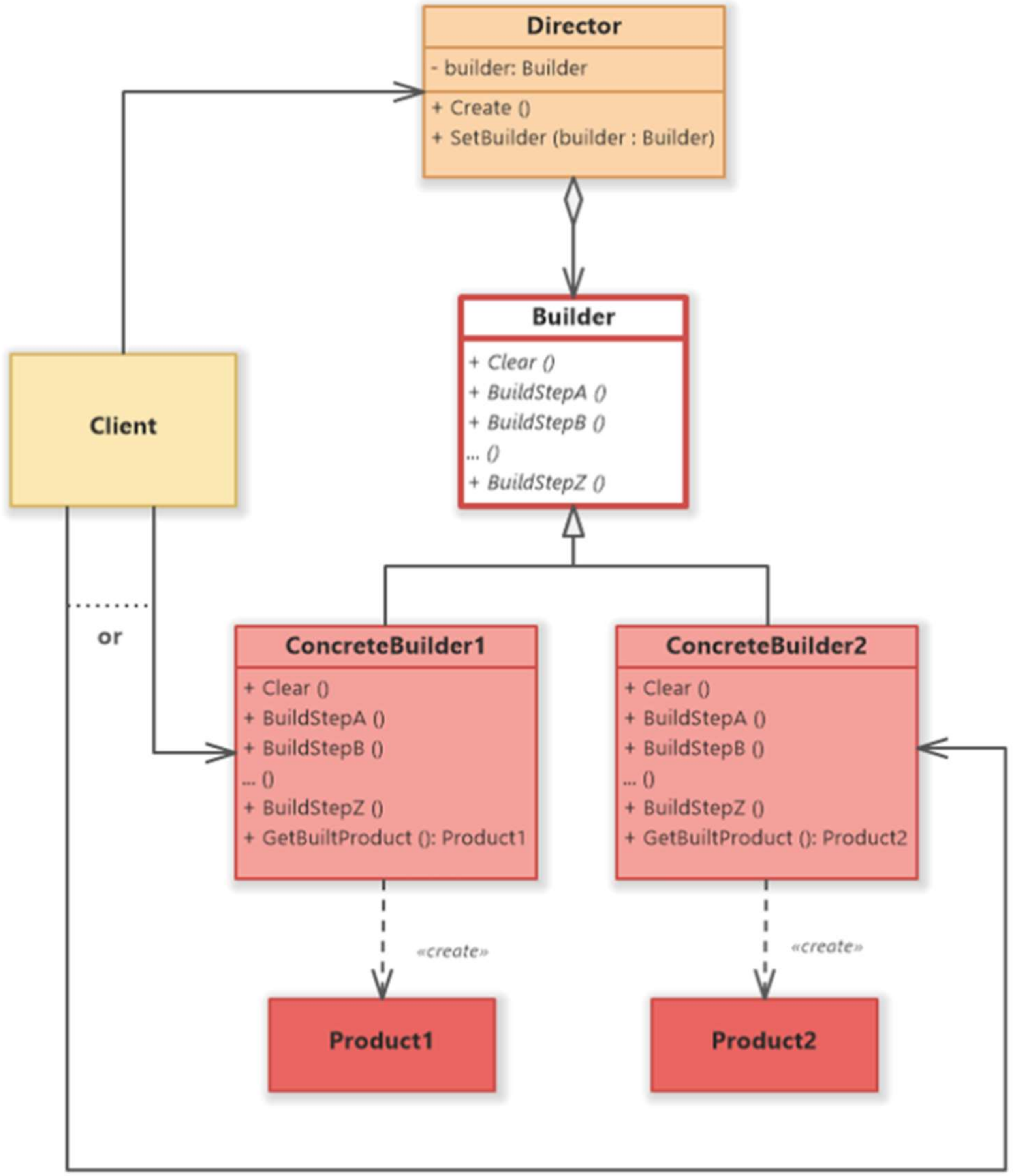
Builder (I)

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations. The pattern allows a client object to construct a complex object by specifying only its type and content. The client is shielded from the details of the object's construction.

Motivation: Consider the problem of writing an email gateway program. The program receives e-mail messages that are in MIME format (Multipurpose Internet Mail Extensions). It forwards them in a different format for different kinds of e-mail systems. This situation is a good fit for the Builder pattern. It is straightforward to organize this program with an object that parses MIME messages. Each message to parse is paired with a builder object that the parser uses to build a message in the required format. As the parser recognizes each header field and message body part, it calls the corresponding method of the builder object it is working with. the MessageManager class is responsible for collecting MIME formatted e-mail messages and initiating their transmission. The e-mail messages it directly manages are instances of the MIMEMsg class.

Builder (II)

Structure

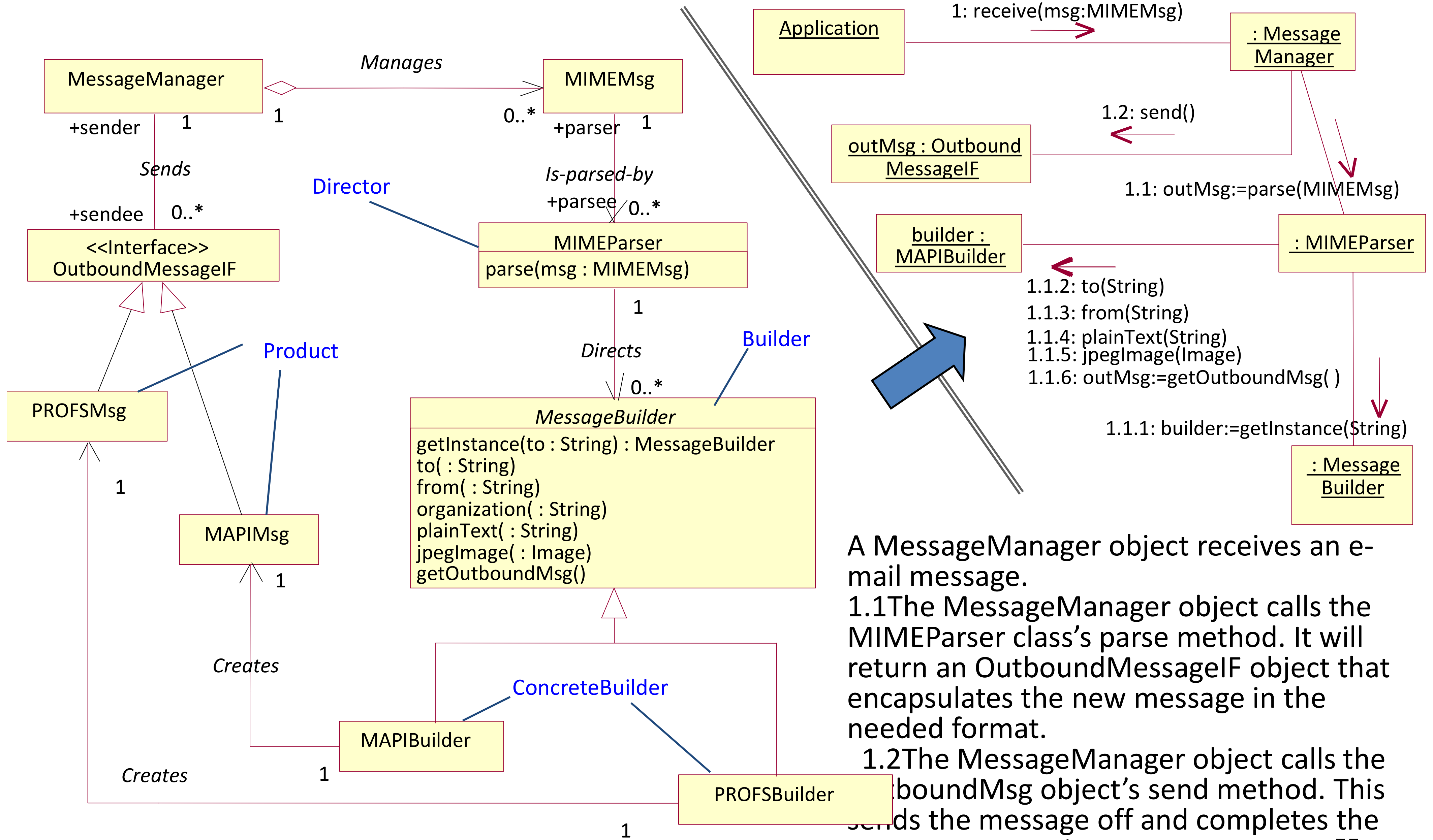


Builder (III)

In order to benefit from polymorphism, MessageBuilder is an abstract builder class. It defines methods that correspond to the various header fields and body types that MIME supports. It declares abstract methods that correspond to required header fields and the most common body types. It declares these methods abstract because all concrete subclasses of MessageBuilder should define these methods. Implementation of the methods is in MessageBuilder’s subclasses which are specialized for different e-mail systems (exp.: MAPI si PROFS).
The MessageBuilder class also defines a class method called getInstance. From the message’s destination address, the getInstance method determines the message format needed for the new message. It returns an instance of the subclass of Message-Builder appropriate for the format of the new message to the MIMEParser object.

The builder classes create product objects that implement the OutboundMsgIF interface. This interface defines a method called send that is intended to send the e-mail message wherever it is supposed to go.

Builder - Example



A MessageManager object receives an e-mail message.
1.1The MessageManager object calls the MIMEParser class's parse method. It will return an OutboundMessageIF object that encapsulates the new message in the needed format.
1.2The MessageManager object calls the [illegible]boundMsg object's send method. This [illegible]ds the message off and completes the message processing.

Builder (IV)

Structure

Builder : specifies an abstract interface for creating parts of a Product object.

ConcreteBuilder :

- constructs and assembles parts of the product by implementing the Builder interface.
- defines and keeps track of the representation it creates.
- provides an interface for retrieving the product.

Director

- constructs an object using the Builder interface.

Product

- represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Applicability

- – the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- – the construction process must allow different representations for the object that's constructed.

Builder (IV)

The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so the same construction process can create different representations.

<u>Colaborations</u>

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

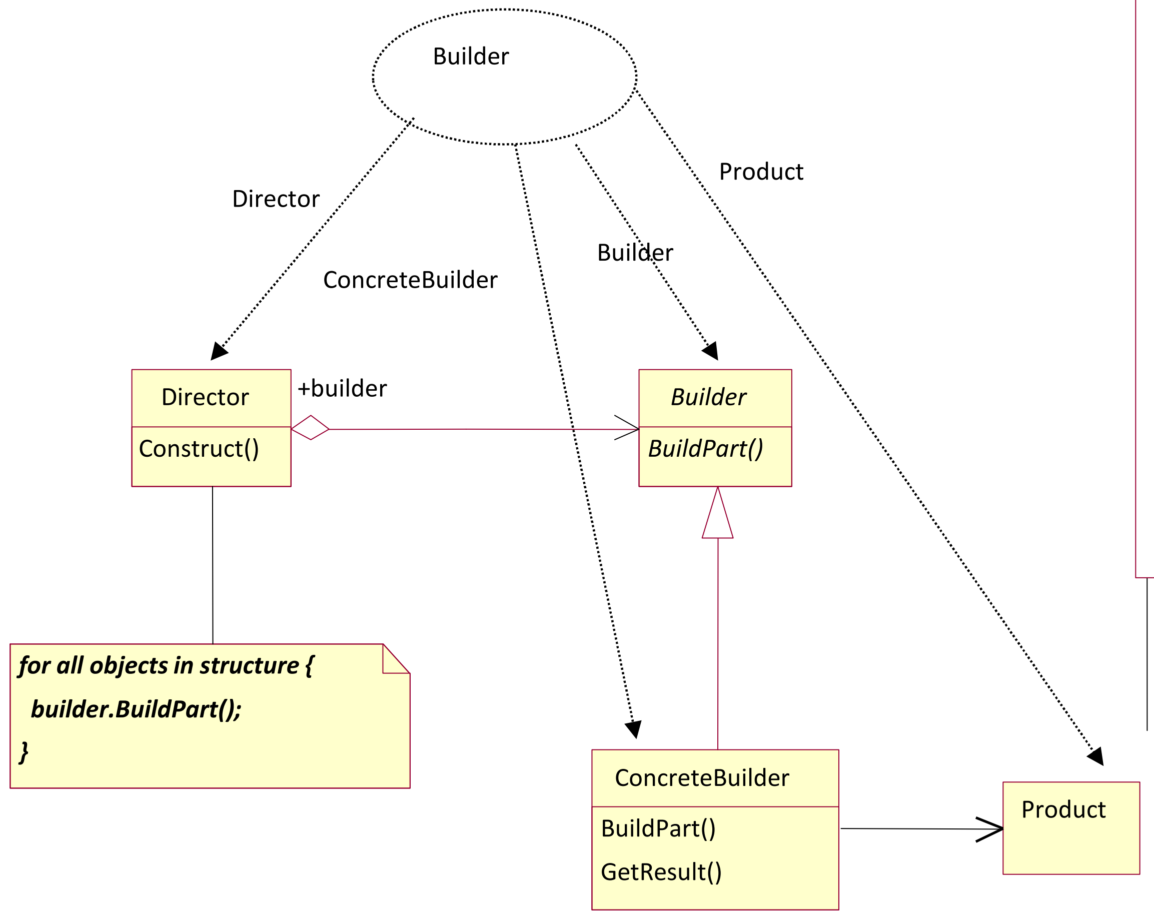
<u>Consequences</u>

It lets you vary a product's internal representation. .

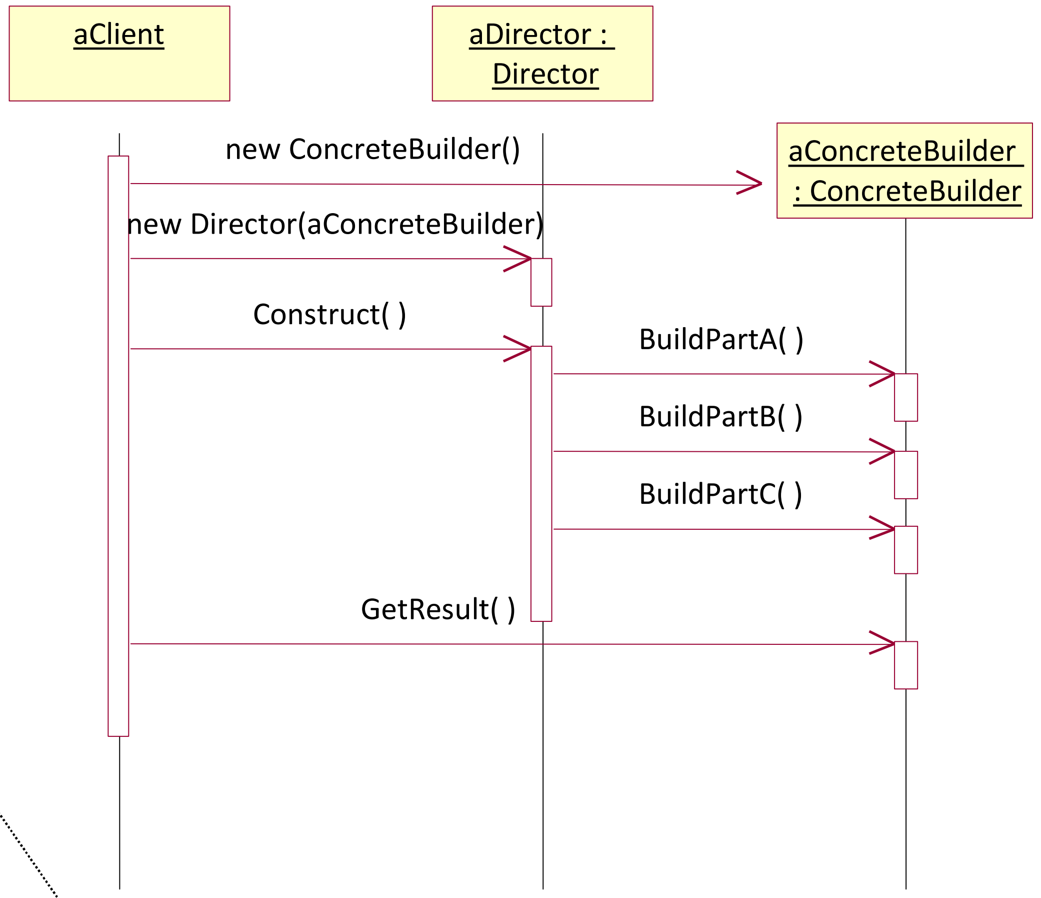
It isolates code for construction and representation

It gives you finer control over the construction process.

Builder (V)



for all objects in structure {
builder.BuildPart();
}



Builder (VI)

Implementation:

1. However, methods that provide optional content or supplementary information about the structure of the content may be unnecessary or even inappropriate for some data representations. Providing a default do-nothing implementation for such methods saves effort in the implementation of concrete builder classes that do not need such methods.
2. Organizing concrete builder classes so that calls to content-providing methods simply add data to the product object is often good enough. In some cases, there will be no simple way to tell the builder where in the finished product a particular piece of the product will go. In those situations, it may be simplest to have the content-providing method return an object to the director that encapsulates such a piece of the product. The director object can then pass the object to another content-providing method in a way that implies the position of the piece of the product within the whole product.

Related Patterns

Interface. Builder uses the Interface pattern for hidding the ProductIF objects.
Composite. The object built with Builder is a Composite object.
Factory Method. Builder uses the Factory Method pattern to decide what Builder class will be instantiated.
Layered Initialization. The Builder pattern uses the Layered Initialization pattern for creation of ConcreteBuilder objects.
Marker Interface. ProductIF interface uses thr Marker Interface pattern.
Visitor. The Visitor pattern allows the client object to be more closely coupled to the construction of the new complex object than the Builder pattern allows. Instead of describing the content of the objects to be built through a series of method calls, the information is presented in bulk as a complex data structure.

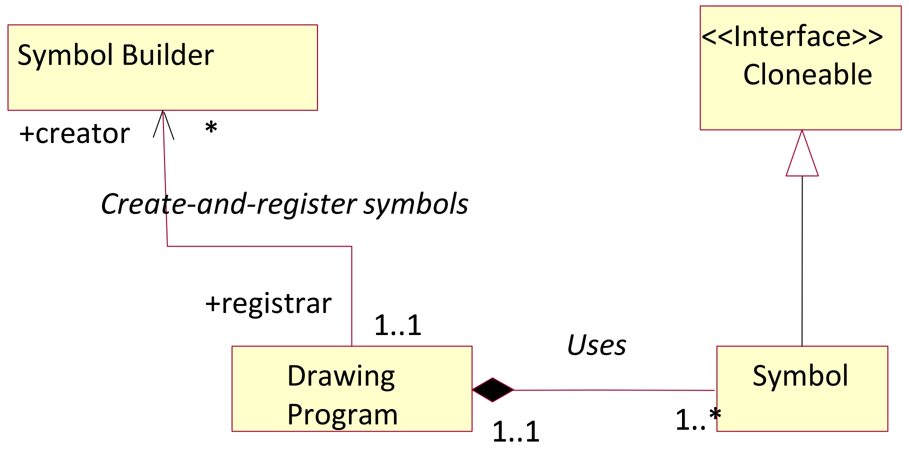
Prototype (I)

Intent:Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. Allows an object to create customized objects without knowing their exact class or the details of how to create them

Motivation: A Computer-Assisted Design (CAD) program that allows its users to draw diagrams from a palette of symbols. The program will have a core set of built-in symbols. However, people with different and specialized interests will use the program and will want additional symbols that are specific to their interests. It must be possible to provide additional sets of symbols that users can add to the program to suit their needs.

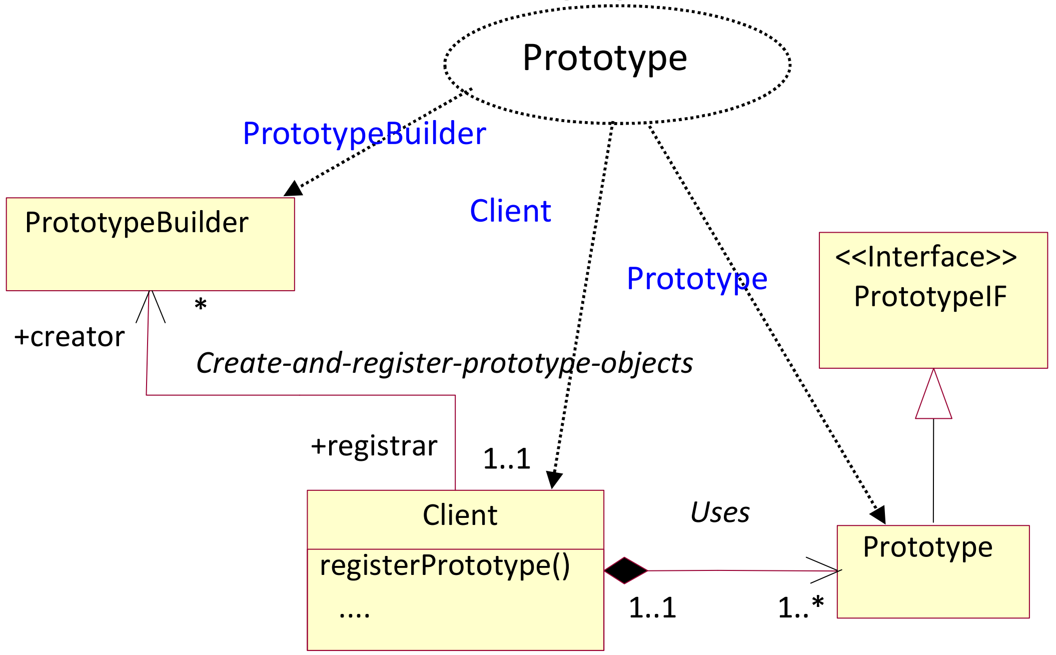
A solution is to provide the drawing program with previously created objects to use as prototypes for creating similar objects.

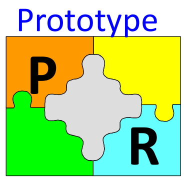
The most important requirement for objects prototypes is that they have a method, typically called *clone*, that returns a new object that is a copy of the original object.



The drawing program maintains a collection of prototypical Symbol objects. It uses the Symbol objects by cloning them. SymbolBuilder objects create Symbol objects and register them with the drawing program.

Prototype (II)





Structure

- **Client.** The client class represents the rest of the program for the purposes of the Prototype pattern. The client class needs to create objects that it knows little about. Client classes will have a method (*registerPrototype* or *registerSymbol*) that can be called to add a prototypical object to a client object's collection.
- **Prototype.** Classes in this role implement the *PrototypeIF* interface and are instantiated for the purpose of being cloned by the client. They can be also commonly abstract classes with a number of concrete subclasses.
- **PrototypeIF.** All prototype objects must implement the interface. The client class interacts with prototype objects through this interface. Interfaces in this role should extend the Cloneable interface so that all objects that implement the interface can be cloned
- **PrototypeBuilder.** This corresponds to any class instantiated to supply prototypical objects to the client object. Such classes should have a name that denotes the type of prototypical object that they build, such as SymbolBuilder.

Prototype (III)

Implementation

A. An implementation issue is how the *PrototypeBuilder* objects add objects to a client object's palette of prototypical objects:

1. The simplest strategy is for the client class to provide a method for this purpose, which *PrototypeBuilder* objects can call. A possible drawback is that the *PrototypeBuilder* objects will need to know the class of the client object.
2. The *PrototypeBuilder* objects can be shielded from knowing the exact class of the client objects by providing an interface or abstract class for the client class to implement or inherit.

B. Another issue is to implement the clone operation for the prototypical objects. Two basic strategies for clone operation:

1. *Shallow copying:* the variables of the cloned object contain the same values as the variables of the original object and all object references are to the same objects.
2. *Deep copying:* the variables of the cloned object contain the same values as the variables of the original object, except that variables that refer to objects refer to copies of the objects referred to by the original object.

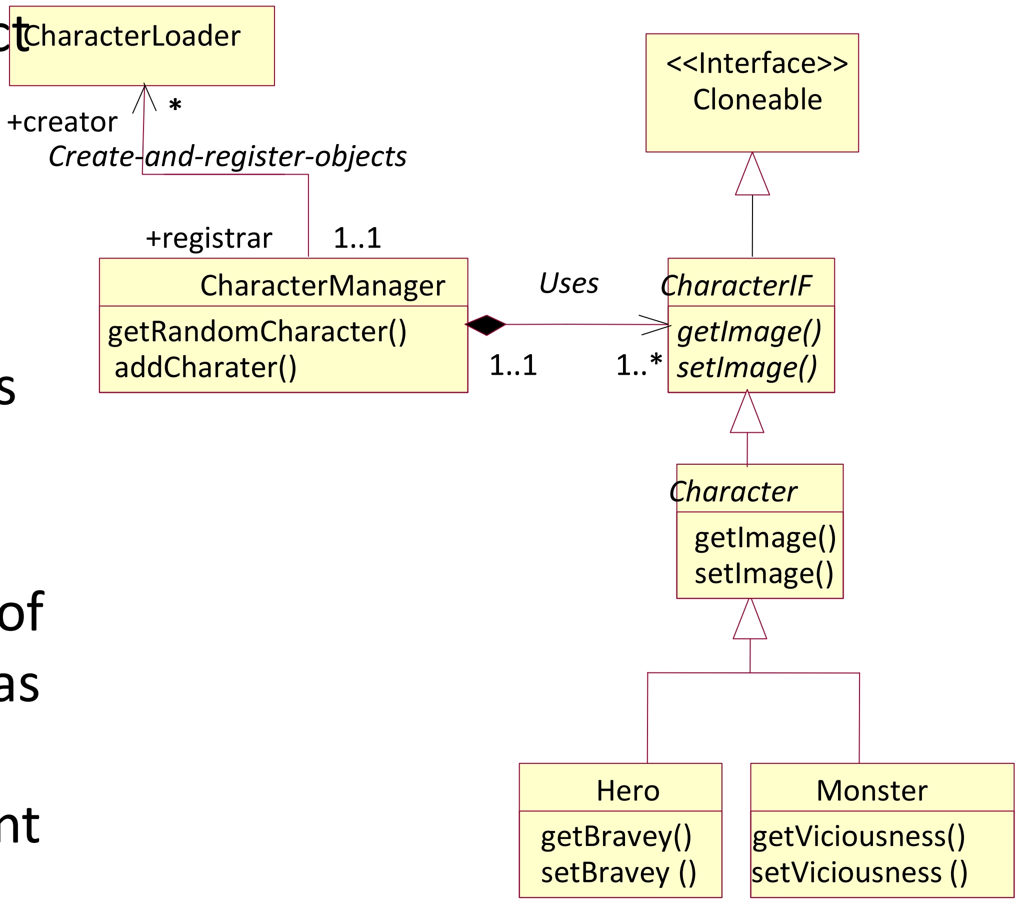
Prototype (IV)

Consequences

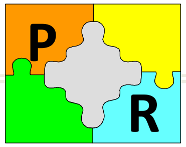
1. *Adding and removing products at run-time.* Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. A program can dynamically add and remove prototypical objects at runtime. This is a distinct advantage offered by none of the other creational patterns.
2. *Specifying new objects by varying values.* A *PrototypeBuilder* object may provide the additional flexibility of allowing new prototypical objects to be created by object composition and changes to the values of object attributes.
3. The client object may also be able to create new kinds of prototypical objects. In the drawing program example we looked at previously, the client object could very reasonably allow the user to identify a sub-drawing and then turn the sub-drawing into a new symbol.
4. *Configuring an application with classes dynamically.* Some run-time environments let you load classes into an application dynamically.
5. The client class is independent of the exact class of the prototypical objects that it uses. Also, the client class does not need to know the details of how to build the prototypical objects.
6. *Reduced subclassing.* There is no need to organize prototypical objects into any sort of class hierarchy.

Prototype - Example

- Suppose an interactive role-playing game to write. The game allows the user to interact with simulated characters. One of the expectations for the game is that people who play want to interact with new characters. An add-on to the game that consists of a few pre-generated characters and a program to generate additional characters are needed.
- The characters in the game are instances of a relatively small number of classes such as Hero, Fool, Villain, and Monster. What makes instances of the same class different from each other is the different attribute values that are set for them, such as the images that are used to represent them, height, weight, intelligence, and dexterity.



Prototype – Coding Example



```
import java.awt.Image;
public interface CharacterIF extends Cloneable {
    public String getName() ;
    public void setName(String name) ;
    public Image getImage() ;
    public void setImage(Image image) ;
    public int getStrength() ;
    public void setStrength(int strength) ;
 ... }
public abstract class Character implements CharacterIF  {
  public Object clone() {
       try {
          return super.clone();
       } catch (CloneNotSupportedException e) {
           throw new InternalError();
       }
   }
   public String getName() { return name; }
   public void setName(String name) { this.name=name; }
    public Image getImage() { return image; }
    public void setImage(Image image) {
      this.image = image; }
   ... }
public class Hero extends Character {
    private int bravery;
 ...
    public int getBravery() { return bravery; }
    public void setBravery(int bravery) {
      this.bravery = bravery;
   }
}
```

```
public class CharacterManager {
    private Vector characters = new Vector();
 ...
   Character getRandomCharacter() {
       int i = (int)(characters.size()*Math.random());
      Character c = (Character)characters.elementAt(i);
      return (Character)c.clone();
    }
    void addCharacter(Character character) {
       characters.addElement(character);
   }
}

class CharacterLoader {
   private CharacterManager mgr;
    CharacterLoader(CharacterManager cm) {
      mgr = cm;
   }
    int loadCharacters(String fname) {
      int objectCount = 0;
      try {
          InputStream in;
          in = new FileInputStream(fname);
          in = new BufferedInputStream(in);
          ObjectInputStream oIn = new ObjectInputStream(in);
          while(true) {
             Object c = oIn.readObject();
             if (c instanceof Character) {
                mgr.addCharacter((Character)c);
             }
          }
       } catch (Exception e) { }
       return objectCount;
    }
}
```

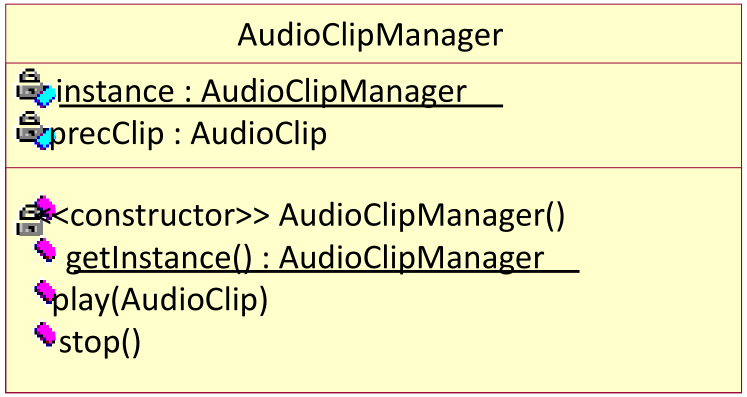
Singleton (I)

Intent: Ensure a class only has one instance, and provide a global point of access to it. Objects that use an instance of that class use the same instance

Motivation. Some classes which involve the central management of a resource should have exactly one instance. The resource may be external, such as an object that manages the reuse of database connections, or internal, such as an object that keeps an error count and other statistics for a compiler.

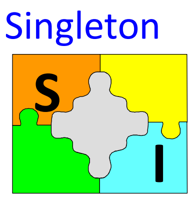
Suppose you need to write a class that could be used to ensure that no more than one audio clip is played at a time.

To avoid the undesirable situation of two audio clips playing at the same time, the class you write should stop the previous audio clip before starting the next audio clip. A way to design a class to implement this policy while keeping the class simple is to ensure that there is only one instance of the class shared by all objects that use that class. If all requests to play audio clips go through the same object, then it is simple for the object to stop the last audio clip it started before starting the next audio clip.


AudioClipManager
instance : AudioClipManager
precClip : AudioClip
<<constructor>> AudioClipManager()
getInstance() : AudioClipManager
play(AudioClip)
stop()

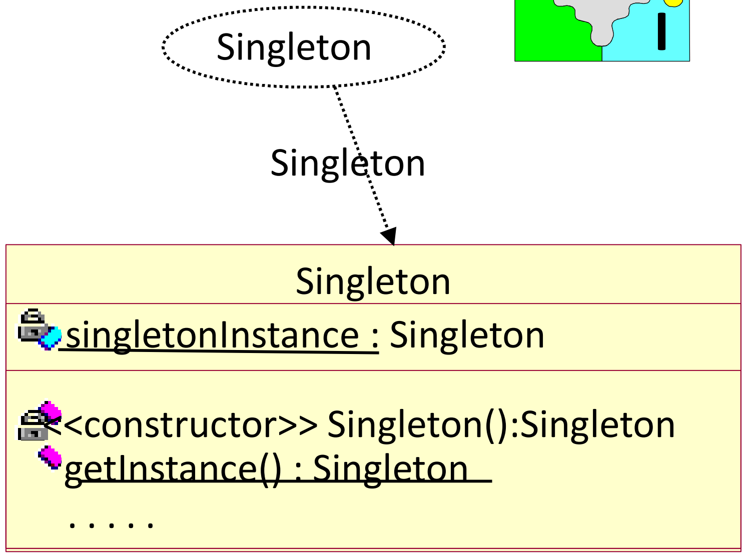
A solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created and it can provide a way to access the instance. This is the Singleton pattern.

Singleton (II)



Solution

A singleton class has a static variable that refers to the one instance of the class you want to use. This instance is created when the class is loaded into memory. You should implement the class in a way that prevents other classes from creating additional instances. That means ensuring that all of the class's constructors are private.


Singleton
Singleton
Singleton
singletonInstance : Singleton
<<constructor>> Singleton():Singleton
getInstance() : Singleton
. . . . .

Applicability

Use the Singleton pattern when:

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

Participants

- **Singleton**
  - defines an *getInstance* operation that lets clients access its unique instance. *getInstance* is a class (static) operation .
  - may be responsible for creating its own unique instance.

Singleton (III)

Consequences

The Singleton pattern has several benefits:

1. *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.
2. *Reduced name space.* The Singleton avoids polluting the name space with global variables that store sole instances.
3. *Permits refinement of operations and representation.* The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class.
4. *Permits a variable number of instances.* The pattern makes it easy to change your mind and allow more than one instance of the class. Only the operation that grants access to the Singleton instance needs to change.
5. *More flexible than class operations.* Another way to package a singleton's functionality is to use static operations. But it is hard to change a design to allow more than one instance of a class. Moreover, static member functions are never virtual, so subclasses can't override them polymorphically.

Singleton (IV)

Singleton

```
public class AudioClipManager implements AudioClip{
    private static AudioClipManager instance  = new AudioClipManager();
    private AudioClip prevClip;
   private AudioClipManager() { }
   public static AudioClipManager getInstance() {
       return instance;
   }
   public void play(AudioClip clip) {
       if (prevClip != null)
         prevClip.stop();
        prevClip = clip;
        clip.play();
    }

   public void loop(AudioClip clip) {
       if (prevClip != null)
         prevClip.stop();
         prevClip = clip;
         clip.loop();
    }
    public void stop() {
       if (prevClip != null)
         prevClip.stop();

 }
```

Object Pool (I)

Intent: Manage the reuse of objects when a type of object is expensive to create or only a limited number of a kind of object can be created.

Motivation

Writing of a class library to provide access to a proprietary database. Clients will send queries to the database through a network connection. The database server will receive queries through the network connection and return the results through the same connection. To query the database, the program must have a connection to the database. A convenient way for programmers who will use the library to manage connections is for each part of a program that needs a connection to create its own connection. However, creating database connections that are not needed is bad for a few reasons:

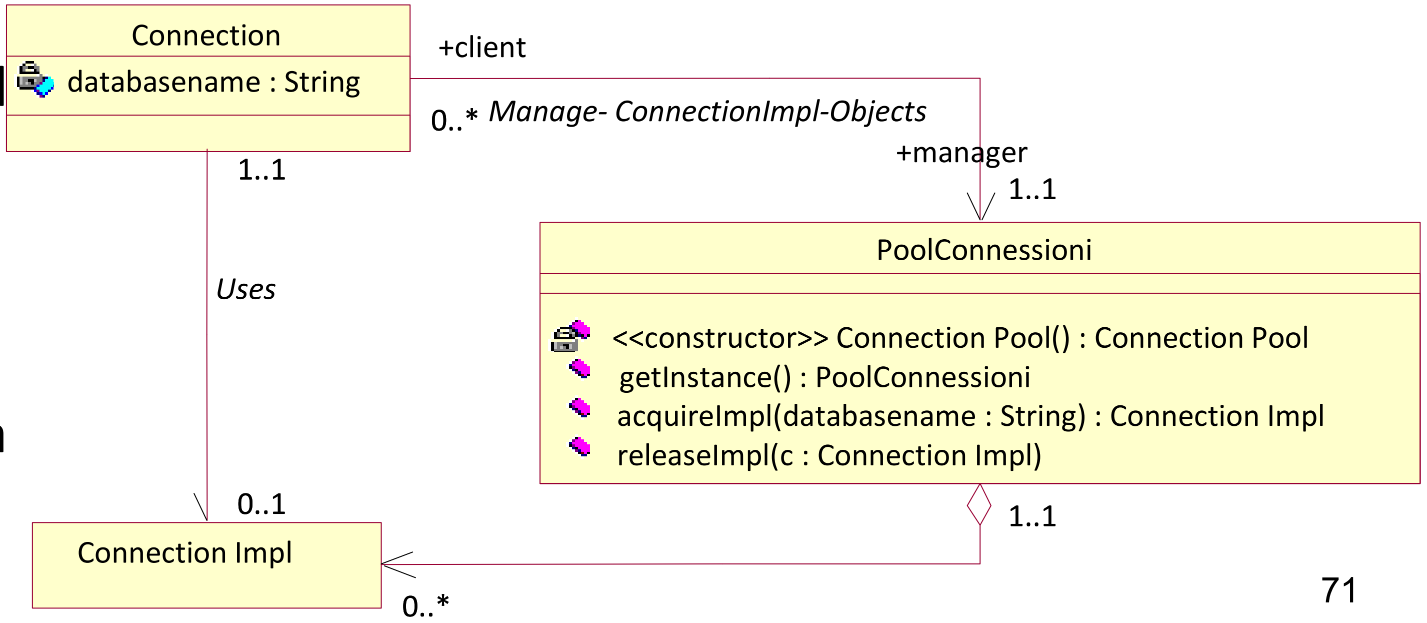
- It can take a few seconds to create each database connection.
- The more connections there are to a database, the longer it takes to create new connections.
- Each database connection uses a network connection. Some platforms limit the number of network connections that they allow.

Object Pool (II)

Solution

- The library will manage database connections based on the premise that a program's database connections are interchangeable. So long as a database connection is in a state that allows it to convey a query to the database, it does not matter which of a program's database connections is used. Using this observation, the database access library will be designed to have a two-layer implementation of database connections.
- A class called Connection will implement the upper layer. Programs that use the database access library will directly create and use Connection objects. Connection objects will identify a database, but will not directly encapsulate a database connection. They will be paired with a ConnectionImpl objects. ConnectionImpl objects encapsulate an actual database connection.

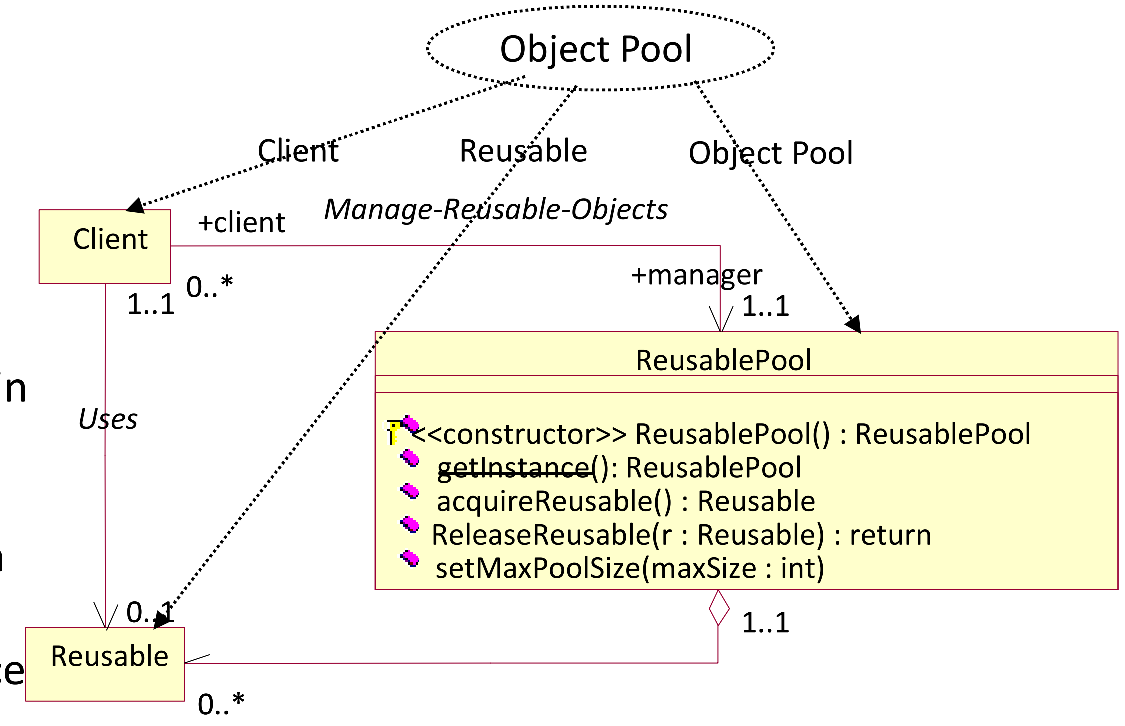
The library will create and manage `ConnectionImpl` objects by maintaining a pool of them that are not currently paired up with a `Connection` object.


Connection
databasename : String
+client
0..*
Manage- ConnectionImpl-Objects
+manager
1..1
1..1
Uses
0..1
PoolConnessioni
<<constructor>> Connection Pool() : Connection Pool
getInstance() : PoolConnessioni
acquireImpl(databasename : String) : Connection Impl
releaseImpl(c : Connection Impl)
1..1
Connection Impl
0..*

Object Pool (III)

Structure:

- **Reusable.** Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.
- **Client.** Instances of classes in this role use Reusable objects.
- **ReusablePool.** Instances of classes in this role manage Reusable objects for use by Client objects. It is usually desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the ReusablePool class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its getInstance method to get the one instance of the ReusablePool class.
- In many applications of the Object Pool pattern, there are reasons for limiting the total number of Reusable objects that may exist.


Object Pool
Client
Reusable
Object Pool
Manage-Reusable-Objects
Client
+client
0..*
1..1
+manager
1..1
Uses
ReusablePool
<<constructor>> ReusablePool() : ReusablePool
~~getInstance~~(): ReusablePool
acquireReusable() : Reusable
ReleaseReusable(r : Reusable) : return
setMaxPoolSize(maxSize : int)
0..1
1..1
Reusable
0..*

Object Pool (IV)

Applicability:

1. A program may not create more than a limited number of instances of a particular class.
2. Creating instances of a particular class is sufficiently expensive that creating new instances of that class should be avoided.
3. A program can avoid creating some objects by reusing objects that it has finished with rather than letting them be garbage-collected.
4. The instances of a class are interchangeable. If you have multiple instances on hand, you can arbitrarily choose one to use for a purpose. It does not matter which one you choose.
5. Resources can be managed centrally by a single object or in a decentralized way by multiple objects. It is easier to achieve predictable results by managing resources centrally with a single object.
6. Some objects consume resources that are in short supply. Some objects may consume a lot of memory. Some objects may periodically check to see whether some condition is true, thereby consuming CPU cycles and perhaps network bandwidth. If the resources that an object consumes are in short supply, then it may be important that the object stop using the resource when the object is not being used.

Object Pool (V)

```
import java.util.Hashtable;
import java.util.Vector;
/**  Istanze di questa classe forniscono le vere connessioni.  */
class ConnessioneImpl {
   // Il nome del DB.
   private String nomeDatabase;
   // Costruttore privato
   private ConnessioneImpl(String nomeDatabase) {
       this.nomeDatabase = nomeDatabase;
       //...
   } // constructor()
   //...
   /**  restituisce il nome del DB a cui questo oggetto è connesso. */
   String getDatabaseName() {
       return nomeDatabase;
   } // getDatabaseName()
   /**     Invia una richiesta al DB e restituisce il risultato.    */
   Object inviaRichiesta(Richiesta richiesta) {
       Object risultato = null;
       //...
       return risultato;
   } // inviaRichiesta(Richiesta)
   //…
  /* Questa classe interna accede al costruttore ConnessioneImpl */
   static class PoolConnessioni {
       // L'unica istanza di questa classe
      private static PoolConnessioni ilPool = new PoolConnessioni();
       /* Questa hash table associa i nomi dei DB con i Vector che
       contengono i pool di connessioni per quel DB. */
      private Hashtable poolDictionary = new Hashtable();
// Costruttore è privato per impedire ad altre classe di creare istanze.
      private PoolConnessioni() {}
       /*  Restituisce l'unica istanza della classe  */
      public static PoolConnessioni getInstance() {
          return ilPool;
      } // getInstance()
```

```
     /* Restituisce una connessione dal pool appropriato o crea una
      nuova se il pool è vuoto.
     @param nomeDatabase Il nome del DB a cui viene fornito
      ConnessioneImpl.    */
     public synchronized ConnessioneImpl esegueImpl
                  (String nomeDatabase) {
      Vector pool=(Vector)poolDictionary.get(nomeDatabase);
        if (pool != null) {
           int size = pool.size();
           if (size > 0)
             return (ConnessioneImpl)pool.remove(size-1);
        } // if null
         /*  Nessuna ConnessioneImpl nel pool, viene creata
             una nuova. */
        return new ConnessioneImpl(nomeDatabase);
     } // esegueImpl(String)
     /** Aggiunge un ConnessioneImpl al pool appropriato.   */
     public synchronized void rilasciaImpl(ConnessioneImpl impl) {
        String nomeDatabase= impl.getDatabaseName();
      Vector pool=(Vector)poolDictionary.get(nomeDatabase);
        if (pool == null) {
           pool = new Vector();
           poolDictionary.put(nomeDatabase, pool);
        } // if null
        pool.addElement(impl);
     } // rilasciaImpl(ConnessioneImpl)
   } // class PoolConnessioni
} // class ConnessioneImpl
```

Object Pool (VI)

```
/**
        Oggetti che trasmettono richieste al DB implementano questa interfaccia
*/
interface Richiesta {
   //...
} // interface Richiesta


/**
 Classe pubblica utilizzata al esterno dalla libreria di accesso al DB per rappresentare la connessione con il DB.
 */
public class Connessione {
   private final static ConnessioneImpl.PoolConnessioni connessionePool = ConnessioneImpl.PoolConnessioni.getInstance();
   private String nomeDatabase;

   //...

   /**
    Invia una richiesta al database e restituisce il risultato.
    */
   Object inviaRichiesta(Richiesta request) {
      Object risultato;
      ConnessioneImpl impl = connessionePool.esegueImpl(nomeDatabase);
      risultato = impl.inviaRichiesta(request);
      connessionePool.rilasciaImpl(impl);
      return risultato;
   } // inviaRichiesta(Richiesta)
} // class Connessione
```

Object Pool (VII)

```
public class SoftObjectPool implements ObjectPoolIF {
   private ArrayList pool ;
   private CreationIF creator ;
   private int instanceCount;
   private int maxInstances ;
   private Class poolClass;
   public SoftObjectPool( Class poolClass,  CreationIF creator ) {
      this( poolClass, creator, Integer.MAX_VALUE);
    }
   public SoftObjectPool( Class poolClass,  CreationIF creator ) {
       this.creator = creator;
       this.poolClass = poolClass;
       pool = new ArrayList();
    }
   public int getSize() {
       synchronized (pool) {
          return pool.size();
    }
   public int getInstanceCount() {
       return instanceCount;
   }
   public int getMaxInstances() {
       return maxInstances;
   }
   public void setMaxInstances(int newValue) {
       maxInstances = newValue;
   }
   public Object getObject() {
       synchronized (pool) {
          Object thisObject = removeObject();
          if (thisObject!=null) {
             return thisObject;
          }
          if (getInstanceCount() < getMaxInstances()){
             return createObject();
         } else {
             return null;
         }
     }
 }
```

```
  public Object waitForObject() throws InterruptedException {
   synchronized (pool) {
       Object thisObject = removeObject();
       if (thisObject!=null) {
         return thisObject;
       }
       if (getInstanceCount() < getMaxInstances()){
          return createObject();
       } else {
          do {
              pool.wait();
              thisObject = removeObject();
          } while (thisObject==null);
          return thisObject;
        } // if
    }
 }
private Object removeObject() {
     while (pool.size()>0) {
     SoftReference thisRef = (SoftReference)pool.remove(pool.size()-1);
        Object thisObject = thisRef.get();
        if (thisObject!=null) {
            return thisObject;
        }
        instanceCount—;
      }
      return null;
  }
  private Object createObject() {
      Object newObject = creator.create();
      instanceCount ++;
      return newObject;
   }
  public void release( Object obj ) {
      if ( obj == null ) {
         throw new NullPointerException();
      }
     if ( !poolClass.isInstance(obj)) {
        String actualClassName = obj.getClass().getName();
        throw new ArrayStoreException(actualClassName);
     }
      synchronized (pool) {
        pool.add(obj);
        pool.notify();
      }
    }
}   // SoftObjectPool
```

Using Soft References

- The Object Pool pattern keeps objects that are not being used available for reuse. If the program that is using an object pool is running out of memory, then you would like the garbage collector to be able to remove objects from the pool and reclaim the memory that they occupy. You can arrange for the garbage collector to do this by using soft references.
- Soft references are implemented in the Java API by the class *java.lang.ref.SoftReference*. A reference to another object is passed to the constructor of a *SoftReference* object. Immediately after a Soft-Reference object is constructed, its get method returns the object reference that was passed to its constructor. The interesting thing about SoftReference objects is that they are special to the garbage collector. If the only live reference to an object is through a SoftReference object, then the garbage collector will set the reference in the SoftReference object to null so that it can safely reclaim the storage occupied by the referenced object.
- If the object pool refers to objects in the pool through soft references, then the garbage collector will reclaim the storage occupied by the objects if there are no other references to the objects and the Java virtual machine (JVM) is running low on memory.

Summary - Creational Patterns

Pattern Name	When it is used	Varying parts
Abstract Factory	Provides an interface for creating families of objects (products) related to each other and (product) dependent on each other.	Families of objects
Factory Method	Defines an interface for creating an object, but let subclasses decide which class to instantiate. Instantiation is resubmitted to subclasses	The subclass of the object to be instatiated
Builder	Separates the construction of a complex object from its representation so that the same construction process can create different representations.	How is created the compound object
Prototype	Specifies the types of objects that can be created using an instance of the prototype, and create objects by copying this prototype.	The class of the object to be instatiated
Singleton	Ensures that a class has a single instance and and provide a global point of access to this instance	The unique instance of the class
Object Pool	Manage the reuse of objects when this type wastes too many of its reusable objects or limits the number of objects to be created	Number of objects

1.4. Structural Patterns

1. Filter
2. Composite
3. Adapter (class and object)
4. Iterator
5. Bridge
6. Façade
7. Decorator
8. Virtual Proxy

- These Patterns use composition to merge objects and classes into larger structures.
- They show you how to glue different pieces of a system together in a flexible and extensible fashion.
- Structural patterns help you guarantee that when one of the parts changes, the entire structure does not need to change.
- They also show you how to recast pieces that do not fit (but that you need to use) into pieces that do fit.

Structural Patterns

- Structural patterns show you how to glue different pieces of a system together in a flexible and extensible fashion. Structural patterns help you guarantee that when one of the parts changes, the entire structure doesn't need to change. They also show you how to recast pieces that don't fit (but that you need to use) into pieces that do fit. They all involve connections between objects.
- A structural design pattern serves as a blueprint for how different classes and objects are combined to form larger structures.
- Structural class patterns use inheritance to combine the interfaces or implementations of multiple classes.
- Structural class patterns are relatively rare. Structural object patterns use object composition to combine the implementations of multiple objects. They can combine the interfaces of all the composed objects into one unified interface or they can provide a completely new interface.

Filter (I)

Intent:

- Objects that have compatible interfaces, but perform different transformations and computations on data streams, can be dynamically connected to perform arbitrary operations.
- The Filter pattern allows objects that have compatible interfaces and perform different transformations and computations on streams of data to be dynamically connected to perform arbitrary operations on streams of data.

Motivation:

Unix: The *uniq* program normally copies all the lines it reads to its output. However, when it finds consecutive lines that contain identical characters, it copies only the first such line to its output.

wc does a simple analysis of a data stream. It produces a count of the number of characters, words, and lines that were in the data stream.

Classes that perform simple transformations and analyses on data streams tend to be very generic in nature. When writing such classes, you cannot anticipate all the ways they will be used. Some applications will want to apply transformations and analyses to only selected parts of a data stream. These classes should be written in a way that allows great flexibility in how their instances can be connected.

One way to accomplish this flexibility is to define a common interface for all of these classes so an instance of one can use an instance of another without having to take into account which class the object is an instance of.

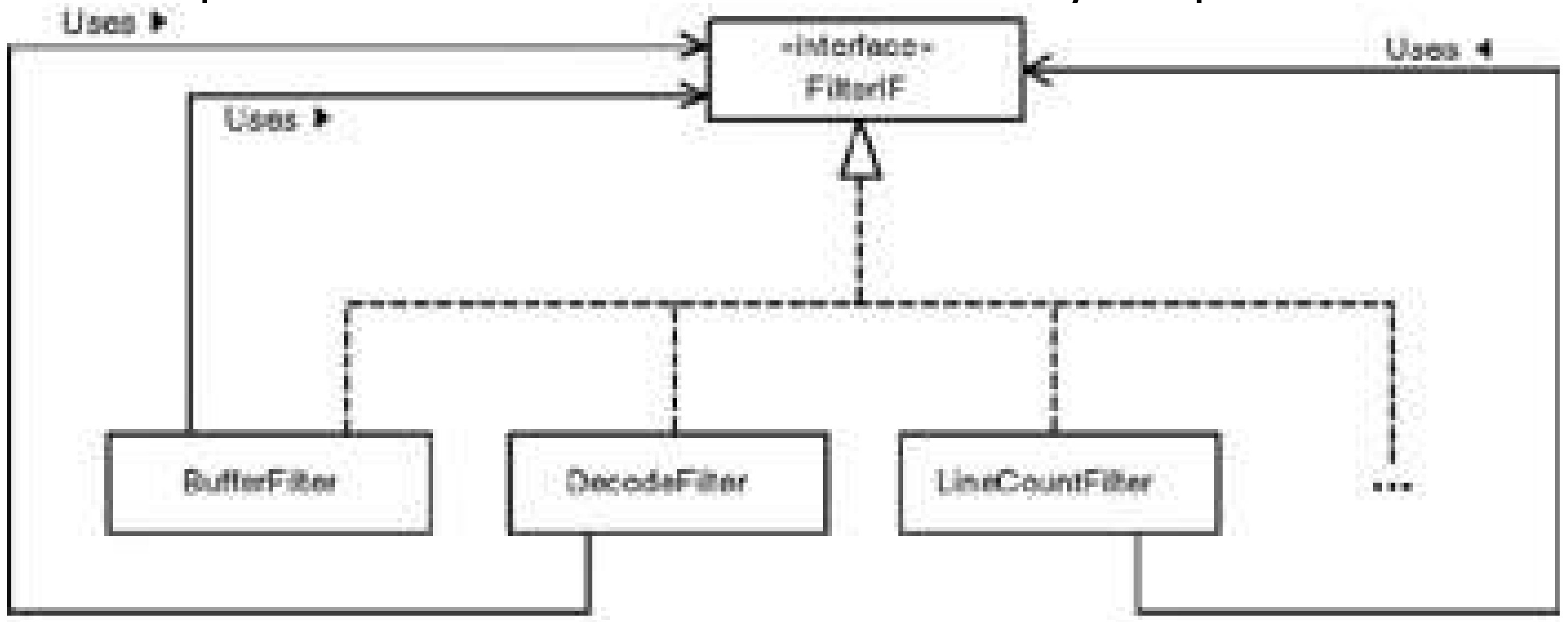
Filter (II)

Applicability

1. Classes that implement common data transformations and analyses can be used in a great variety of programs.
2. You can dynamically combine data analysis and transformation objects by connecting them together.
3. The use of transformation/analysis objects should be transparent to other objects.

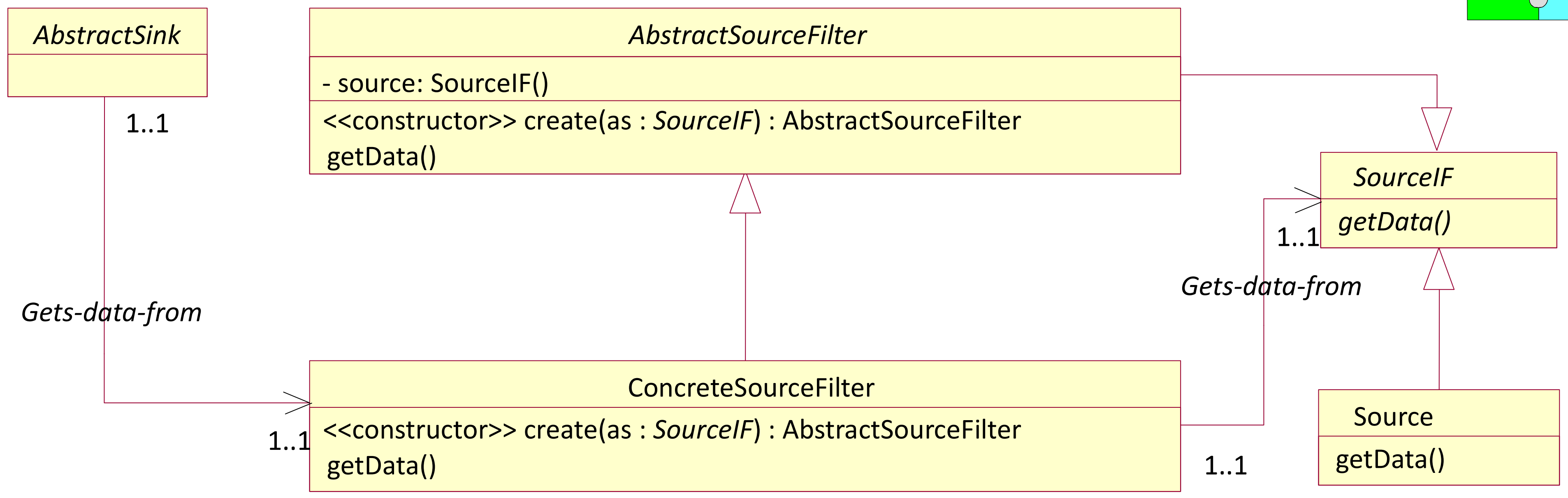
Solution

Base a solution on common interfaces and delegation. The Filter pattern organizes the classes that participate in it as data sources, data sinks, and data filters. The data filter classes perform the transformation and analysis operations.



There are two basic forms of the Filter pattern.

Source Filter (I)



Intent:

Data sink objects get data by calling methods in data sources. This form of Filter is sometimes called a *pull filter*.

Structure Source filter

SourceIF. An interface in this role declares one or more methods that return data when it is called (getData).

Source. A class in this role is responsible primarily for providing data rather than transforming or analyzing data. Classes in this role are also required to implement the SourceIF interface.

Source Filter (II)

Structure Source filter

AbstractSourceFilter. A class in this role is an abstract superclass of classes that transform and analyze data. It has a constructor that takes an argument that is a SourceIF object. Instances of this class delegate the fetching of data to the SourceIF object that was passed to their constructor.

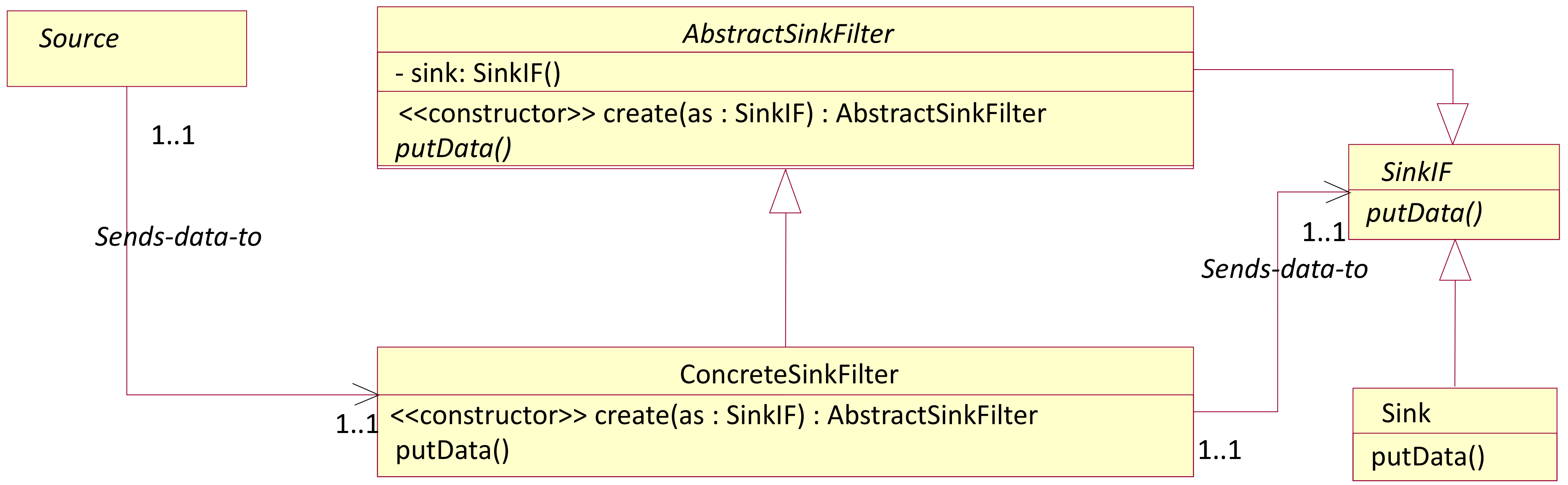
AbstractSourceFilter classes typically have an instance variable that is set by their constructor and refers to the SourceIF object passed to their constructor. However, to ensure that their subclasses do not depend on this instance variable, the instance variable should be private.

AbstractSourceFilter classes typically define a getData method that simply calls the getData method of the SourceIF object referred to by the instance variable.

ConcreteSourceFilter. Classes in this role are a concrete subclass of an AbstractSourceFilter class. They override the getData method that they inherit to perform the appropriate transformation or analysis operations.

Sink. Instances of classes in this role call the getData method of a SourceIF object. Unlike ConcreteSourceFilter objects, instances of Sink classes use data without passing it on to another AbstractSourceFilter object.

Sink Filter (I)



Intent

Data source objects pass data to methods of data sink objects. This form of Filter is sometimes called a *push filter.*

Structure Sink Filter

SinkIF. An interface in this role declares one or more methods that take data through one of its parameters (putData).

Sink. A class in this role is responsible primarily for receiving and processing data rather than transforming or analyzing data. Classes in this role are also required to implement the SinkIF interface. Data is passed to Sink objects by passing the data to the Sink object's putData method.

Sink Filter (II)

Structure Sink filter (cont'd)

AbstractSinkFilter. A class in this role is an abstract superclass of classes that transform and analyze data. It has a constructor that takes an argument that is a SinkIF object. Instances of this class pass data to the SinkIF object that was passed to their constructor. Because subclasses of this class inherit the fact that it implements the SinkIF interface, their instances can accept data from other objects that pass data to SinkIF objects.

AbstractSinkFilter classes typically have an instance variable that is set by their constructor and refers to the SinkIF object passed to their constructor. However, to ensure that their subclasses do not depend on this instance variable, the instance variable should be private. AbstractSinkFilter classes typically define a putData method that simply calls the putData method of the SinkIF object referred to by the instance variable.

ConcreteSinkFilter. Classes in this role are a concrete subclass of an AbstractPushFilter class. They override the putData method that they inherit to perform the appropriate transformation or analysis operations.

Source. Instances of classes in this role call the putData method of a SinkIF object.

Filter (III)

Consequences

1. The portion of a program that follows the Filter pattern can be structured as a set of sources, sinks, and filters.
2. Filter objects that do not maintain internal state can be dynamically replaced while a program is running. This property of stateless filters allows dynamic change of behavior and adaptation to different requirements at runtime.
3. It is quite reasonable for a program to incorporate both forms of the Filter pattern. However, it is unusual for the same class to participate in both forms.
4. If your design calls for filters to be dynamically added to or removed while processing a data stream, then you will need to design a mechanism to manage this change in a predictable way.

Implementation

Making filter classes independent of the programs that they are used in increases their reusability. In some cases it is needed that a filter object should use context-specific information. You could define one or more interfaces that declare methods for providing context-specific information to a filter object. If a program detects that a filter object implements one of those interfaces, it can use the interface to provide additional information to the filter.

Filter (IV)

Java Exemple

The java.io package includes the FilterReader class, which participates in the Filter pattern as an abstract source filter class. The corresponding abstract source class is Reader. Concrete subclasses of the FilterReader class include BufferedReader, FileReader, and LineNumberReader. There is no separate interface that fills the SourceIF role. The Reader class also fills the SourceIF role.

The java.io package includes the FilterWriter class, which participates in the Filter pattern as an abstract sink filter class. The corresponding abstract sink class is Writer. Concrete subclasses of the FilterWriter class include BufferedWriter, FileWriter, and PrintWriter. The Writer class also fills the SinkIF role.

A program that reads lines of text as commands and needs to track line numbers for producing error messages:

```
LineNumberReader in;
void init(String fName) {
 FileReader fin;
 try {
  fin = new FileReader(fName);
  in =new LineNumberReader(new BufferedReader(fin));
} catch (FileNotFoundException e) {
 System.out.println("Unable to open "+fName);
 ... }
 ...
```

Filter - Code

```
import java.io.IOException;
public abstract class InStream {
    public abstract int read(byte[] array) throws IOException;
  // getData()
} // class InStream

import java.io.IOException;
import java.io.RandomAccessFile;
public class FileInStream extends InStream {
   private RandomAccessFile file;
public FileInStream(String fName) throws IOException {
      file = new RandomAccessFile(fName, "r");
   } // Constructor(String)

   public int read(byte[] array) throws IOException {   // getData()
      return file.read(array);
   } // read(byte[])
} // class FileInStream

import java.io.IOException;
public class FilterInStream extends InStream {
   private InStream inStream;

  public FilterInStream(InStream inStream) throws IOException {
      this.inStream = inStream;
   } // Constructor(InStream)

    public int read(byte[] array) throws IOException {   // getData()
      return inStream.read(array);
   } // read(byte[])
} // class FilterInStream
```

```
import java.io.IOException;
public class ByteCountInStream extends FilterInStream {
   private long byteCount = 0;
   public ByteCountInStream(InStream inStream) throws IOException {
      super(inStream);
   } // Constructor(InStream)
   public int read(byte[] array) throws IOException {    //getData()
      int count;
      count = super.read(array);
      if (count >0)
        byteCount += count;
      return count;
   } // read(byte[])
   public long getByteCount() {
      return byteCount;
   } // getByteCount()
} // class ByteCountInStream

import java.io.IOException;
public class TranslateInStream extends FilterInStream {
   private byte[] translationTable;
   private final static int TRANS_TBL_LENGTH = 256;
 public TranslateInStream(InStream inStream,byte[] table) throws
        IOException {super(inStream);
     // Creates the transaltion table by copying translation data
      translationTable = new byte[TRANS_TBL_LENGTH];
      System.arraycopy(table, 0, translationTable, 0,
                   Math.min(TRANS_TBL_LENGTH, table.length));
      for (int i = table.length; i < TRANS_TBL_LENGTH; i++) {
         translationTable[i] = (byte)i;
      } // for
   } // Constructor(InStream)
   public int read(byte[] array) throws IOException {   // getData()
      int count;
      count = super.read(array);
      for (int i = 0; i < count; i++) {
         array[i] = translationTable[array[i]];
      } // for
      return count;
   } // read(byte[])
} // class ByteCountInStream
```

Composite

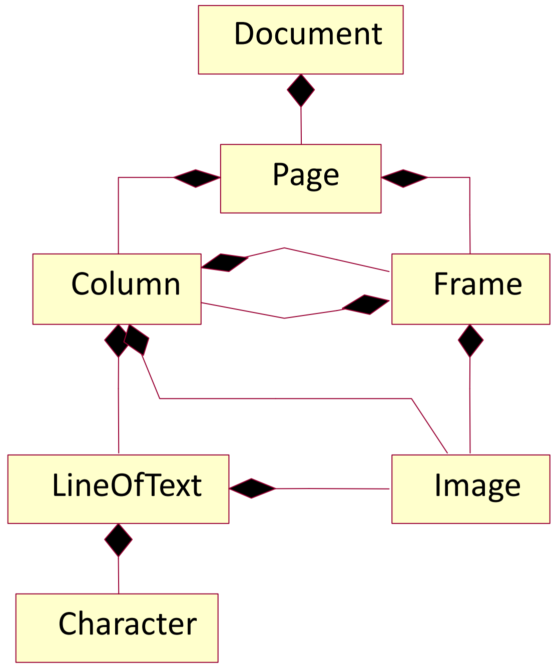
Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. It is also known as the Recursive Composition pattern. It is a partitioning pattern because during the design it is often used to recursively decompose a complex object into simpler objects.

Motivation

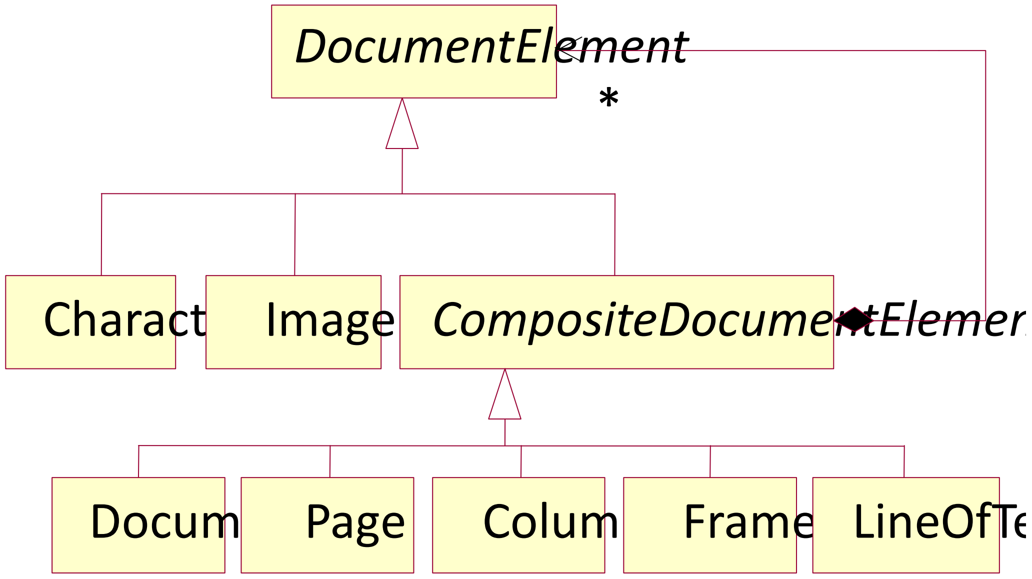
Suppose that we are writing a document formatting program. It formats characters into lines of text organized into columns that are organized into pages. However, a document may contain other elements. Columns and pages can contain frames that can contain columns. Columns, frames, and lines of text can contain images etc. as in the following class diagram that shows these relationships.

As we can see, there is a fair amount of complexity here. Page and Frame objects must know how to handle and combine two kinds of elements. Column objects must know how to handle and combine three kinds of elements.



Composite - Motivation

- The Composite pattern removes that complexity by allowing these objects to know how to handle only one kind of element. It accomplishes this by insisting that all document element classes implement a common interface.
- By applying the Composite pattern, we have introduced a common interface for all document elements and a common superclass for all container classes.
- Doing this the number of aggregation relationships is reduced to one. Management of the aggregation is now the responsibility of the `CompositeDocumentElement` class. The concrete container classes (`Document`, `Page`, `Column`, etc.) only need to understand how to combine one kind of element.

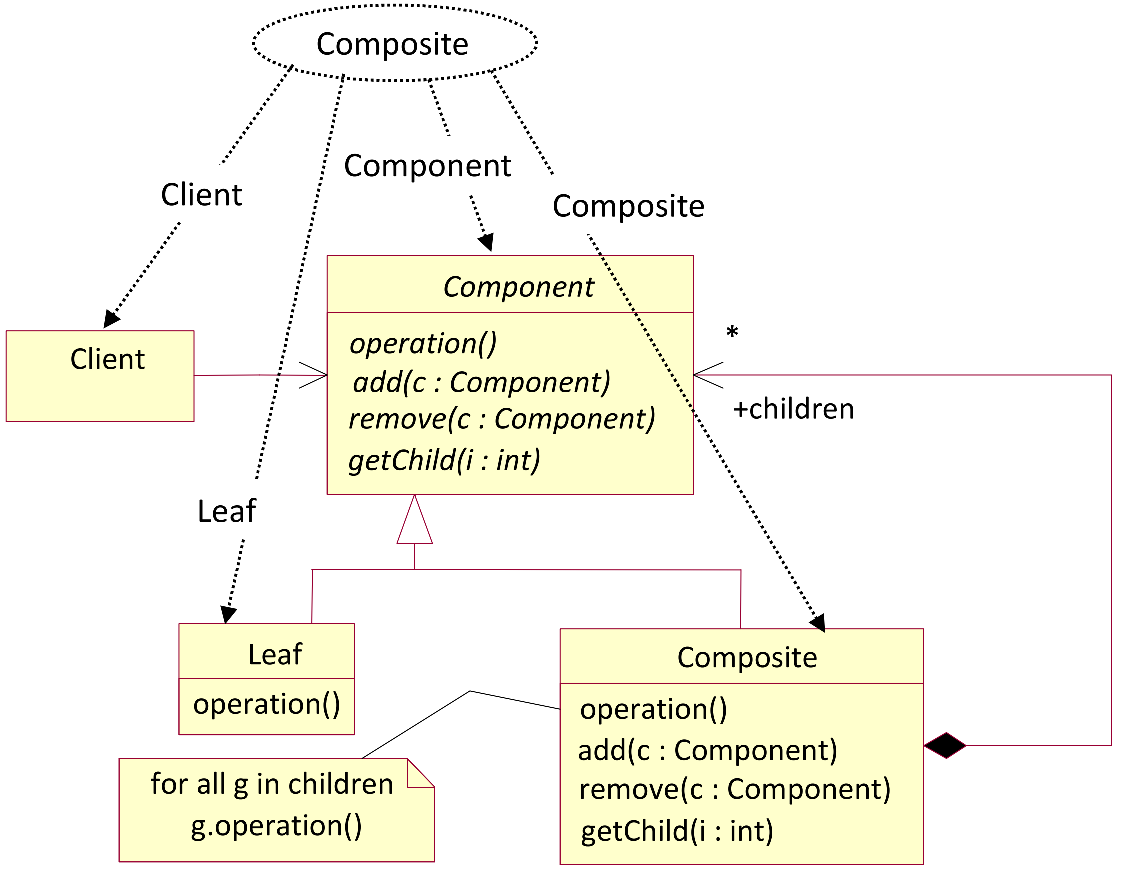


Composite - Applicability

Applicabilitaty

1. You have a complex object you want to decompose into a part-whole hierarchy of objects.
2. You want to minimize the complexity of the part-whole hierarchy by minimizing the number of different kinds of child objects that objects in the tree need to be aware of.
3. There is no requirement to distinguish between most of the part-whole relationships.

Structure



Composite - Structure

Structure

Component

- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Leaf

- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

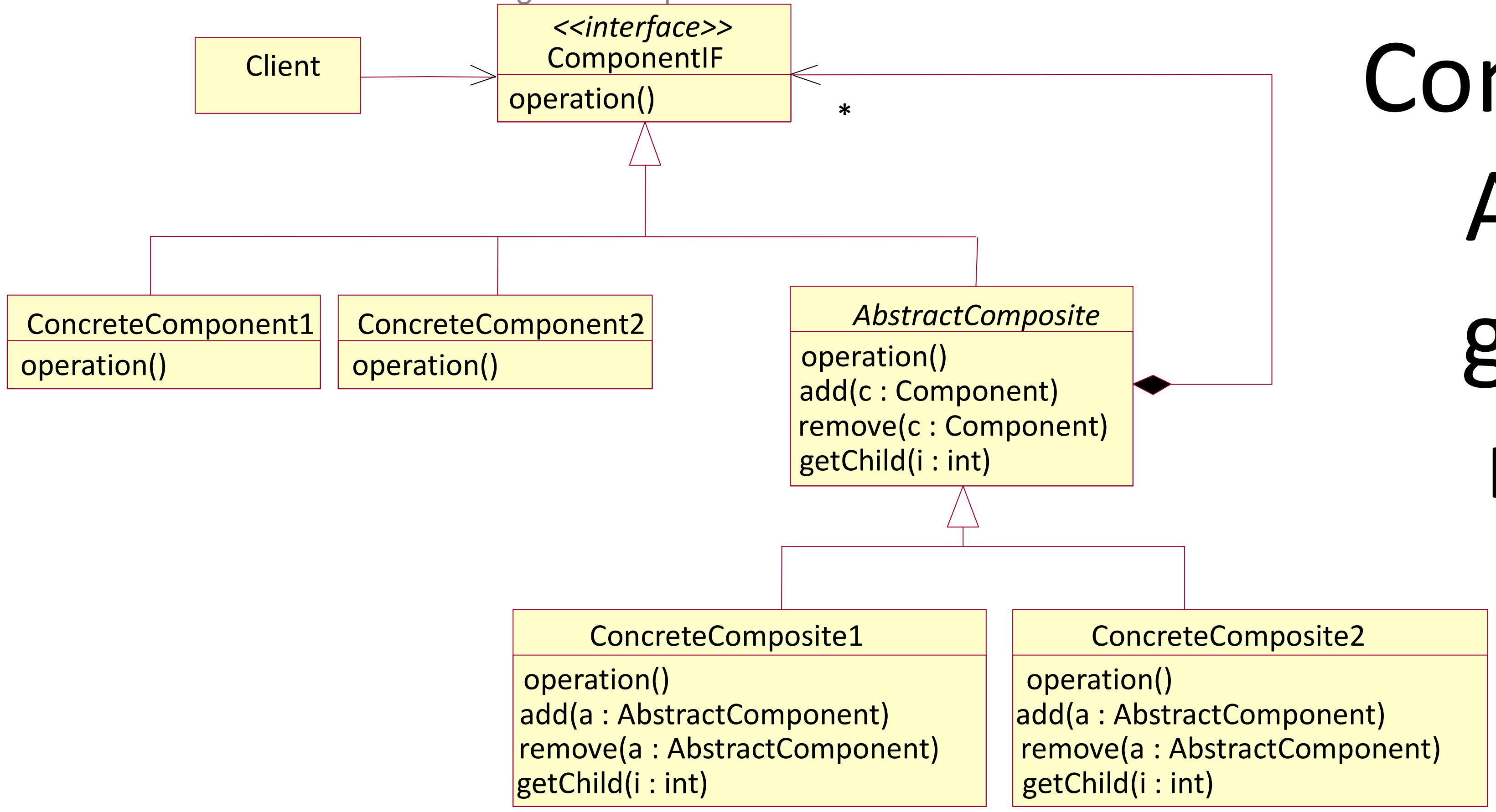
Composite

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.

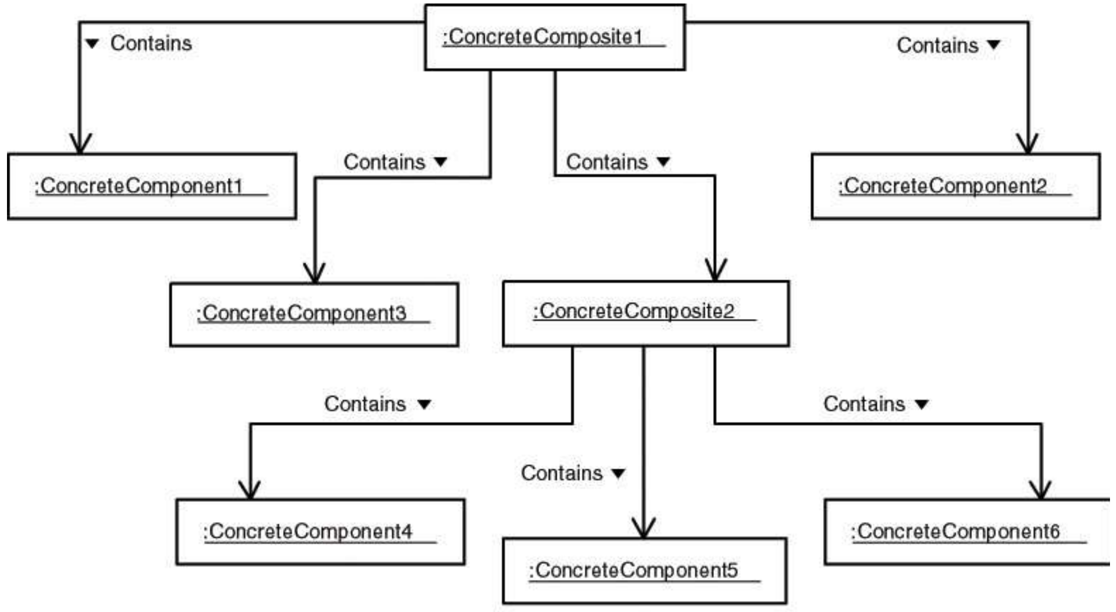
Client

- manipulates objects in the composition through the Component interface.

Composite - A more general model



- Instances of these classes can be assembled in a tree-like manner.



Composite - A more general model

Structure

- **ComponentIF.** An interface in the ComponentIF role is implemented by all the objects in the hierarchy of objects that make up a composite object. Composite objects normally treat the objects that they contain as instances of classes that implement the ComponentIF interface rather than as instances of their actual class.
- **Component1, Component2, and so on.** Instances of these classes are used as leaves in the tree organization.
- **AbstractComposite.** A class in this role is the abstract superclass of all composite objects that participate in the Composite pattern. AbstractComposite defines and provides default implementations of methods for managing a composite object's components. The add method adds a component to a composite object. The remove method removes a component from a composite object. The getChild method returns a reference to a component object of a composite object.
- **ConcreteComposite1, ConcreteComposite2, and so on.** Instances of these are composite objects that use other instances of AbstractComposite.

Conclusions

- You can access a tree-structured composite object and the objects that constitute it through the ComponentIF interface, whether they are simple objects or composite.

Composite - Conclusions

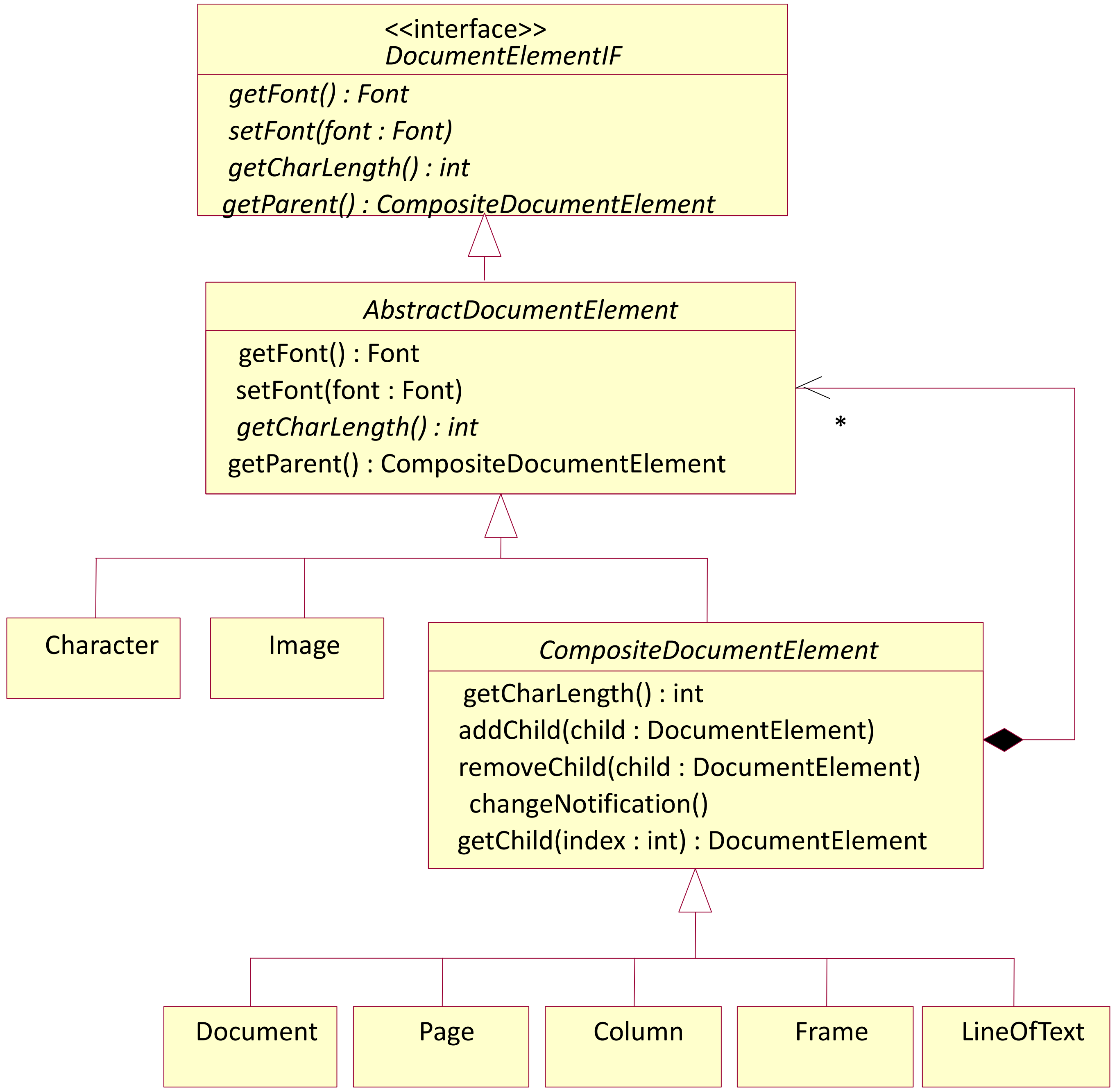
- Client objects of an AbstractComponent can simply treat it as an AbstractComponent, without having to be aware of any subclasses of AbstractComponent.
- If a client invokes a method of a ComponentIF object that is supposed to perform an operation and the ComponentIF object is an AbstractComposite object, then it may delegate the operation to the ComponentIF objects that constitute it. Similarly, if a client object calls a method of a ComponentIF object that is *not* an AbstractComposite and the method requires some contextual information, then the ComponentIF object delegates the request for contextual information to its parent.
- Some components may implement operations that are unique to that component. For example, under the Motivation of this pattern is a design for the recursive composition of a document. At the lowest level, it has a document consisting of character and image elements. It is very reasonable for the character elements of a document to have a `getFont` method. A document's image elements have no need for a `getFont` method (see the Example section). The main benefit of the Composite pattern is to allow the clients of a composite object and the objects that constitute it to be unaware of the specific class of the objects they deal with.
- The Composite pattern allows any ComponentIF object to be a child of an AbstractComposite. If you need to enforce a more restrictive relationship, then you will have to add type-aware code to AbstractComposite or its subclasses. That reduces some of the value of the Composite pattern.

Composite - Implementation

Implementation

- If classes that participate in the Composite pattern implement any operations by delegating them to their parent object, then the best way to preserve speed and simplicity is to have each instance of AbstractComponent contain a reference to its parent. It is important to implement the parent pointer in a way that ensures consistency between parent and child. It must always be true that a ComponentIF object identifies an AbstractComposite object as its parent if, and only if, the AbstractComposite identifies it as one of its children. The best way to enforce this is to modify parent and child references only in the AbstractComposite class's add and remove methods.
- The AbstractComposite class may provide a default implementation of child management for composite objects. However, it is very common for concrete composite classes to override the default implementation.
- If a concrete composite object delegates an operation to the objects that constitute it, then caching the result of the operation may improve performance. If a concrete composite object caches the result of an operation, it is important that the objects that constitute the composite notify the composite object to invalidate its cached values.

Composite - Example



Composite - Code

```
public interface DocumentElementIF {
 ...
   public CompositeDocumentElement getParent() ;
   public Font getFont() ;
   public void setFont(Font font) ;
   public int getCharLength() ;
 }  // interface DocumentElementIF

abstract class AbstractDocumentElement
                              implements DocumentElementIF {
   private Font font;
   private CompositeDocumentElement parent;
   ...
   public CompositeDocumentElement getParent() {
       return parent; } // getParent()
   protected void setParent(CompositeDocumentElement parent) {
       this.parent = parent; } // setParent(AbstractDocumentElement)
   public Font getFont() {
       if (font != null)
         return font;
      else if (parent != null)
         return parent.getFont();
       else
        return null; } // getFont()
   public void setFont(Font font) {
     this.font = font;
    } // setFont(Font)
   public abstract int getCharLength() ;
} // class AbstractDocumentElement
```

```
public abstract class CompositeDocumentElement
                              extends AbstractDocumentElement {
   private Vector children = new Vector();
   private int cachedCharLength = -1;
   public DocumentElementIF getChild(int index) {
       return (DocumentElementIF)children.elementAt(index);
   } // getChild(int)
   public    synchronized void addChild(DocumentElementIF child) {
       synchronized (child) {
          children.addElement(child);
         ((AbstractDocumentElement)child).setParent(this);
          changeNotification(); } // synchronized
    }  // addChild(DocumentElementIF)
  public synchronized
    void removeChild(AbstractDocumentElement child) {
       synchronized (child) {
          if (this == child.getParent()) {
             child.setParent(null);
          }  // if children.removeElement(child);
          changeNotification();
       } // synchronized
  } // removeChild(AbstractDocumentElement)
...
 public void changeNotification() {
       cachedCharLength = -1;
       if (getParent() != null)
         getParent().changeNotification();
    }  // changeNotification()
   public int getCharLength() {
       int len = 0;
       for (int i = 0; i < children.size(); i++) {
          AbstractDocumentElement  thisChild;
          thisChild = (AbstractDocumentElement)children.elementAt(i);
          len += thisChild.getCharLength();
       }  // for
      cachedCharLength = len;
      return len;
    } // getCharLength()
} // class CompositeDocumentElement
```

Adapter (I)

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. An adapter class implements an interface known to its clients and provides access to an instance of a class not known to its clients. An adapter object provides the functionality promised by an interface without having to assume what class is used to implement that interface.

Motivation

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

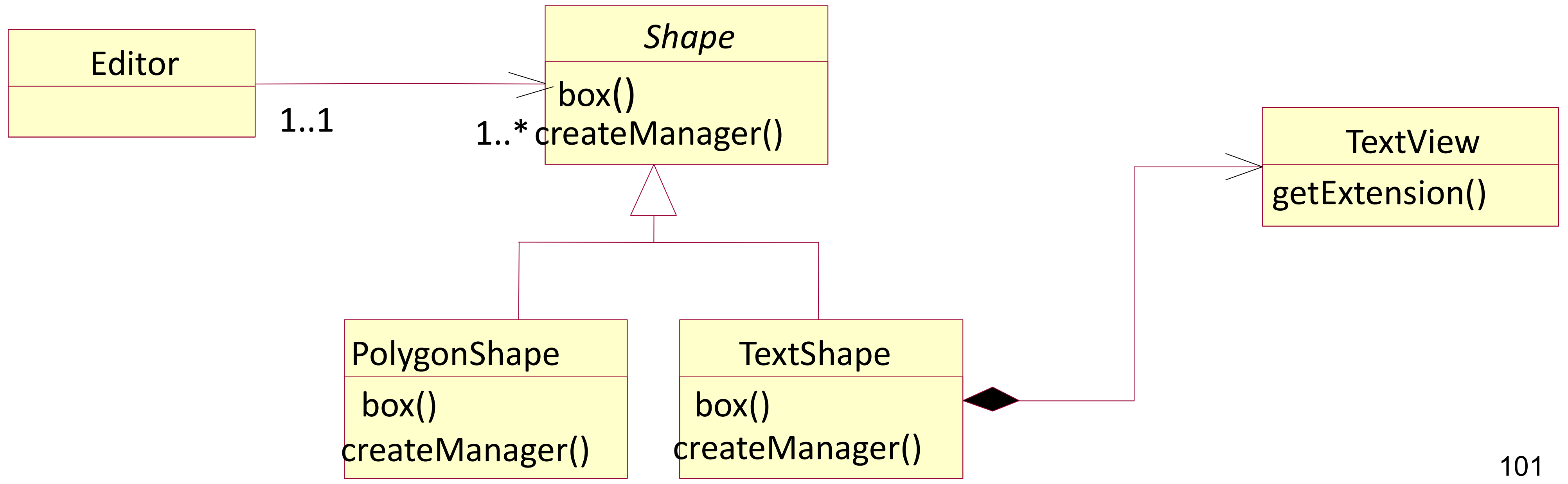
Classes for elementary geometric shapes like LineShape and PolygonShape are rather easy to implement, because their drawing and editing capabilities are inherently limited. But a TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management.

Adapter (II)

Motivation (cont'd)

Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text. Ideally we'd like to reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind. So we can't use TextView and Shape objects interchangeably.

Instead, we could define TextShape so that it *adapts* the TextView interface to Shape's. We can do this in one of two ways: (1) by inheriting Shape's interface and TextView's implementation or (2) by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface. These two approaches correspond to the class and object versions of the Adapter pattern. We call TextShape an **adapter**.



Adapter (III)

Applicability

1. you want to use an existing class, and its interface does not match the one you need.
2. you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
3. *(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

Target (Shape) defines the domain-specific interface that Client uses.

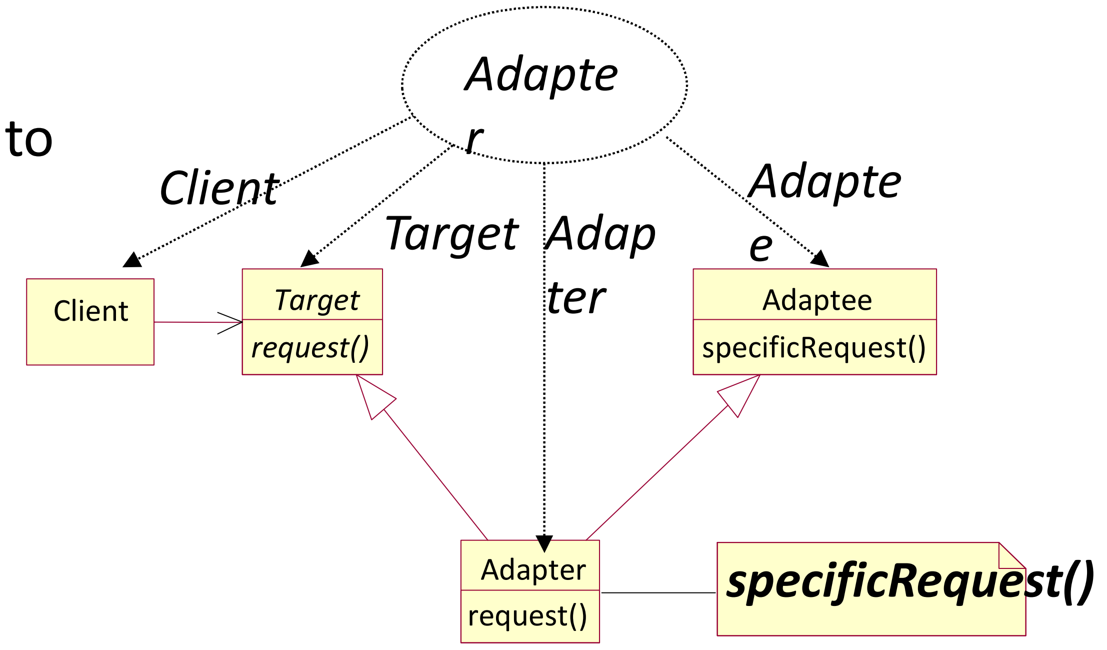
Client (DrawingEditor) collaborates with objects conforming to the Target interface.

Adaptee (TextView) defines an existing interface that needs adapting.

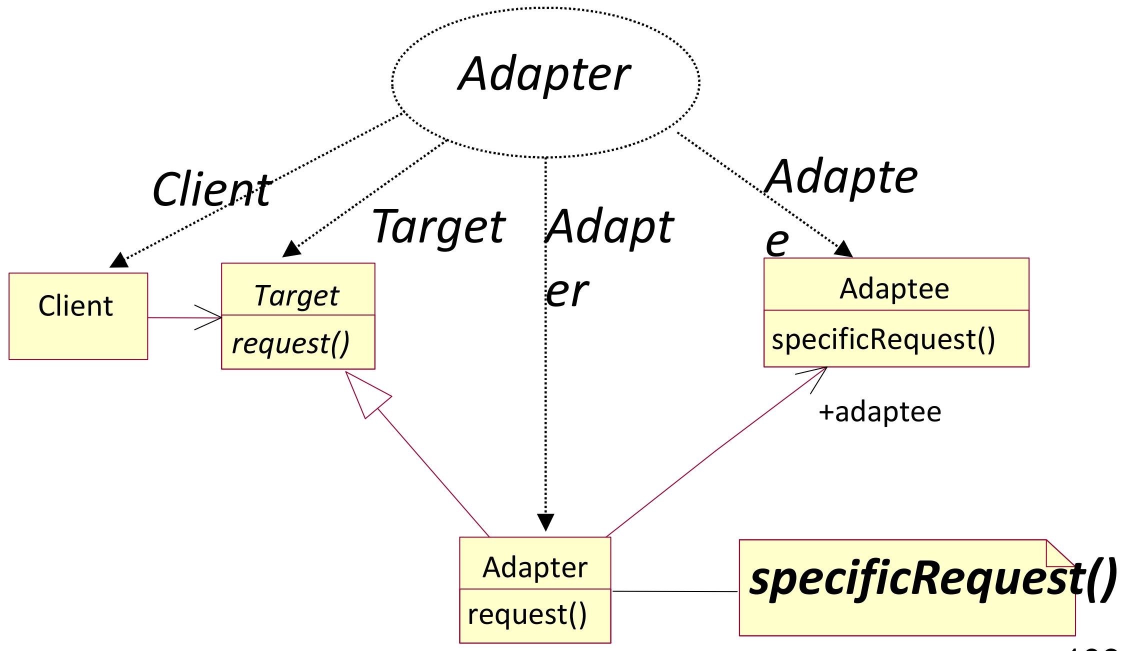
Adapter (TextShape) adapts the interface of Adaptee to the Target interface.

Adapter (IV)

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



Adapter (V)

Colaborations

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

Consequences

Class and object adapters have different trade-offs. A class adapter

1. 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.
2. lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
3. introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

1. 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.
2. makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Adapter - Implementare

Implementation

Consider a TreeDisplay widget that can display tree structures graphically. If this were a special-purpose widget for use in just one application, then we might require the objects that it displays to have a specific interface; that is, all must descend from a Tree abstract class. But if we wanted to make TreeDisplay more reusable (say we wanted to make it part of a toolkit of useful widgets), then that requirement would be unreasonable. Applications will define their own classes for tree structures. They shouldn't be forced to use our Tree abstract class. Different tree structures will have different interfaces.

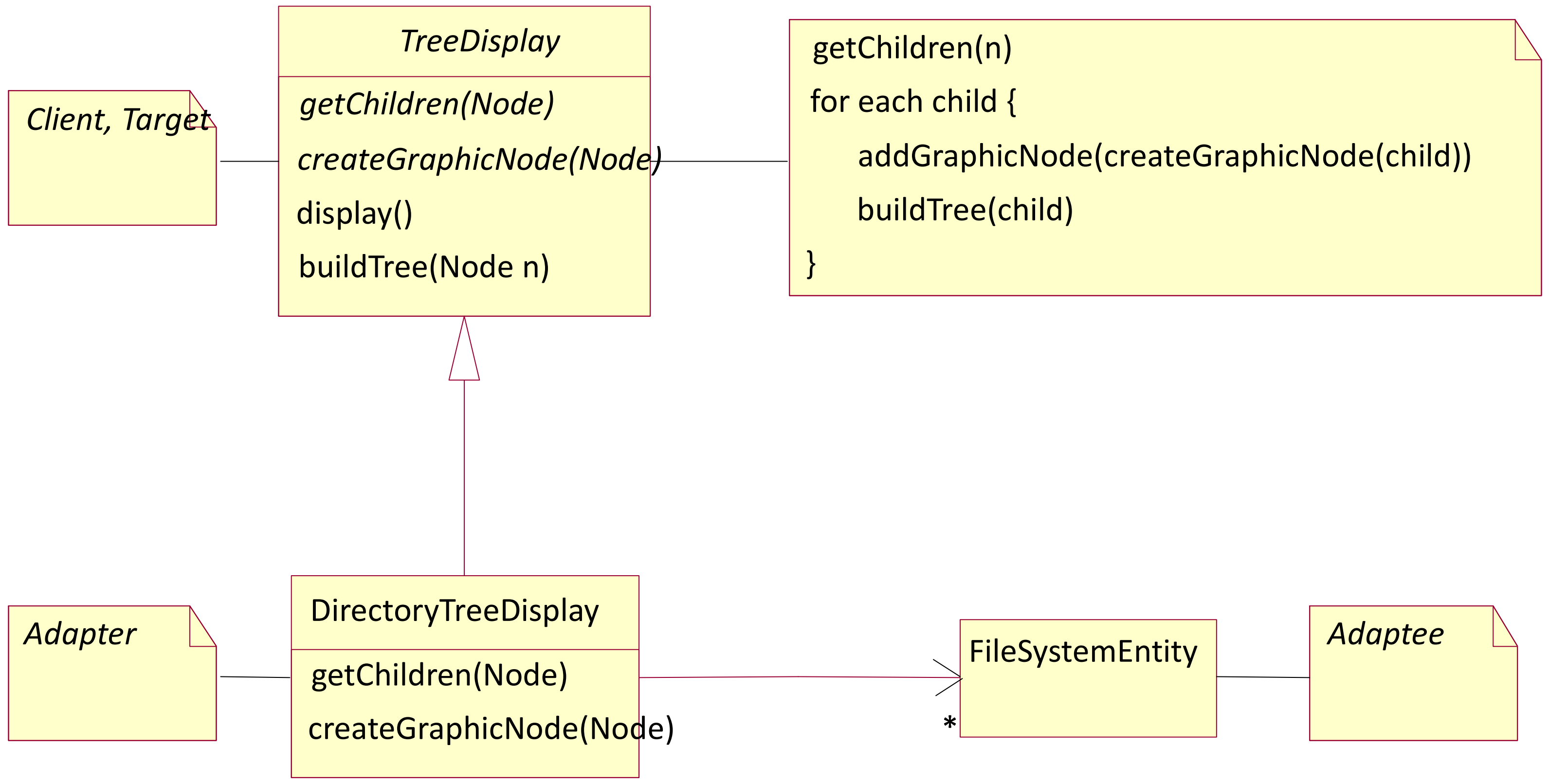
The first step, which is common to all three of the implementations discussed here, is to find a "narrow" interface for Adaptee, that is, the smallest subset of operations that lets us do the adaptation. A narrow interface consisting of only a couple of operations is easier to adapt than an interface with dozens of operations. For TreeDisplay, the adaptee is any hierarchical structure. A minimalist interface might include two operations, one that defines how to present a node in the hierarchical structure graphically, and another that retrieves the node's children.

The narrow interface leads to two implementation approaches:

Adapter (VI)

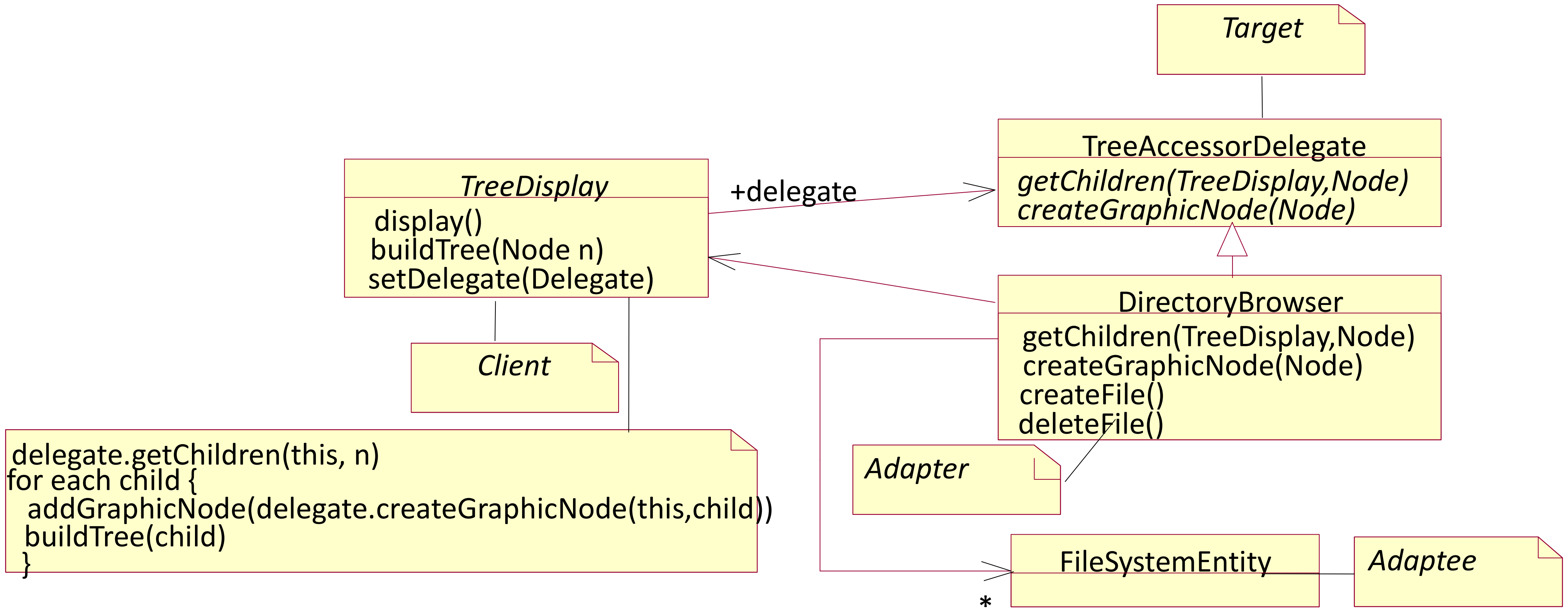
Using abstract operations. Define corresponding abstract operations for the narrow Adaptee interface in the TreeDisplay class. Subclasses must implement the abstract operations and adapt the hierarchically structured object. For example, a DirectoryTreeDisplay subclass will implement these operations by accessing the directory structure.

DirectoryTreeDisplay specializes the narrow interface so that it can display directory structures made up of FileSystemEntity objects.



Adapter (VII)

- *Using delegate objects.* In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a **delegate** object. TreeDisplay can use a different adaptation strategy by substituting a different delegate. For example, suppose there exists a DirectoryBrowser that uses a TreeDisplay. DirectoryBrowser might make a good delegate for adapting TreeDisplay to the hierarchical directory structure.



```
delegate.getChildren(this, n)
for each child {
  addGraphicNode(delegate.createGraphicNode(this,child))
  buildTree(child)
}
```

Iterator (I)

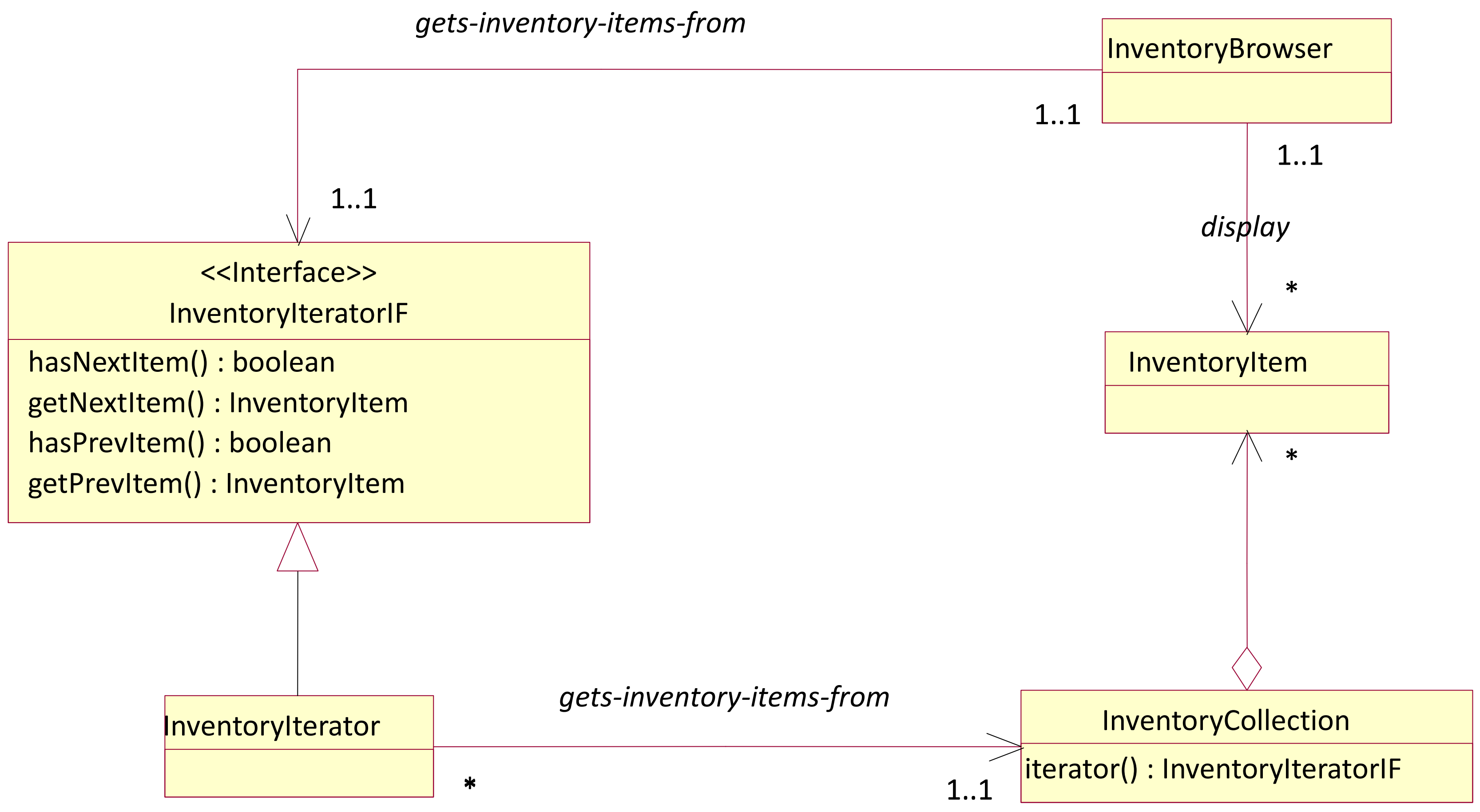
Intent

The Iterator pattern defines an interface that declares methods for sequentially accessing the objects in a collection without exposing its underlying representation . A class that accesses a collection only through such an interface is independent of the class that implements the interface and the class of the collection.

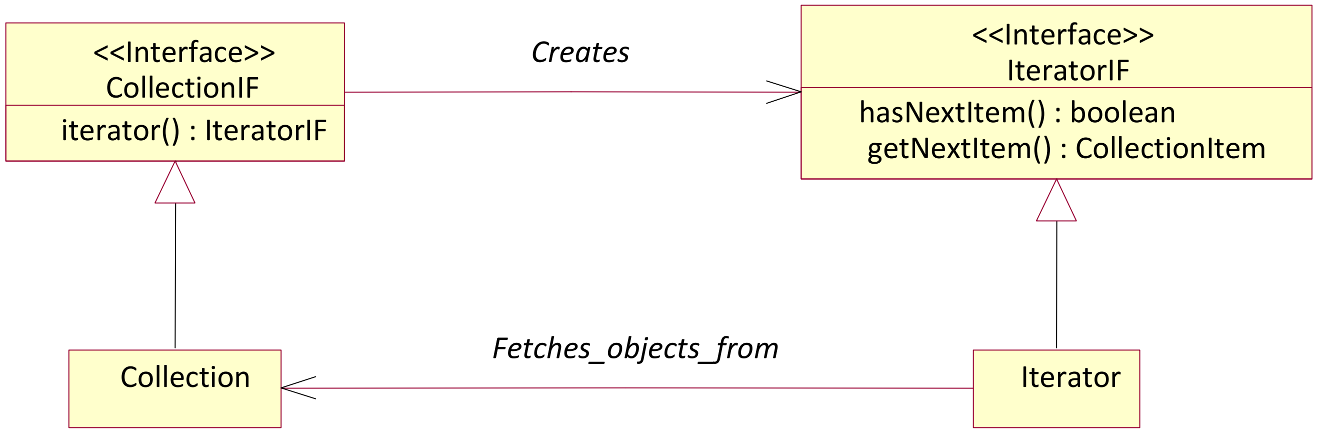
Motivation

- Suppose you are writing classes to browse inventory in a warehouse. There will be a user interface that allows a user to see the description, quantity on hand, location, and other information about each inventory item.
- The inventory browsing classes will be part of a customizable application. For this reason, they must be independent of the actual class that provides collections of inventory items. To provide this independence, you design an interface to allow the user interface to sequentially access a collection of inventory items without having to be aware of the actual collection class being used.
- An instance of the InventoryBrowser class is asked to display InventoryItem objects in the collection encapsulated by an InventoryCollection object. The InventoryBrowser object does not directly access the InventoryCollection object. Instead, it is given an object that implements the InventoryIteratorIF interface. The InventoryIteratorIF interface defines methods to allow an object to sequentially fetch the contents of a collection of InventoryItem objects.

Iterator (II)


gets-inventory-items-from
InventoryBrowser
1..1
1..1
1..1
display
*
<<Interface>>
InventoryIteratorIF
hasNextItem() : boolean
getNextItem() : InventoryItem
hasPrevItem() : boolean
getPrevItem() : InventoryItem
InventoryItem
*
InventoryIterator
gets-inventory-items-from
InventoryCollection
iterator() : InventoryIteratorIF
*
1..1

Iterator (III)



Structure

- **Collection.** A class in this role encapsulates a collection of objects or values.
- **IteratorIF.** An interface in this role defines methods to sequentially access the objects that are encapsulated by a Collection object.
- **Iterator.** A class in this role implements an IteratorIF interface. Its instances provide sequential access to the contents of the Collection object associated with Iterator object.
- **CollectionIF.** Collection classes normally take responsibility for creating their own iterator objects. It is convenient to have a consistent way to ask a Collection object to create an Iterator object for itself. To provide that consistency, all Collection classes implement a CollectionIF interface that declares a method for creating Iterator objects.

Iterator (IV)

- Applicability
- A class needs access to the contents of a collection without becoming dependent on the class that is used to implement the collection (without exposing its internal representation).
- A class needs a uniform way of accessing the contents of multiple collections (that is, to support polymorphic iteration).

Collaborations

A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

- It is possible to access a collection of objects without knowing the source of the objects.
- By using multiple iterator objects, it is simple to have and manage multiple traversals at the same time.
- A collection class may provide different kinds of iterator objects to traverse the collection in different ways. For example, a collection class that maintains an association between key objects and value objects may have a different method for creating iterators that traverse just the key objects and for creating iterators that traverse just the value objects.

Iterator - Implementation

Implementation

1. *Who controls the iteration?* The iterator or the client that uses the iterator? When the client controls the iteration, the iterator is called an **external iterator**, and when the iterator controls it, the iterator is an **internal iterator**. Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. In contrast, the client hands an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate. External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators.
2. *Who defines the traversal algorithm?* The iterator is not the only place where the traversal algorithm can be defined. The collection might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a **cursor**, since it merely points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.[3] If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates. On the other hand, the traversal algorithm might need to access the private variables of the aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

Iterator – Implementation (II)

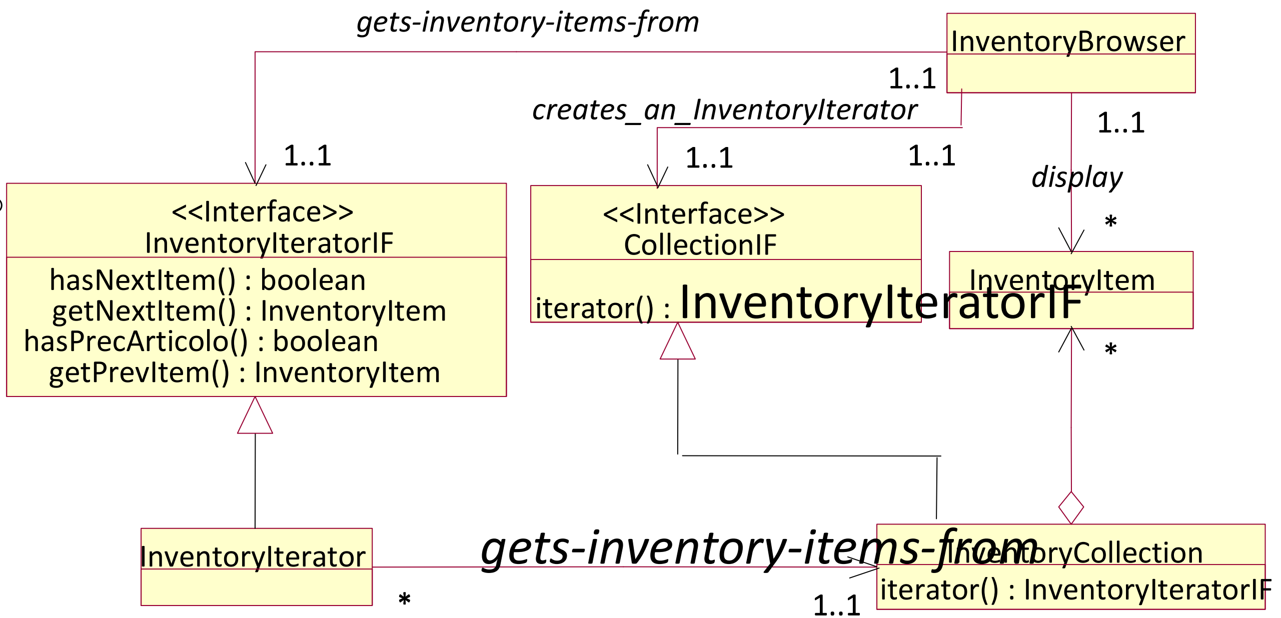
3. *How robust is the iterator?* It can be dangerous to modify an aggregate while you're traversing it. If elements are added or deleted from the aggregate, you might end up accessing an element twice or missing it completely. A simple solution is to copy the aggregate and traverse the copy, but that's too expensive to do in general. A **robust iterator** ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate. There are many ways to implement robust iterators. Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.
4. *Additional Iterator operations.* The minimal interface to Iterator consists of the operations First, Next, IsDone, and CurrentItem. Some additional operations might prove useful. For example, ordered aggregates can have a Previous operation that positions the iterator to the previous element. A SkipTo operation is useful for sorted or indexed collections. SkipTo positions the iterator to an object matching specific criteria.
5. *Iterators may have privileged access.* An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled. However, such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend. To avoid this problem, the Iterator class can include protected operations for accessing important but publicly unavailable members of the aggregate. Iterator subclasses (and *only* Iterator subclasses) may use these protected operations to gain privileged access to the aggregate.

Iterator - Implementation (III)

6. *Iterators for composites.* External iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates. Therefore an external iterator has to store a path through the Composite to keep track of the current object. Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack. If the nodes in a Composite have an interface for moving from a node to its siblings, parents, and children, then a cursor-based iterator may offer a better alternative. The cursor only needs to keep track of the current node; it can rely on the node interface to traverse the Composite. Composites often need to be traversed in more than one way. Preorder, postorder, inorder, and breadth-first traversals are common. You can support each kind of traversal with a different class of iterator.
7. *Null iterators.* A **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is *always* done with traversal; that is, its IsDone operation always evaluates to true. NullIterator can make traversing tree-structured aggregates (like Composites) easier. At each point in the traversal, we ask the current element for an iterator for its children. Aggregate elements return a concrete iterator as usual. But leaf elements return an instance of NullIterator. That lets us implement traversal over the entire structure in a uniform way.

Iterator – Code Sample

```
 public interface InventoryIteratorIF {
      public boolean hasNextInventoryItem() ;
      public InventoryItem getNextInventoryItem() ;
      public boolean hasPrevInventoryItem() ;
      public InventoryItem getPrevInventoryItem() ;
 }  // interface InventoryIterator
/**
 * Istanze di questa classe rappresenta articoli
   nelle scorte
 */
public class InventoryItem {
    //...
} // class InventoryItem
public class InventoryCollection {
 ...
      public InventoryIteratorIF iterator() {
           return new InventoryIterator();
      } // iterator()
   private class InventoryIterator implements
    InventoryIteratorIF {
           public boolean hasNextInventoryItem() {
            ...
           } // hasNextInventoryItem()
           public InventoryItem
    getNextInventoryItem() {
            ...
           } // getNextInventoryItem()
           public boolean hasPrevInvento
            ...
           } // hasPrevInventoryItem()
           public InventoryItem
    getPrevInventoryItem() {
            ...
           } // getPrevInventoryItem()
   } // class InventoryIterator
   ...
} // class InventoryCollection
```


gets-inventory-items-from
InventoryBrowser
1..1
creates_an_InventoryIterator
1..1
1..1
1..1
1..1
display
*
<<Interface>>
InventoryIteratorIF
hasNextItem() : boolean
getNextItem() : InventoryItem
hasPrecArticolo() : boolean
getPrevItem() : InventoryItem
<<Interface>>
CollectionIF
iterator() : InventoryIteratorIF
InventoryItem
*
InventoryIterator
gets-inventory-items-from
*
1..1
InventoryCollection
iterator() : InventoryIteratorIF

Bridge (I)

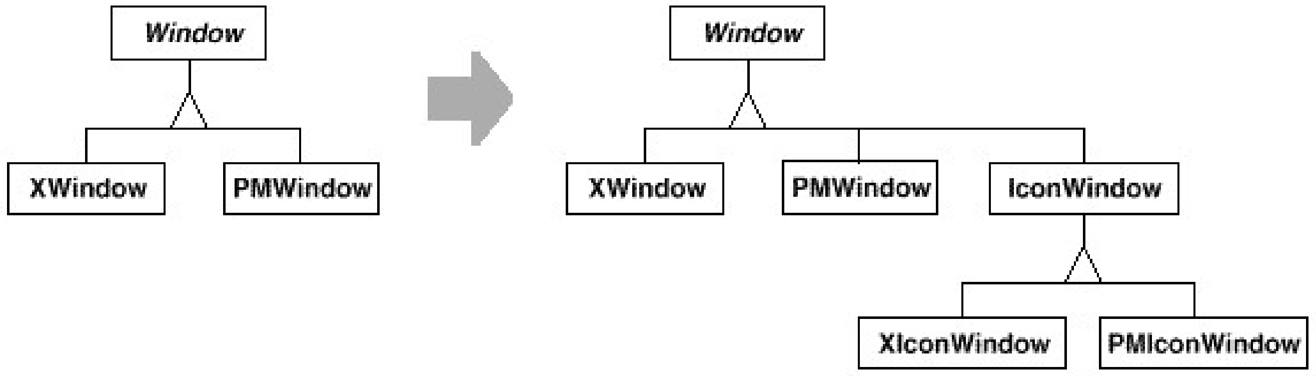
Intent

Decouple an abstraction from its implementation so that the two can vary independently. The Bridge pattern is useful when there is a hierarchy of abstractions and a corresponding hierarchy of implementations. Rather than combining the abstractions and implementations into many distinct classes, the Bridge pattern implements the abstractions and implementations as independent classes that can be combined dynamically.

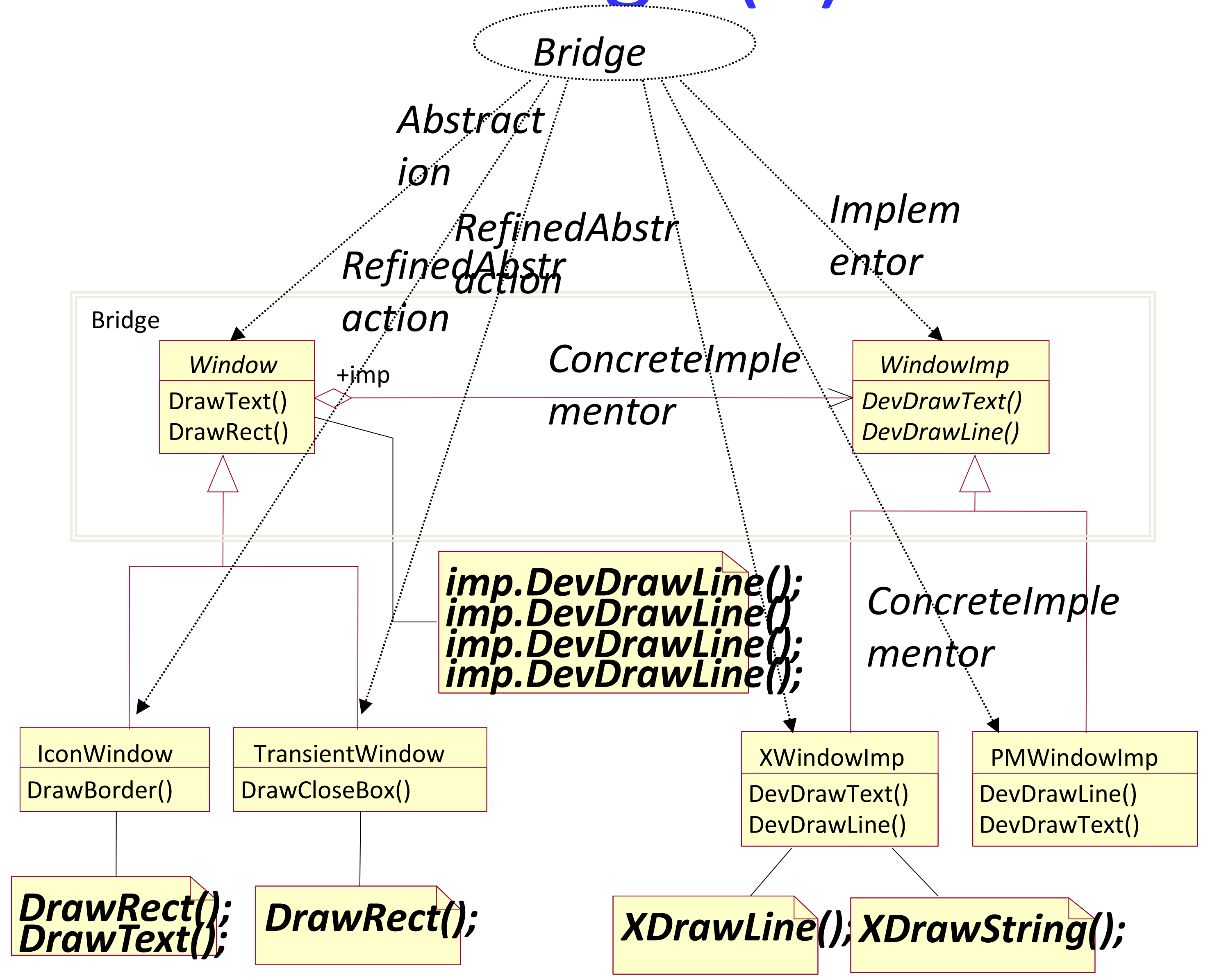
Motivation

Consider the implementation of a portable Window abstraction in a user interface toolkit. This abstraction should enable us to write applications that work on both the X Window System and IBM's Presentation Manager (PM), for example. Using inheritance, we could define an abstract class Window and subclasses XWindow and PMWindow that implement the Window interface for the different platforms. Drawbacks!

1. It makes client code platform-dependent.
2.

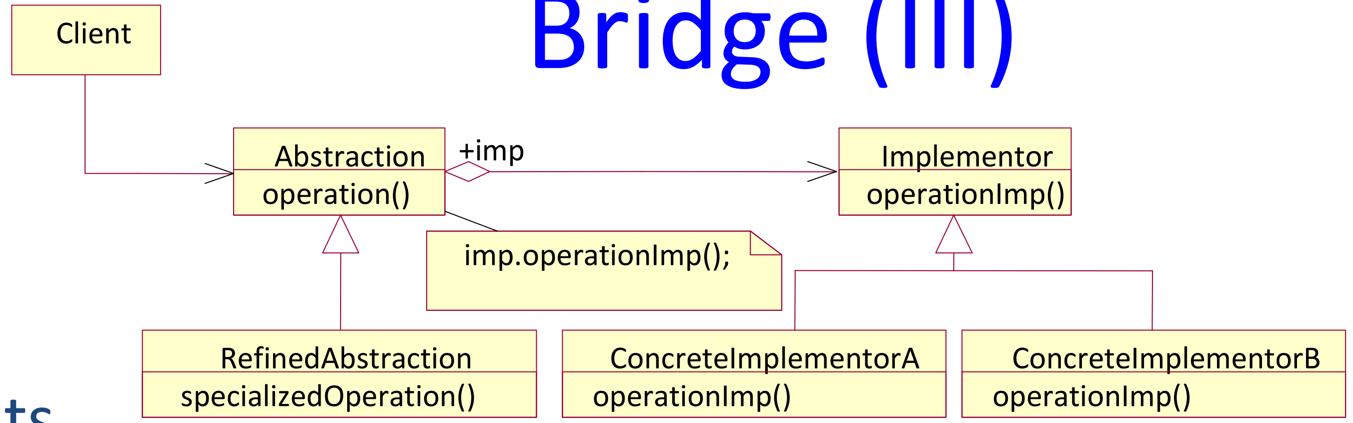


Bridge (II)



- Bridge puts the Window abstraction and its implementation in separate class hierarchies.

Bridge (III)



Participants

Abstraction (Window)

- defines the abstraction's interface.
- maintains a reference to an object of type Implementor.

RefinedAbstraction (IconWindow)

Extends the interface defined by Abstraction.

Implementor (WindowImp)

- defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

ConcreteImplementor (XWindowImp, PMWindowImp)

- implements the Implementor interface and defines its concrete implementation.

Bridge (IV)

Bridge

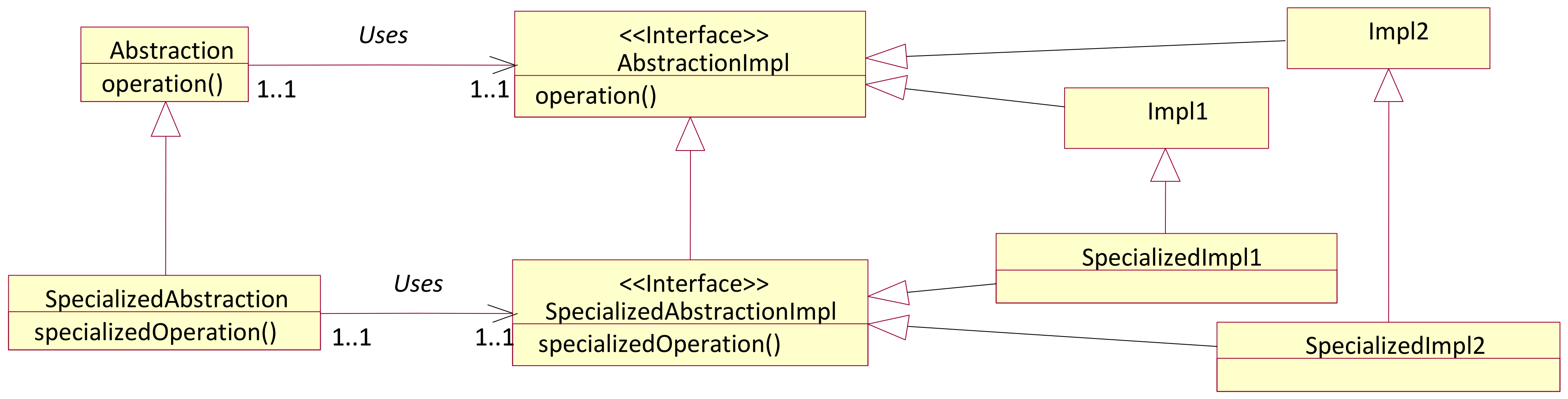
B R

Applicability

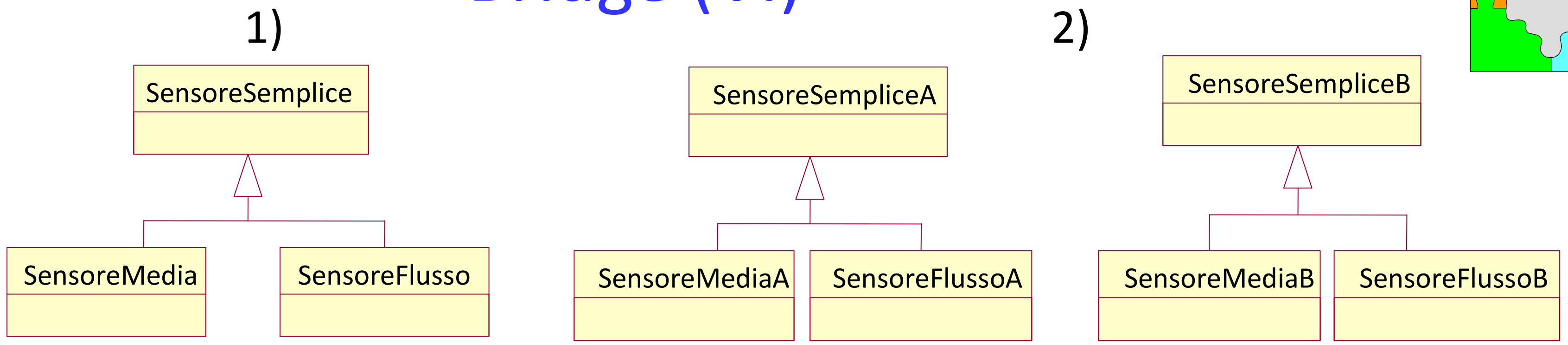
- When you combine hierarchies of abstractions and hierarchies of their implementations into a single class hierarchy, classes that use those classes become tied to a specific implementation of the abstraction. Changing the implementation used for an abstraction should not require changes to the classes that use the abstraction.
- You would like to reuse logic common to different implementations of an abstraction. The usual way to make logic reusable is to encapsulate it in a separate class.
- You would like to be able to create a new implementation of an abstraction without having to re-implement the common logic of the abstraction.
- You would like to be able to extend the common logic of an abstraction by writing one new class rather than writing a new class for each combination of the base abstraction and its implementation.
- When appropriate, multiple abstractions should be able to share the same implementation.

Bridge (V)

Extended Model:



Bridge (VI)



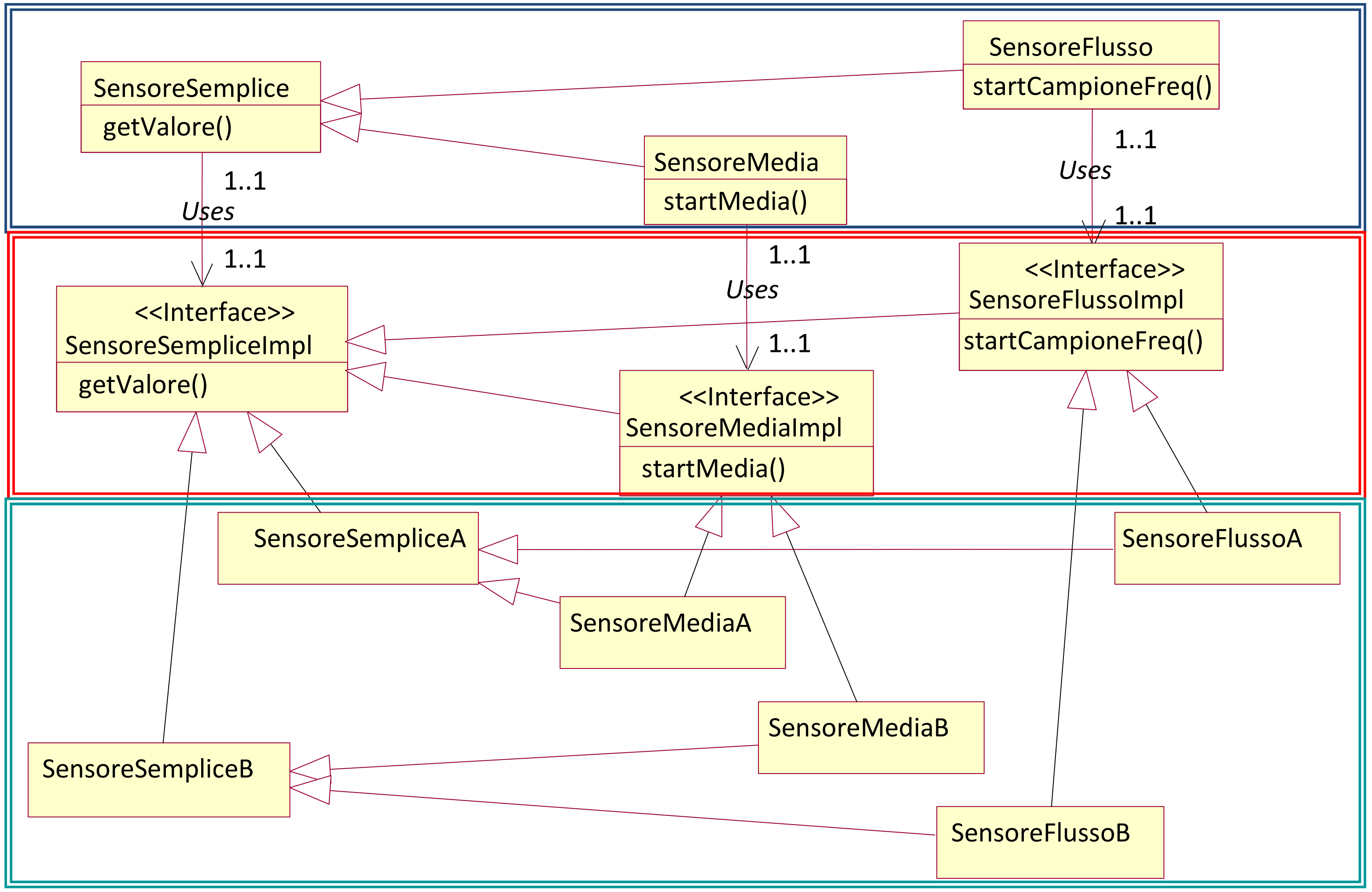
Exemple

Suppose you need to provide classes that access physical sensors for control applications. These are devices such as scales, speed-measuring devices, and location-sensing devices. What these devices have in common is that they perform a physical measurement and produce a number.A way that these devices differ is in the type of measurement that they produce:

- The scale produces one number based on a measurement at a single point in time.
- The speed-measuring device produces a single measurement that is an average over a period of time.
- The location-sensing device produces a stream of measurements.

- A difficulty in achieving the reuse is that the details of communicating with sensors from different manufacturers vary.
- A way to accomplish that is to add some indirection that shields a hierarchy of classes that support abstractions from classes that implement those abstractions. Have the abstraction classes access implementation classes through a hierarchy of implementation interfaces that parallels the abstraction hierarchy.

Bridge – An extended model (VII)



1. Ierahia sezorilor independenta de producatori

2. Ierarhie de interfete care permite independenta ierarhiei de senzori abstracti de ierarhia de senzori concreti

3. Ierarhia de senzori specifici pentru producatorii A si B

Bridge - Consequences

Consequences

- *Decoupling interface and implementation.* An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time. Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the Abstraction class and its clients. This property is essential for ensuring binary compatibility between different versions of a class library. This decoupling encourages layering that can lead to a better-structured system. The high-level part of a system only has to know about Abstraction and Implementor.
- *Improved extensibility.* You can extend the Abstraction and Implementor hierarchies independently.
- *Hiding implementation details from clients.* You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).

Implementation

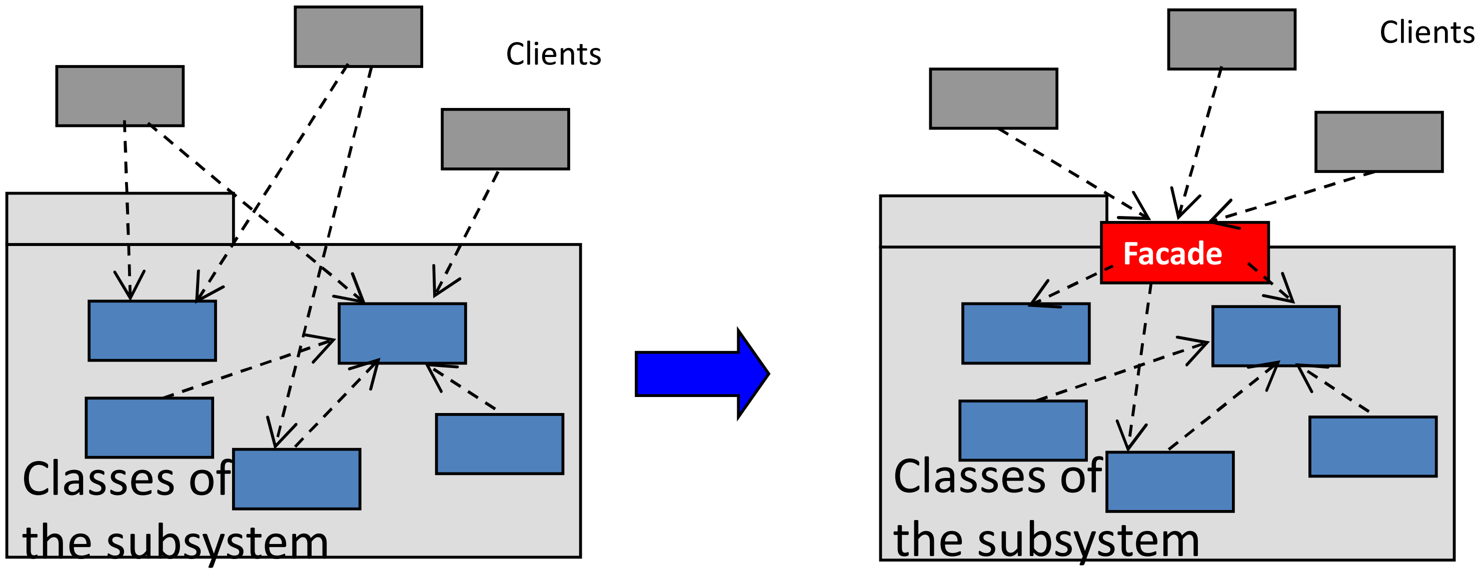
- One issue that always must be decided when implementing the Bridge pattern is how to create implementation objects for each abstraction object. The most basic decision to make is whether abstraction objects will create their own implementation objects or delegate the creation of their implementation objects to another object.
- Having the abstraction objects delegate the creation of implementation objects is usually the best choice. It preserves the independence of the abstraction and implementation classes. If abstraction classes are designed to delegate the creation of implementation objects, then the design usually uses the Abstract Factory pattern to create the implementation objects.

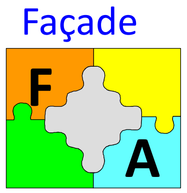
Bridge - Implementation

```
// Instances of this class are used to represent sensors only.
public class SimpleSensor {
     // object which implements specific operations of the effective sensor.
  private SimpleSensorImpl impl;
   /*   This constructor is called by a FactoryMethod object thst is found in the same
          package with the class SimpleSensor and the classi which implement it.
        @param impl is the object that implements the specific operations of the
          effective sensor.
    */
   SimpleSensor(SimpleSensorImpl impl) {
       this.impl = impl;
   } // constructor(SimpleSensorImpl)
  /* With this method the subclasses of this class find the object to implement.
    */
   protected SimpleSensorImpl getImpl() {
       return impl;
   } // getImpl()
//...
   /*     Return of the measurement value.
   */
   public int getValue() throws SensorException {
       return impl.getValue();
   } // getValue()
} // class SimpleSensor

 /*  This interface is implemented by all objects that implement the operations of
          the objects SimpleSensor.
 */
interface SimpleSensorImpl {
     public int getValue() throws SensorException;
} // interface SimpleSensorImpl

/*  the class implements the operations of the SimpleSensor for the provider A.
 */
class SimpleSensorA implements SimpleSensorImpl {
    public int getValue() throws SensorException {
        int valore;
        //...
        return valore;
    } // getValue()
} // class SimpleSensorA

/* the class implements the operations of the SimpleSensor for the provider B.
 */
class SimpleSensorB implemenSimpleSensorImpl {
    public int getValue() throws SensorException {
        int valore;
        //...
        return valore;
    } // getValue()
} // class SimpleSensorB
ts
```

```
/*  The instances of this class are used to represent any sensor that return average
       values of measurements for a period.
 */
public class MediaSensor extends SimpleSensor {
   /*
       This constructor is called by a FactoryMethod object thst is found in the same
        package with the class MediaSensor and the classi which implement it.
       @param impl is the object that implements the specific operations of the
        effective sensor.
    */
   MediaSensor(MediaSensorImpl impl) {
      super(impl);
   } // constructor(MediaSensorImpl)

   //...

   /*  Return of the measurement value
    @exception SensorException
   */
   public void startMedia() throws SensorException {
      ((MediaSensorImpl)getImpl()).startMedia();
   } // startMedia()
} // class MediaSensor

/* This interface is implemented by all objects that implement the operations of
         the objects MediaSensor.
interface MediaSensorImpl extends SimpleSensorImpl {
    public void startMedia() throws SensorException;
} // interface MediaSensorImpl

/* The class implements the operations of the MediaSensor for the provider A.
 */
class MediaSensorA extends SimpleSensorA implements MediaSensorImpl {
   public void startMedia() throws SensorException {
       //...
    } // startMedia()
} // class MediaSensorA

/* the class implements the operations of the MediaSensor for the provider B.
 */
class MediaSensorB extends SimpleSensorB implements MediaSensorImpl {
   public void startMedia() throws SensorException {
       //...
    } // startMedia()
} // class MediaSensorB
```

Façade (I)





Intent

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. The Façade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.

Motivation

Consider the organization of classes to support the creation and sending of email messages.

1. A MessageBody class whose instances will contain message bodies.
2. An Attachment class whose instances will contain message attachments that can be attached to a MessageBody object.
3. A MessageHeader class whose instances will contain header information (to, from, subject, etc.) for an email message.

Façade (II)

Motivation (cont.d)

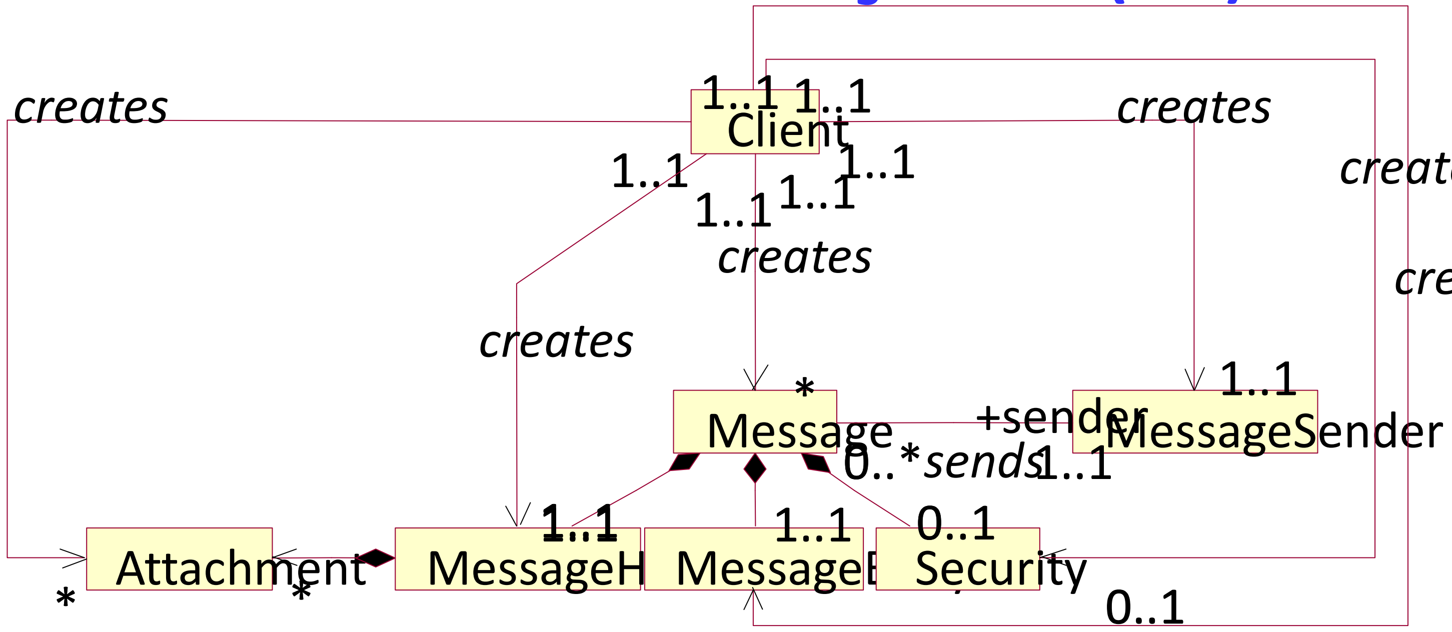
4. A Message class whose instances will tie together a MessageHeader object and a MessageBody object.
5. A Security class whose instances can be used to add a digital signature to a message.
6. A MessageSender class whose instances are responsible for sending Message objects to a server that is responsible for delivering the email to its destination or to another server.

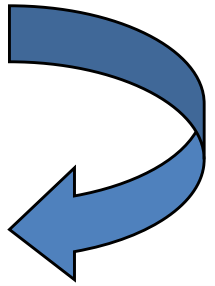
Working with these email classes adds complexity to a client class. To use these classes, a client must know of at least these six of them, the relationships between them, and the order in which it must create instances of the classes.

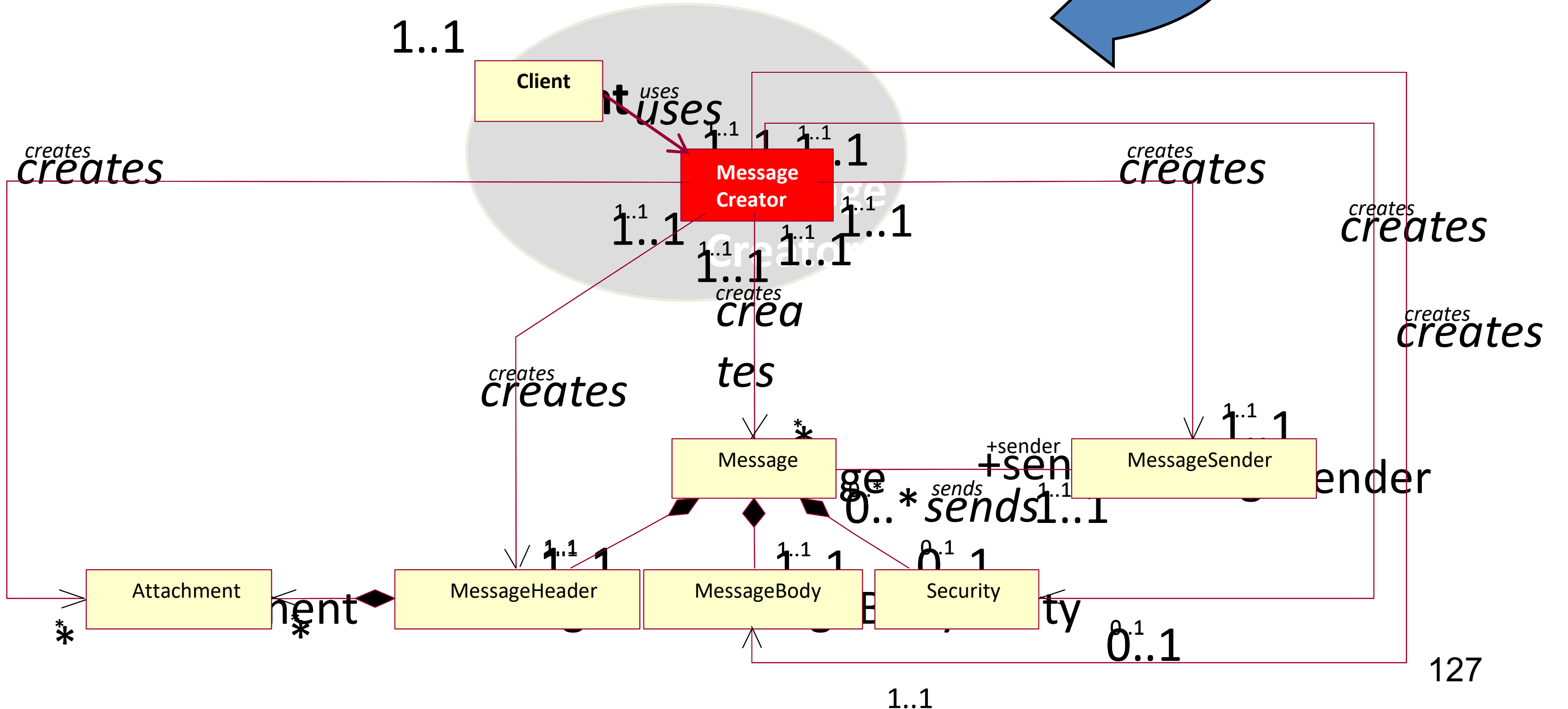
The Façade pattern is a way to shield clients of classes like these email classes from their complexity. It works by providing an additional reusable object that hides most of the complexity of working with the other classes from client classes.

In this new scheme, the portion of the Client class that was responsible for interacting with the email classes has been refactored into a separate reusable class. Client classes now need only be aware of the MessageCreator class. Furthermore, the internal logic of the MessageCreator class can shield client classes from having to create the parts of an email message in any particular order.

Façade (III)


Client
creates
creates
creates
creates
creates
creates
1..1
Message
+sender
MessageSender
0..*
sends
1..1
0..1
Attachment
MessageHeader
MessageBody
Security
*




Client
uses
Message Creator
creates
creates
creates
creates
creates
1..1
Message
+sender
MessageSender
0..*
sends
1..1
0..1
Attachment
MessageHeader
MessageBody
Security
*

Façade (IV)

Structure

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

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

Applicability

1. There are many dependencies between classes that implement an abstraction and their client classes. The dependencies add noticeable complexity to clients.
2. You want to simplify the client classes, because simpler classes result in fewer bugs. Simpler clients also mean that less work is required to reuse the classes that implement the abstraction.
3. You are designing classes to function in cleanly separated layers. You want to minimize the number of classes that are visible from one layer to the next.

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

Façade (V)

Consequencies

- 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.
- It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation

Consider the following issues when implementing a facade:

- *Reducing client-subsystem coupling.* The coupling between clients and the subsystem can be reduced even further by making Façade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Façade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.
- *Public versus private subsystem classes.* A subsystem is analogous to a class in that both have interfaces, and both encapsulate something - a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem. The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Façade class is part of the public interface, of course, but it's not the only part.

Façade (VI)

```
import java.util. *;
/*
        the instances of this class are used for the construction and sending of
        email messages. A message is composed from a header, a body and
        zero o several attachments. A body may be a  string or an object which
        implements an interface RichText.  An  attachment can contain any
        object.
 */
public class MessageCreator {
   // Constants of the message type  .
   public final static int MIME = 1;
   public final static int MAPI = 2;
   public final static int NOTES = 3;
   public final static int BANYAN = 4;
   private Hashtable headerFields = new Hashtable();
   private RichText messageBody;
   private Vector attachments = new Vector();
   private boolean signMessage;
/*   Costructors of a MessageCreator.    */
   public MessageCreator(String to, String from, String subject) {
      this(to, from , subject, inferMessageType(to));
   } // Constructor(String, String, String)
  public MessageCreator(String to, String from, String subject, int type) {
      headerFields.put("to", to);
      headerFields.put("from", from);
      headerFields.put("subject", subject);
      //...
   } // Constructor(String, String, String, int)
/*  Composes a MessageBody from a string. */
   public void setMessageBody(String messageBody) {
      setMessageBody(new RichTextString(messageBody));
   } // setMessageBody(String)

   /* Composes a MessageBody from a RichText.  */
   public void setMessageBody(RichText messageBody) {
      this.messageBody = messageBody;
   } // setMessageBody(RichText)

   /* Adds an attachment to a message. */
   public void addAttachment(Object attachment) {
      attachments.addElement(attachment);
   } // addAttachment(Object)
/*  Marks if the message is to be signed .   */
  public void setSignMessage(boolean signFlag) {
     signMessage = signFlag;
  } // setSignMessage(boolean)

  /*  Assigns a value to a HeaderField  */
  public void setHeaderField(String name, String value) {
     headerFields.put(name.toLowerCase(), value);
  } // setHeaderField(String, String)
  /* Sends the message.  */
  public void send() {
     MessageBody body = new MessageBody(messageBody);
     for (int i = 0; i < attachments.size(); i++) {
        body.addAttachment(new Attachment(attachments.elementAt(i)));
     } // for
     MessageHeader header = new MessageHeader(headerFields);
     Message msg = new Message(header, body);
     if (signMessage) {
        msg.setSecurity(createSecurity());
     } // if
     createMessageSender(msg);
  } // send()
  /* Infers the message type from the  destination address .  */
  private static int inferMessageType(String address) {
     int type = 0;
     //...
     return type;
  } // inferMessageType(String)
  /* Creates an object Security appropriate for the signature of the message.
      */
  private Security createSecurity() {
     Security s = null;
     //...
     return s;
  } // createSecurity()
  /* Creates an object MessageSender appropriate for  the  tyoe of the
      message to be sent.  */
  private void createMessageSender(Message msg) {
     //...
  } // createMessageSender(Message)
     //...
} // class MessageCreator
```

Decorator (Wrapper)

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. The Decorator pattern extends the functionality of an object in a way that is transparent to its clients, by implementing the same interface as the original class and delegating operations to the original class.

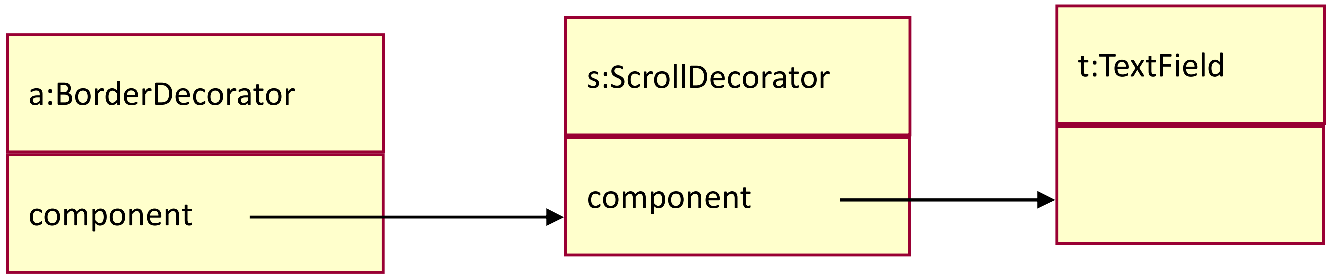
Motivation

- *Sometimes we want to add responsibilities to individual objects, not to an entire class.* A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component. One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.
- A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

Decorator (II)

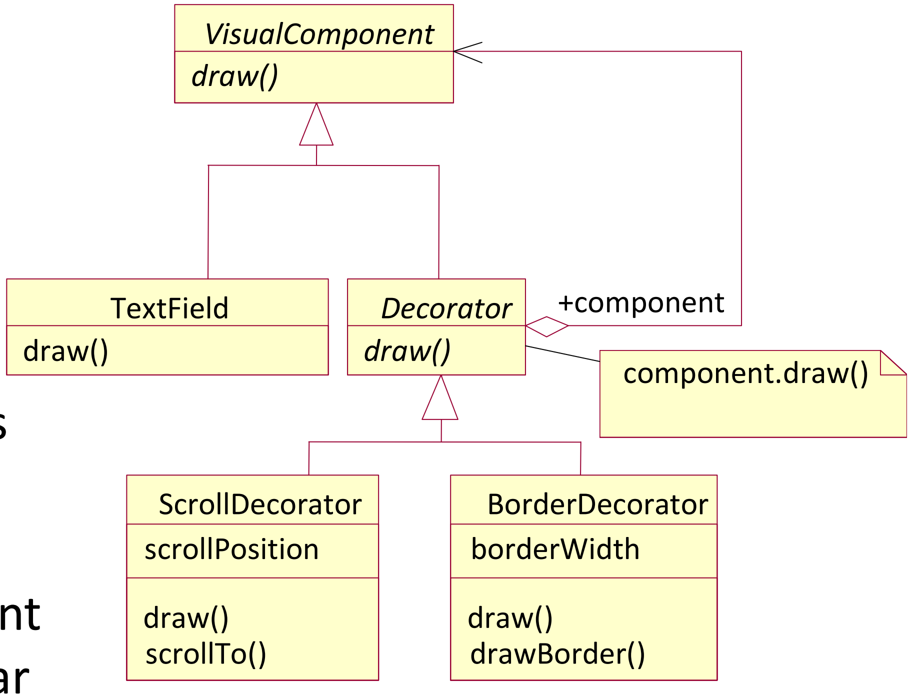
For example, suppose we have a TextView object that displays text in a window. TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them. Suppose we also want to add a thick black border around the TextView. We can use a BorderDecorator to add this as well. We simply compose the decorators with the TextView to produce the desired result.

The following object diagram shows how to compose a TextView object with BorderDecorator and ScrollDecorator objects to produce a bordered, scrollable text view:

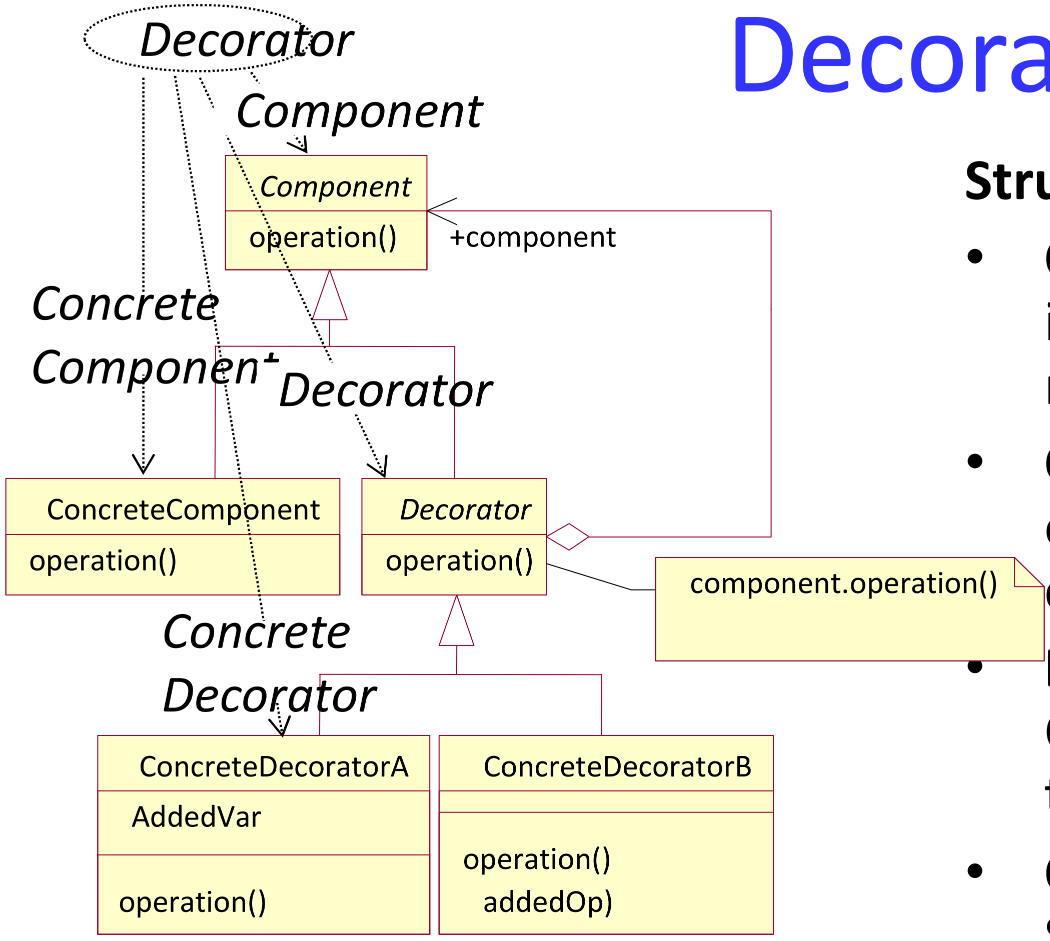


Decorator (III)

- The ScrollDecorator and BorderDecorator classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.
- VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface. Note how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation.
- Decorator subclasses are free to add operations for specific functionality. For example, ScrollDecorator's ScrollTo operation lets other objects scroll the interface *if* they know there happens to be a ScrollDecorator object in the interface. The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration.



Decorator (IV)



Structure

- **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.

Decorator (V)

Consequences

The Decorator pattern has at least two key benefits and two liabilities:

1. *More flexibility than static inheritance.* The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView). This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities. Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.
2. *Avoids feature-laden classes high up in the hierarchy.* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.

Decorator (VI)

3. *A decorator and its component aren't identical.* A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.
4. *Lots of little objects.* A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Decorator - Implementation

1. *Interface conformance.* A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class.
2. *Omitting the abstract Decorator class.* There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than designing a new one. In that case, you can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.
3. *Keeping Component classes lightweight.* To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.
4. *Changing the skin of an object versus changing its guts.* We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy pattern is a good example of a pattern for changing the guts.

Decorator - Code

Code

```
interface VisualComponent {
   public void draw();
}
abstract class Decorator implments VisualComponent {
    private VisualComponent wrappee;
    Decorator(VisualComponent wrapee) {
       this.wrapee = wrappee;
    }
    public void draw () {
       wrappe.draw();
    }
}
```

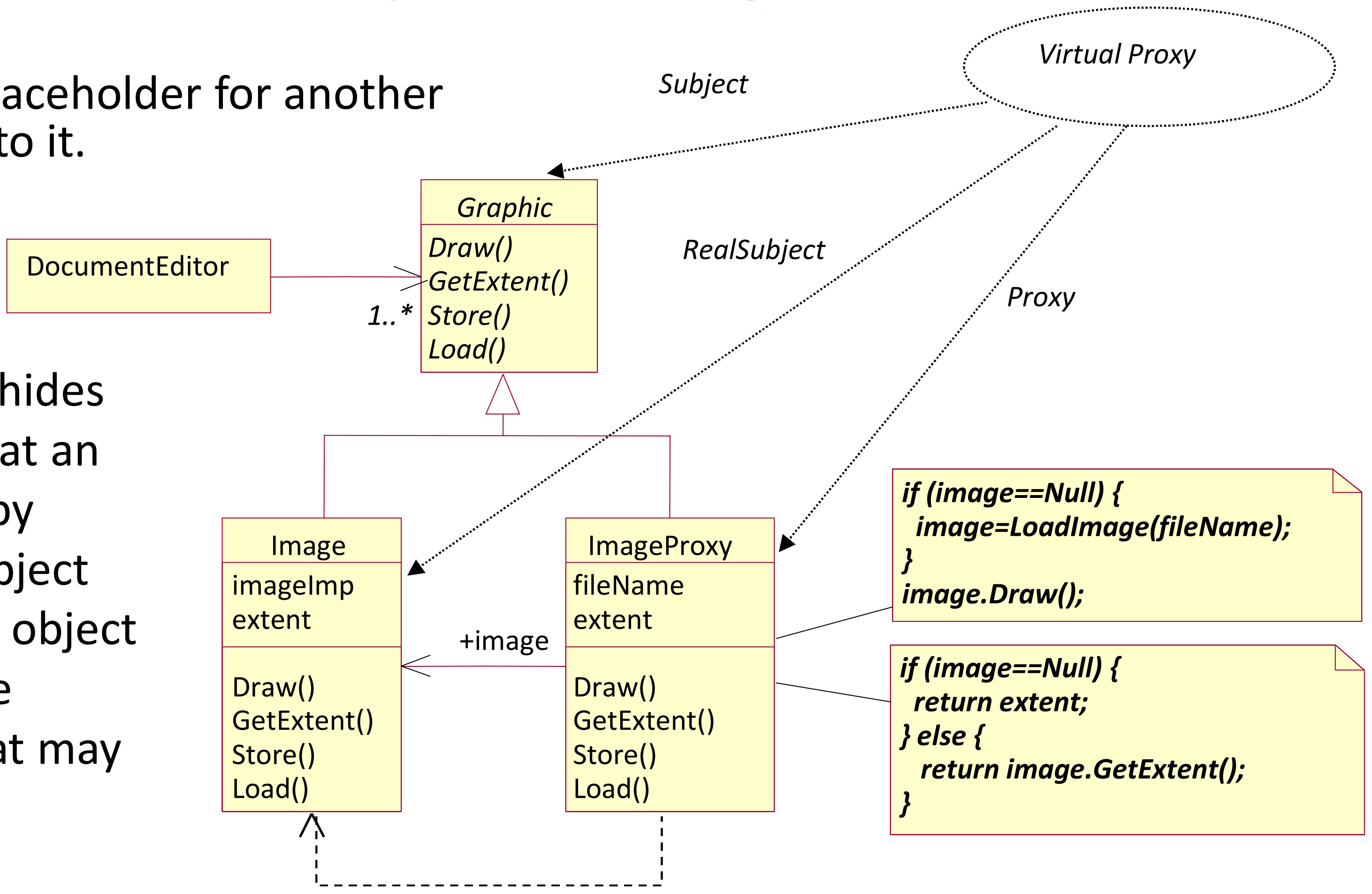
```
class ScrollDecorator extends Decorator {
   private Position scrollPosition;
   ScrollDecorator(VisualComponent wrappee, Position scrollPosition) {
       super(wrappee);
       this.scrollPosition = scrollPosition;
    }
    public void draw() {
       super.draw();
       scrollTo(scrollPosition);
    }
```

Virtual Proxy (Surrogate)

Intent

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

The Virtual Proxy pattern hides from its clients the fact that an object may not yet exist, by having them access the object indirectly through a proxy object that implements the same interface as the object that may not exist.



```
if (image==Null) {
  image=LoadImage(fileName);
}
image.Draw();
```

```
if (image==Null) {
  return extent;
} else {
  return image.GetExtent();
}
```

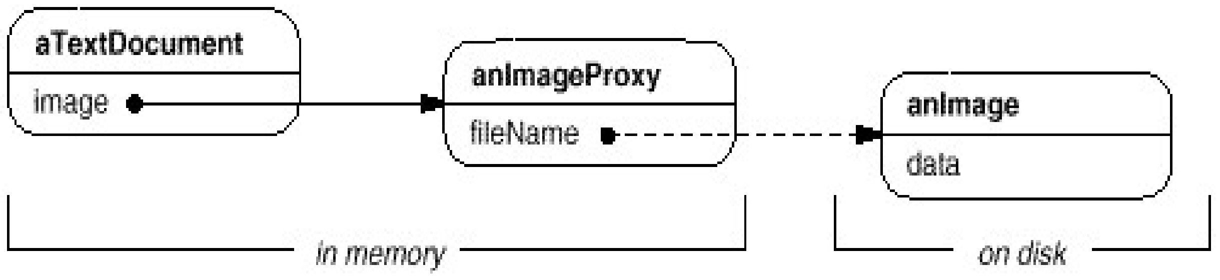
Exemplu

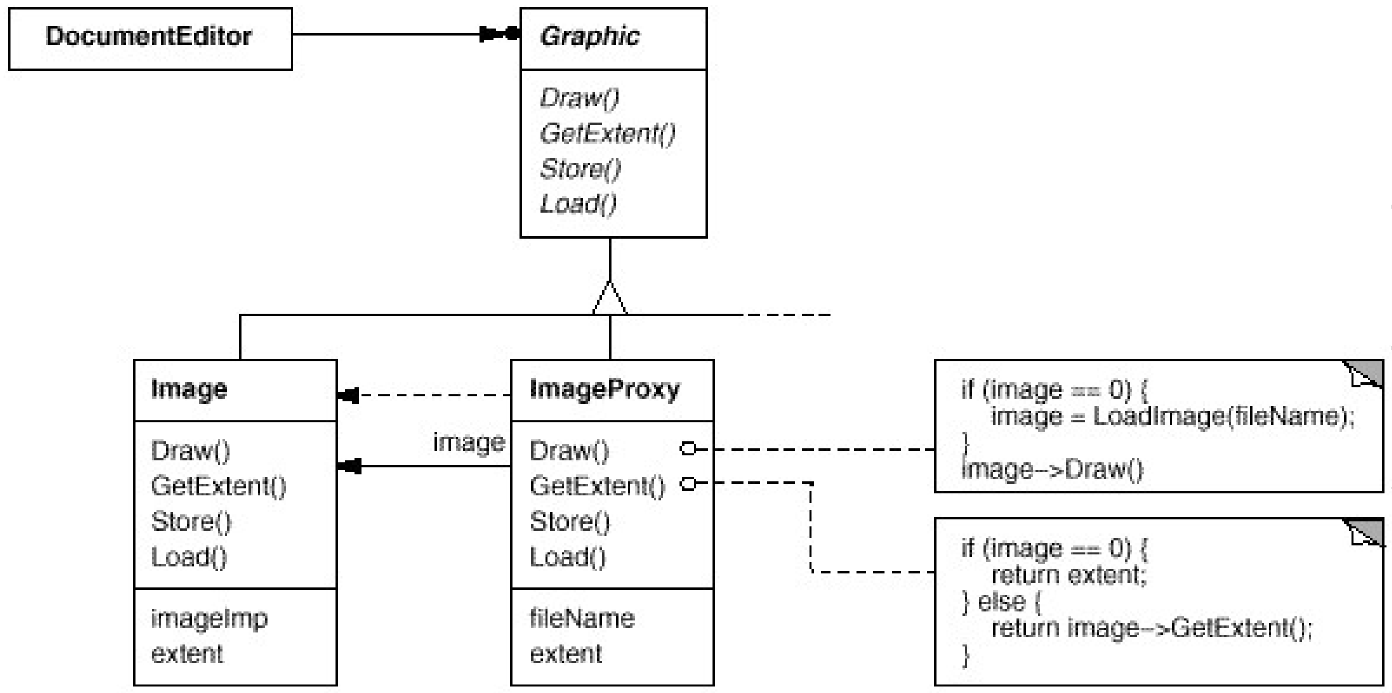
Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened.

These constraints would suggest creating each expensive object *on demand*, which in this case occurs when an image becomes visible. Meantime we can use another object, an image **proxy**, that acts as a stand-in for the real image.

ImageProxy

The proxy acts just like the image and takes care of instantiating it when it's required. The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.



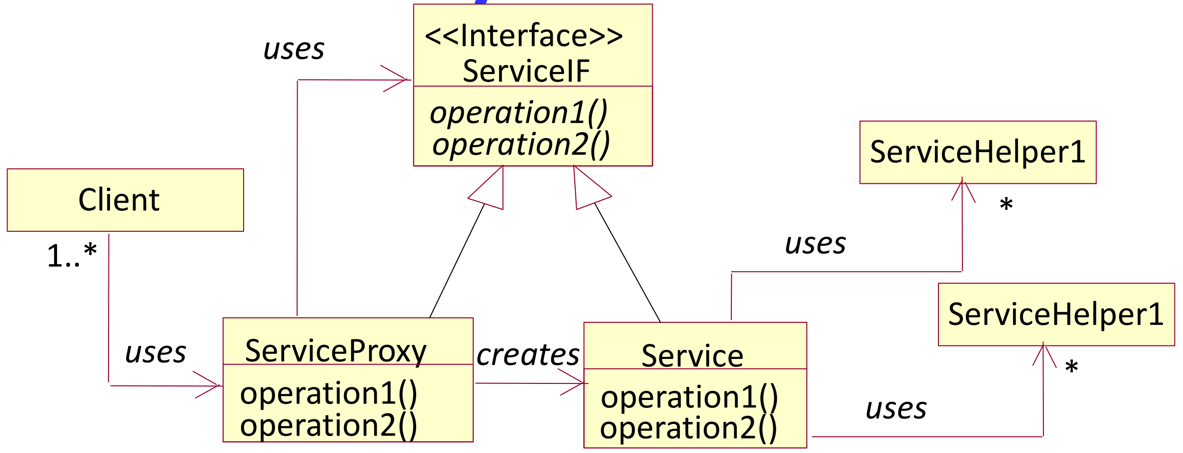


```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

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

ImageProxy is a class for images that are created on demand. It maintains the file name as a reference to the image on disk and the file name is passed as an argument to the constructor. ImageProxy also stores the bounding box of the image and a reference to the real Image instance. This reference won't be valid until the proxy instantiates the real image.

Virtual Proxy – Structure



Structure

- **Service.** A Service class supplies the top-level logic for a service that it provides. When an instance of it is created, ithis also creates the other objects that it needs (ServiceHelper…).
- **Client.** A class in this role uses the service provided by the Service class. Client classes never directly use a Service class. Instead, they use a ServiceProxy class that provides the functionality of the Service class.
- **ServiceProxy.** The purpose of the ServiceProxy class is to delay creating instances of the Service class until they are actually needed. It provides indirection between Client classes and a Service class. The indirection hides from Client objects the fact that when a ServiceProxy object is created, the corresponding Service object does not exist and the Service class may not even have been loaded. A ServiceProxy object is responsible for creating the corresponding Service object. A ServiceProxy object creates the corresponding Service object the first time it is asked to perform an operation that requires the existence of the Service object.
- **ServiceIF.** A ServiceProxy class creates an instance of the Service class through method calls that do not require static references to the Service class.

Virtual Proxy

```
public interface ServiceIF {
   public void operation1();
   public void operation2();
   //...
} // interface ServiceIF

import java.lang.reflect.Constructor;
public class ServiceProxy {
   private ServiceIF assistant = null;
   private String myParam;
   public ServiceProxy(String s) {
      myParam = s;
   } // constructor(String)

   private ServiceIF getService() {
      if (assistant == null) {
         try {
             Class clazz=Class.forName("Service");  // ** 1 **
            Constructor constructor;
            Class[] formalArgs = new Class [] { String.class };
            constructor = clazz.getConstructor(formalArgs); // ** 2 **

           Object[] actuals = new Object[] { myParam };
           assistant = (ServiceIF)constructor.newInstance(actuals); // ** 3 **
         } catch (Exception e) {
         } // try
         if (assistant == null) {   // ** 4 **
             throw new RuntimeException();
         } // if
      } // if
      return assistant;
   } // getService()

   //...
   public void operation1() {
      getService().operation1();
   } // operation1()

   public void operation2() {
      getService().operation2();
   } // operation2()
} // class ServiceProxy
```

1. Furnizeaza obiectul clasa care reprezinta clasa Service
2. Furnizeaza un obiect constructor pentru crearea obiectului Service.
3. Utilizeaza obiectul constructor.
4. Lanseaza o exceptie dupa esuarea tentativei de creare a unui obiect Service.

Virtual Proxy - Conclusions

Conclusions

- Classes accessed by the rest of a program exclusively through a virtual proxy are not loaded until they are needed.
- Objects accessed through a virtual proxy are not created until they are needed.
- Classes that use the proxy do not need to be aware of whether the Service class is loaded, of whether an instance of it exists, or that the class
- All classes other than the proxy class must access the services of the Service class indirectly through the proxy. This is critical. If just one class accesses the Service class directly, then the Service class will be loaded before it is needed. This is a quiet sort of bug. It generally affects performance but not function, so it is hard to track down.

Some comments on the structural patterns

Adapter and Bridge

Both patterns promotes flexibility by providing an indirect reference to another object. Both filter requests to this object through an interface different of its. But the intents of the two patterns are different.

Adapter is committed to resolve the incompatibility between two existing interface. No matter how interfaces are implemented or are generated as independent. It's just a way of keeping the possibility that two independent two classes can work together without having to reimplement any of them.

Bridge is a bridge between an abstraction and its many potential implementations. Provides an interface letting customers the ability to vary the classes that implements it. In this way facilitates adding new implementations.

Adapter and Bridge are therefore used in different stages of the life cycle of software development. An adapter is required at the end of the design when it is discovered incompatibility of two classes that must work together and offer a solution to avoid replicating code. In this case switching classes was not originally envisaged. Bridge instead promotes the idea that an abstraction will have multiple implementations and that all classes can be developed independently. Therefore Bridge will be applied prior to initiating design.

Facade and Adapter

Façade may seem an Adapter for a lot of objects, but it is not so because Façade defines a new interface while Adapter reuses an old one.

Decorator and Composite

Have similar structures as they rely on recursive composition to organize an unlimited number of objects. Although their intents are different, they are not complementary.

Decorator as used for adding new responsibilities to objects without sub-classifying them. Avoid subclasses explosion that would result when trying to cover all combinations of responsibilities in a static mode.

Composite instead engages in structuring classes so that related objects can be treated in a uniform manner and composed objects are treated as a whole.

1.5. Behavioral Patterns

1. Observer
2. Command
3. Strategy
4. Template Method

- The behavioral patterns are patterns that describe the ways objects and classes interact and divide responsibilities among themselves.
- They are used to organize, manage, and combine behavior.
- A behavioral pattern abstracts an action you want to take from the object or class that takes the action.
- By changing the object or class, you can change the algorithm used, the objects affected, or the behavior, while still retaining the same basic interface for *client* classes.

Behavioral Patterns

A behavioral pattern explains how objects interact. It describes how different objects and classes send messages to each other to make things happen and how the steps of a task are divided among different objects. Where creational patterns mostly describe a moment of time (the instant of creation), and structural patterns describe a more or less static structure, behavioral patterns describe a process or a flow.

Behavioral class patterns

Behavioral class patterns use inheritance, subclassing, and polymorphism to adjust the steps taken during a process. Behavioral class patterns focus on changing the exact algorithm used or task performed depending on circumstances.

Behavioral object patterns

Behavioral object patterns describe how different objects work together to accomplish a task. Behavioral object patterns accomplish tasks that would be difficult or impossible to accomplish with single objects. Furthermore, they generally make the entire flow simpler, more understandable, and more robust than the string-and-bailing-wire solutions that are built without a clear design in mind.

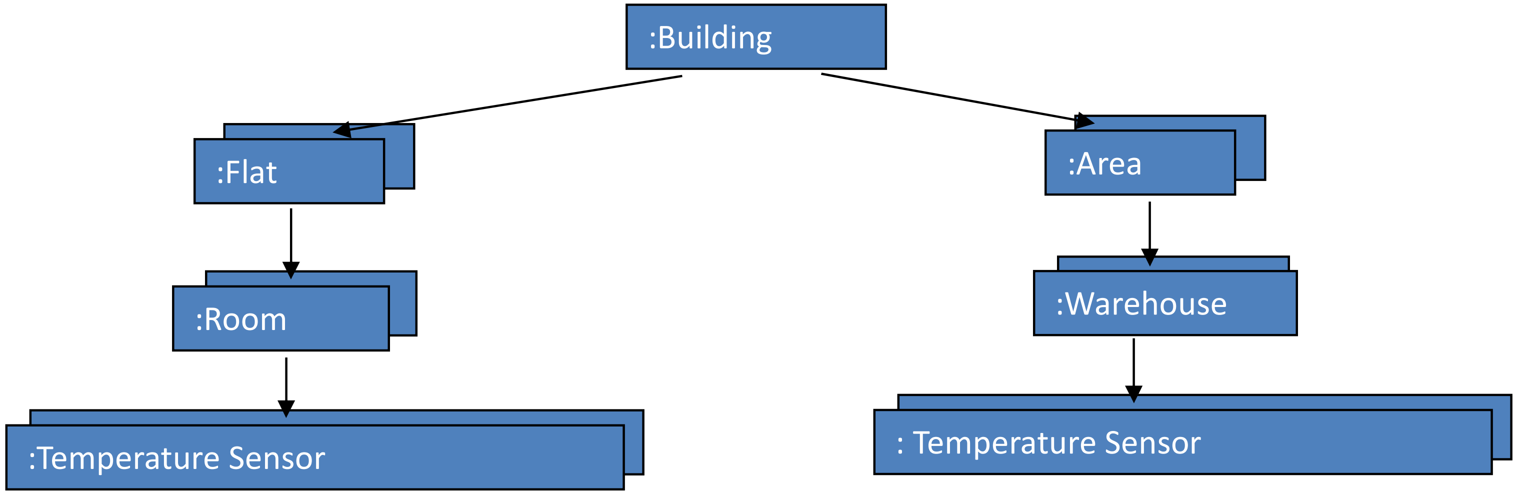
Chain of Responsibility - Intent

Intent

It avoids coupling between a sender and a recipient by allowing other objects to carry out the request. The request is sent to a chain of objects until one of them decides to carry it out.

Motivation

A security system consists of a number of sensors and a computer that monitors them. For this, the sensors transmit their status to the computer. It stores the status information, visualizes the current status and starts the alarm if an emergency is detected. Such a system must be scalable: it easily adapts to different environments. Each sensor will have its own object that represents the state of the sensor. For reasons of scalability, these objects are not stored in any environment, except when they are at the basic level of the hierarchical organization of the system components: apartment, room, refrigerator, cellar, etc. Each component of the organization will be represented by an object. In this way, the state of the sensor placed in a component can be interpreted according to the environment in which the sensor or component is found. A temperature sensor will behave in a certain way in a room and in another way in a refrigerator or cellar. Therefore it is not the sensor object that has to decide how to interpret its state. It will delegate the decision to an object from a higher level in the hierarchy that knows its context better. This object either decides what to do, or it will announce an object from a higher level.

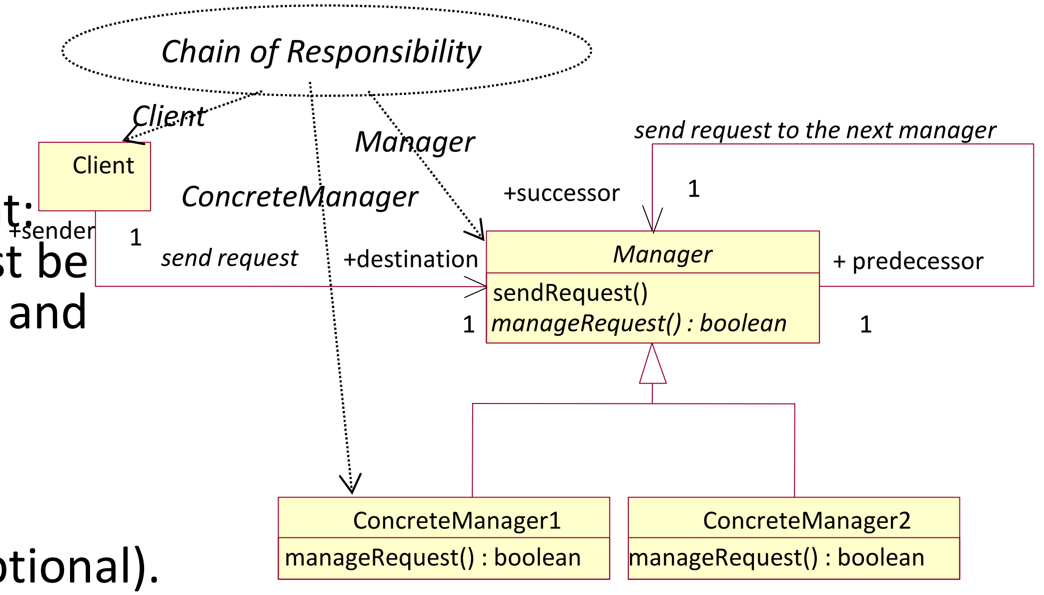


Chain of Responsibility - Structure

Aplicability

1. When several objects have to manage a request and the manager is not known in advance. It will be set dynamically.
2. If it is desired to produce a request for one or more objects without explicitly specifying the recipient.
3. The set of objects that can handle the request must be specified dynamically.

Structure

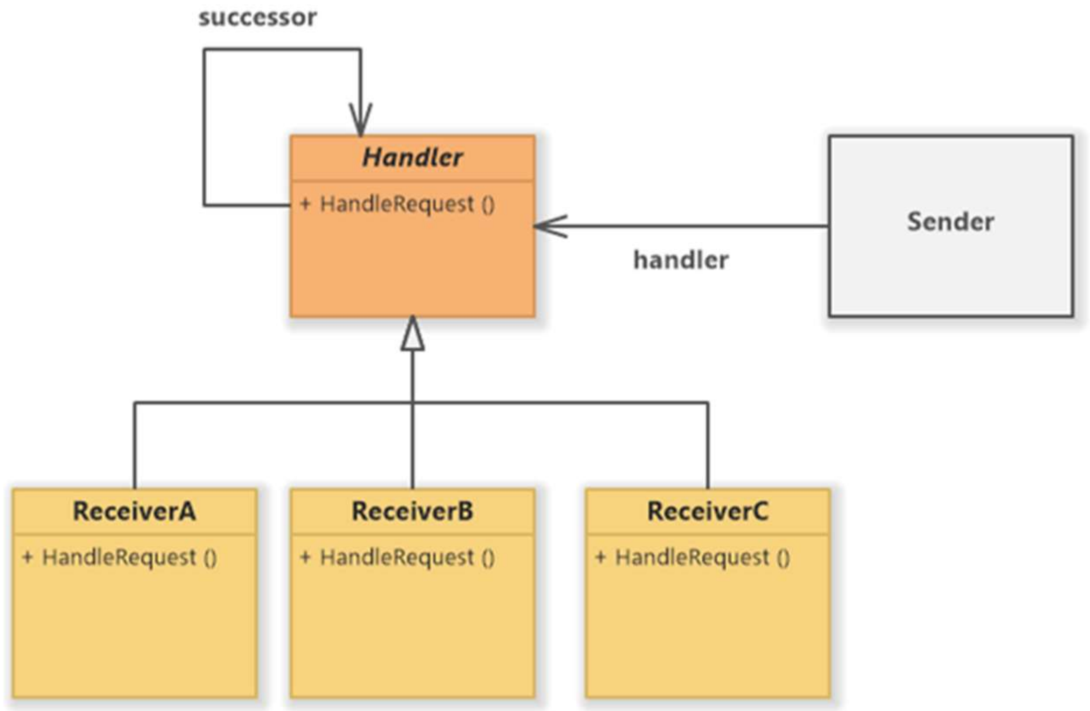


Manager - the superclass of all objects from the chain of objects that can solve the request.
Defines the interface for request management:
a) the manageRequest() method which must be redefined by all ConcreteManager subclasses and
b) the sendRequest() method that calls the method manageRequest() and if the result of the call is false, send the request to the successor in the chain.
- implements the link of the successors (optional).

ConcreteManager - Instances are objects in the object chain that can handle the request.
– manages the request for which it is responsible,
– may access successors,
– If ConcreteManager can solve the request, it solves it. Otherwise, it sends it to its successor.

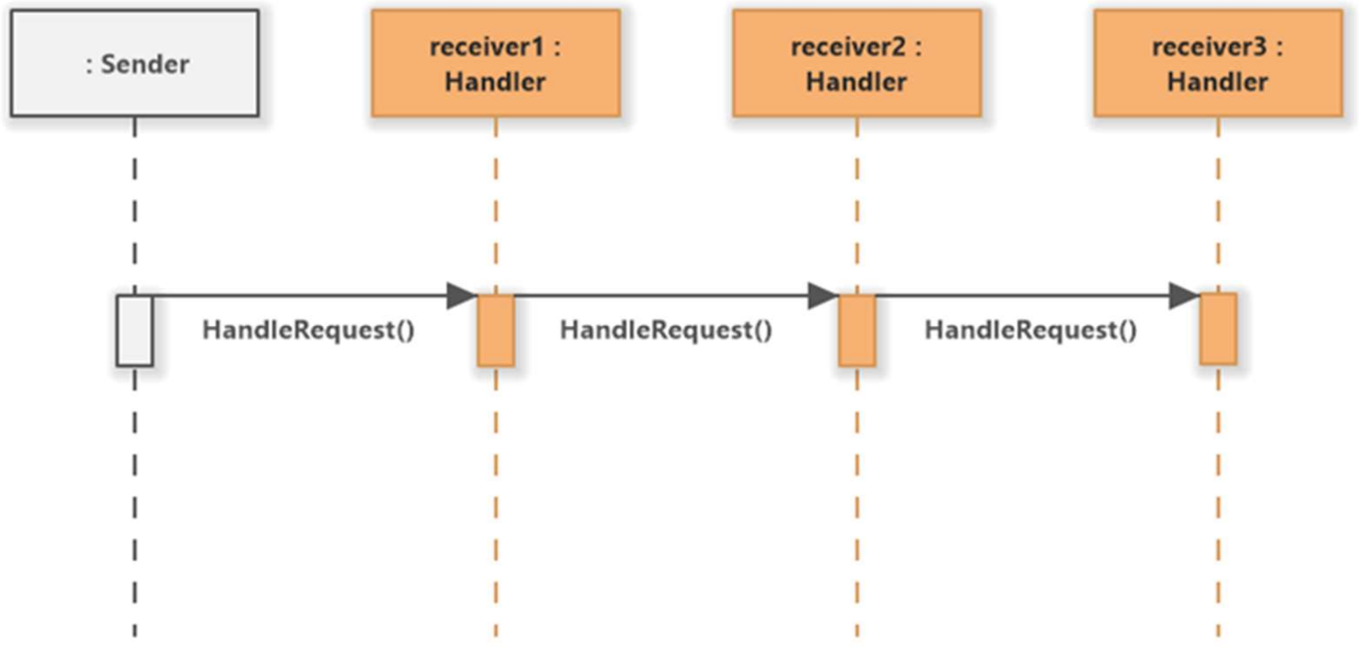
Client - instances send the request to the first object in a chain of objects that can handle the request. Use the sendRequest() method for sending.
– Initiates the request of a ConcreteManager object in the chain.

Chain of Responsibility

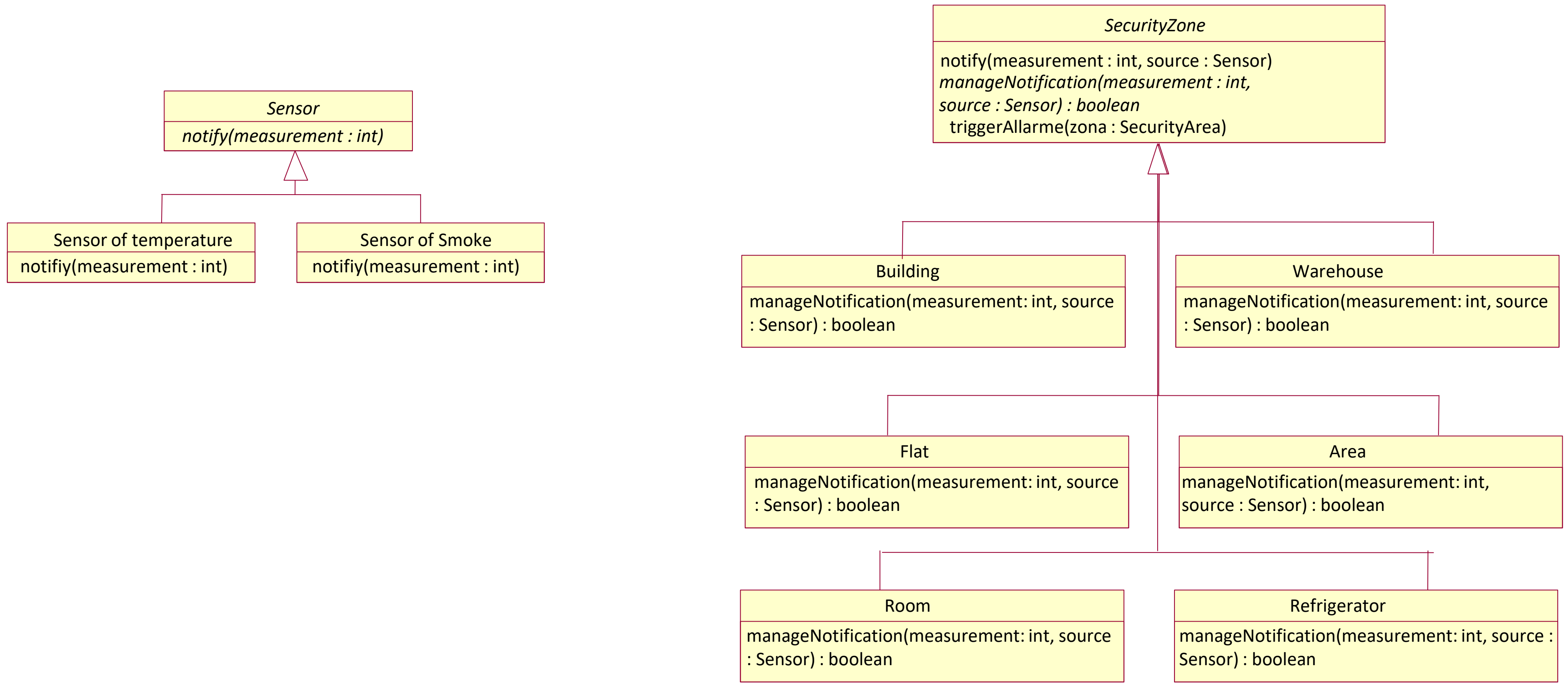


Structure

Collaborations



Chain of Responsibility - Conclusions


Sensor
notify(measurement : int)
Sensor of temperature
notifiy(measurement : int)
Sensor of Smoke
notifiy(measurement : int)
SecurityZone
notify(measurement : int, source : Sensor)
manageNotification(measurement : int, source : Sensor) : boolean
triggerAllarme(zona : SecurityArea)
Building
manageNotification(measurement: int, source : Sensor) : boolean
Warehouse
manageNotification(measurement: int, source : Sensor) : boolean
Flat
manageNotification(measurement: int, source : Sensor) : boolean
Area
manageNotification(measurement: int, source : Sensor) : boolean
Room
manageNotification(measurement: int, source : Sensor) : boolean
Refrigerator
manageNotification(measurement: int, source : Sensor) : boolean

Conclusions

1. The model reduces the coupling between the object that sends a request and the one that will manage it.
2. 2. It introduces a major flexibility in the decision of who manages the request, but does not guarantee that the request will be satisfied.

Mediator

Intent

The mediator pattern is used to define simplified communication between classes. It uses an object that encapsulates the way several objects interact. The object coordinates the modification of the state of other objects. In this way, the object promotes weak coupling and increases cohesion, avoiding that the objects interact directly with each other. That is why the interaction can be diversified, independent of the objects involved.

Motivation

Let consider organizing a reception. A system is used that offers a dialogue window in which all the requirements for the reception must be completed:

- a text field "No. persons" for the number of persons,
- three other text fields for date ("Date"), start time ("Ora inizio") and end time ("Ora fine"),
- a group of radio buttons "Type of service" for the type of dinner: at the table or a buffet,
- a list of “Dishes" with available dishes,
- two buttons: "OK" and "Cancel".

The scenario for the operation of the window is:

1. At the beginning, only "No. people" and "Cancel".
2. When entering a number between 10 and 99 in the "No. Persons" the fields "Date", "Start time", "Finish time" and the group "Type of service" appear only if there is a room big enough for the number of people. Each change in the number of people deletes the content of the fields viewed.
3. "Start time" < "Fine time".
4. Completing the fields "Date", "Start time" and "End time" and the "Type of service" group, the "Markets" list is visualized because the list depends on the season and the reception time.
5. After choosing the first course, the "OK" button becomes visible.Another level of connection between the window components results from the scenario. Each component is involved in at least two dependencies. It will be difficult to code 15 possible links in the window.
   The solution is to use an object that has a single connection with all components and leave the components separate. The object is called Mediator and encapsulates the logic of the window's behavior.

Mediator (II)

Applicability

1. When a group of objects communicate with each other in a well-defined but complex way.
2. When reusing an object is difficult because the object uses and communicates with other objects.
3. When a behavior distributed in several classes must be customized without using sub-classification.

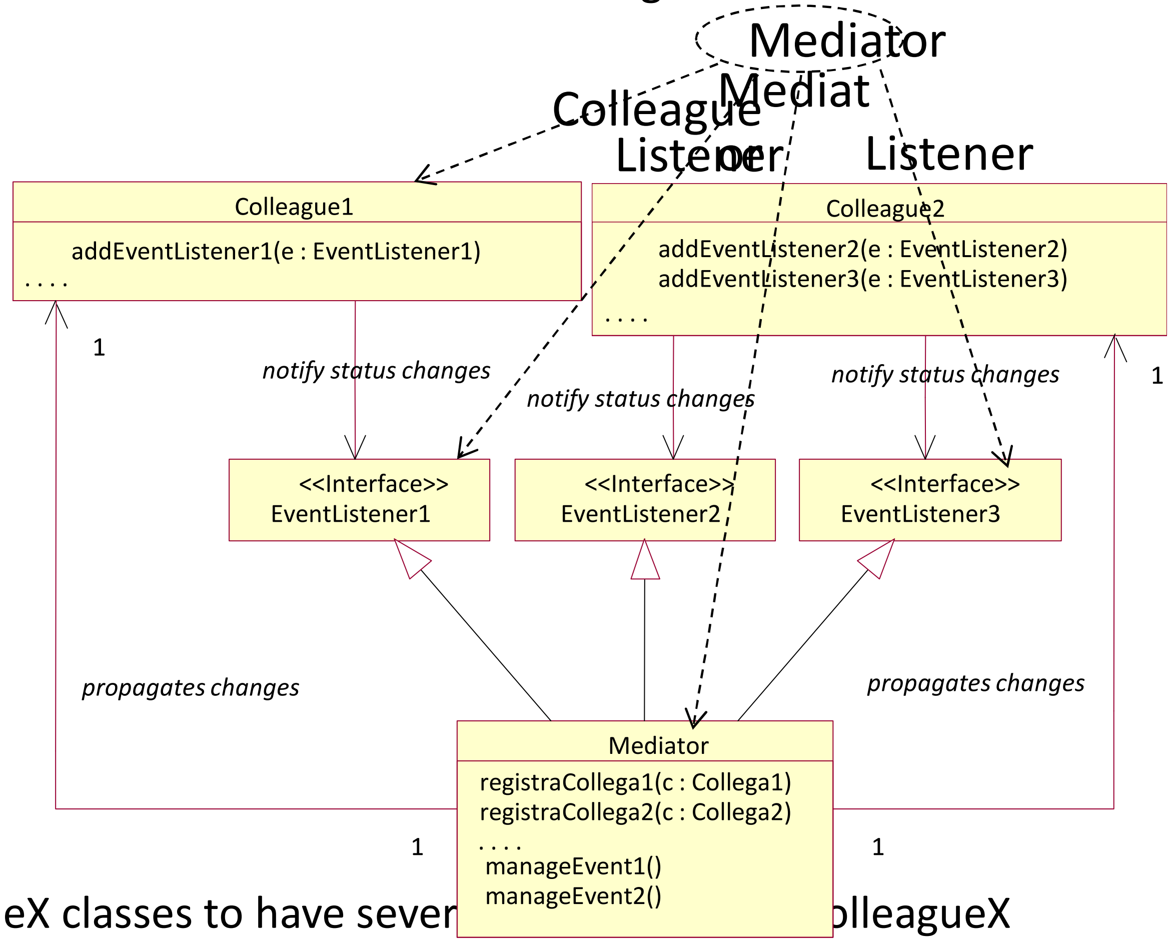
Structure

ColleagueX - Instance that have mutual dependency due to their condition. These dependencies can be of two types:

- An object requires the approval of another object to change its state into a certain way.
- An object must announce other objects after making a certain type of modification of the state.

The two dependencies are managed the same way. ColleagueX instances are associated with a Mediator object:

- When we want the approval for a changing of the state calls a method of the Mediator.
- When we want to announce other objects on another method is called for changing the state of the Mediator.



Mediator
Colleague
Mediat
Listener
Listener
Colleague1
addEventListener1(e : EventListener1)
. . . .
Colleague2
addEventListener2(e : EventListener2)
addEventListener3(e : EventListener3)
. . . .
1
notify status changes
notify status changes
notify status changes
1
<<Interface>> EventListener1
<<Interface>> EventListener2
<<Interface>> EventListener3
propagates changes
propagates changes
Mediator
registraCollega1(c : Collega1)
registraCollega2(c : Collega2)
. . . .
manageEvent1()
manageEvent2()
1
1

EventListener - Interface that allows ColleagueX classes to have sever… ColleagueX classes are not aware that they are working with Mediator. Each interface defines how to work with a certain type of events. To announce a certain state, ColleagueX calls the method corresponding to the interface without knowing the class of the Mediator object that implements the method.

Mediator (III)

Mediator - the instances of the class have a logic for processing notifications coming from ColleagueX objects. The class implements several EventListenerY interfaces. Through these interfaces the Mediator object is informed about the state change. In the case of a request to modify the status of one part of the ColleagueX objects, the invoked method approves or rejects the modification. To announce the state change, the method usually boils down to propagating the announcement to the other objects.Mediator has registerColleagueX(ColleagueX) methods that can be called to associate its objects with ColleagueA objects. The registerColleagueX method passes a ColleagueX object and usually calls one or more aggEventListenerY() methods to inform the ColleagueX object which in turn informs the Mediator object of its state change. The mechanism is similar to the event delegation model in Java.

Implementation

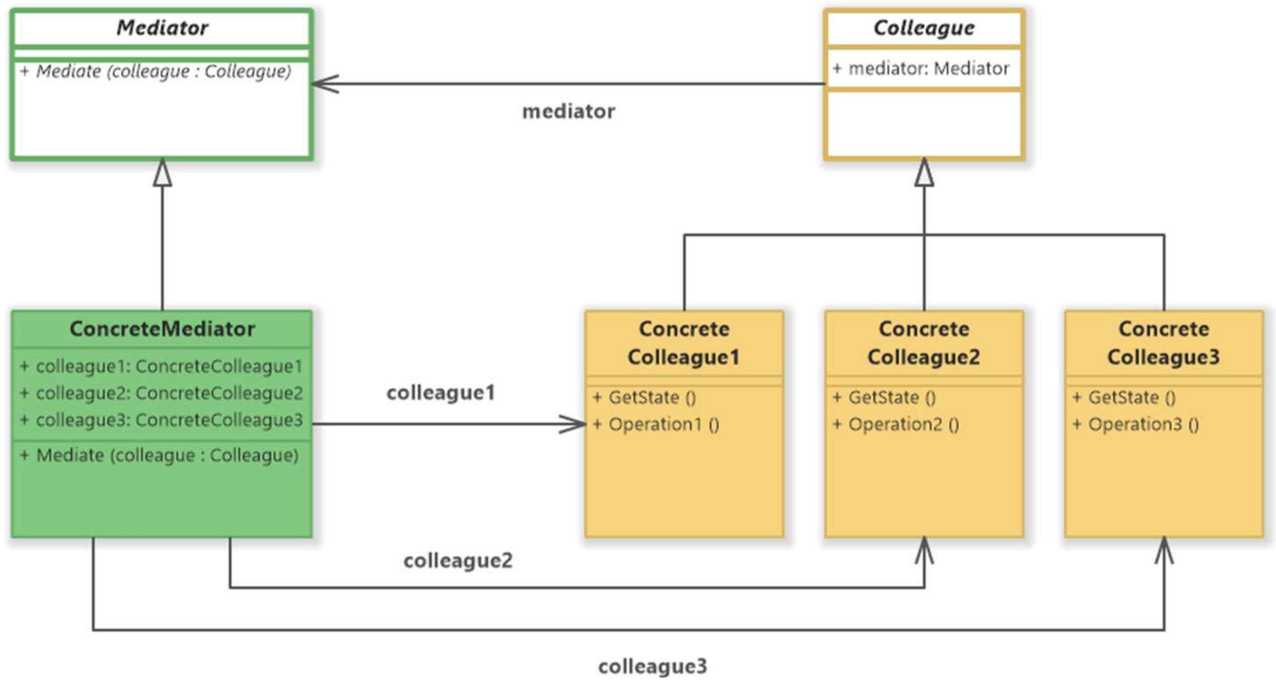
Usually, a single object (frame or dialog) is responsible for the creation of ColleagueX objects and the Mediator object, which is also their container. In this case, Mediator is an internal class of this object. This design increases the robustness of the program.

For implementation, it will be decided whether the Mediator is aware of the state of the ColleagueX objects or must be informed every time it knows the state of the object.In the first case, the Mediator will have a variable for each ColleagueX object, which when initializing the creation of the object, this variable is added when the ColleagueX object will announce a change in status.

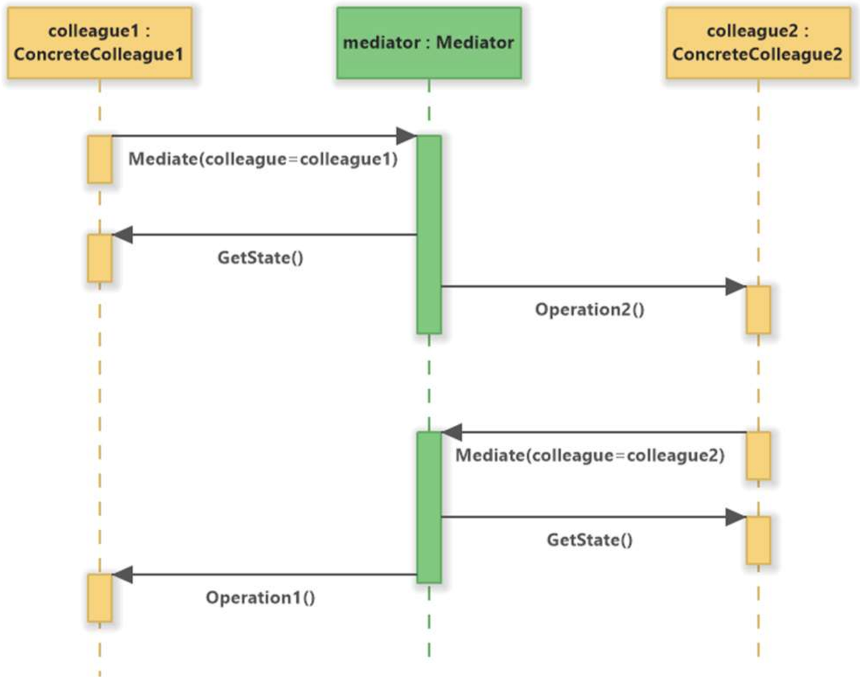
In the second case, every time the Mediator object receives a status change announcement from a ColleagueX, it receives the status of these ColleagueX objects on which its decisions are based. This approach is just as good, however in the first case, it happens that the Mediator does not receive the correct state of the ColleagueX objects..

Mediator (IV)

Structure



Collaborations


colleague1 : ConcreteColleague1
mediator : Mediator
colleague2 : ConcreteColleague2
Mediate(colleague=colleague1)
GetState()
Operation2()
Mediate(colleague=colleague2)
GetState()
Operation1()

Observer (I)

Intent:

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Allow objects to dynamically register dependencies between objects so that an object will notify those objects that are dependent on it when its state changes.

Motivation

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
- For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too.
- The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.
- This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Observer (II)

Structure

Subject

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

Observer

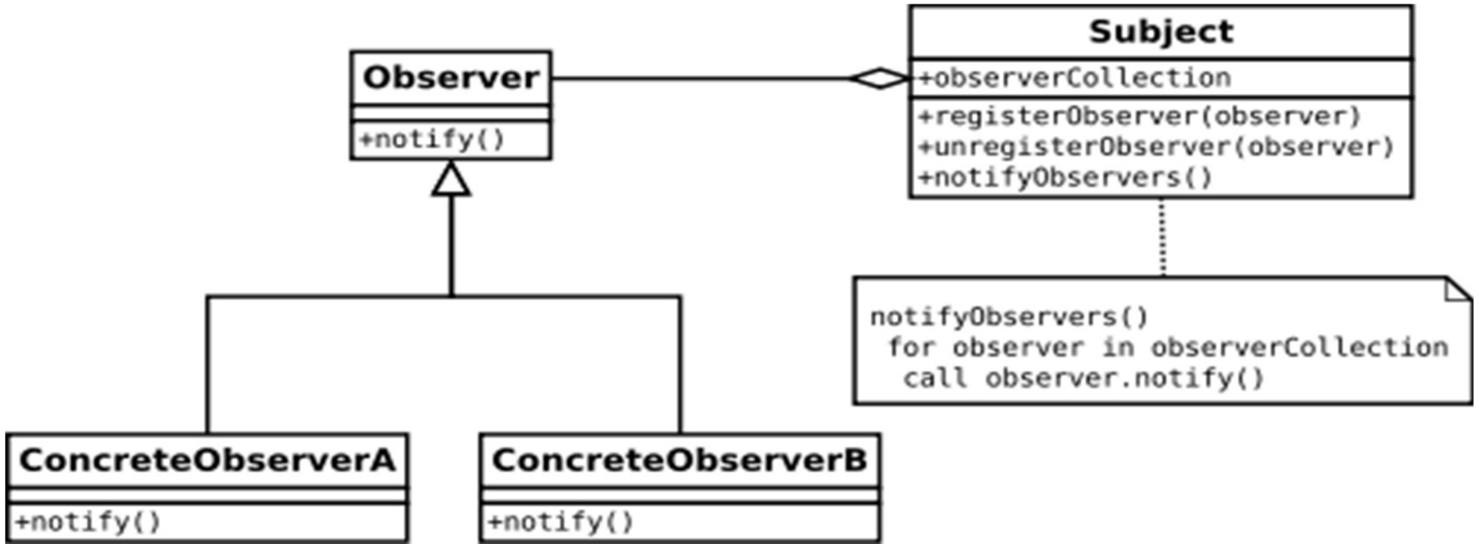
- defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject

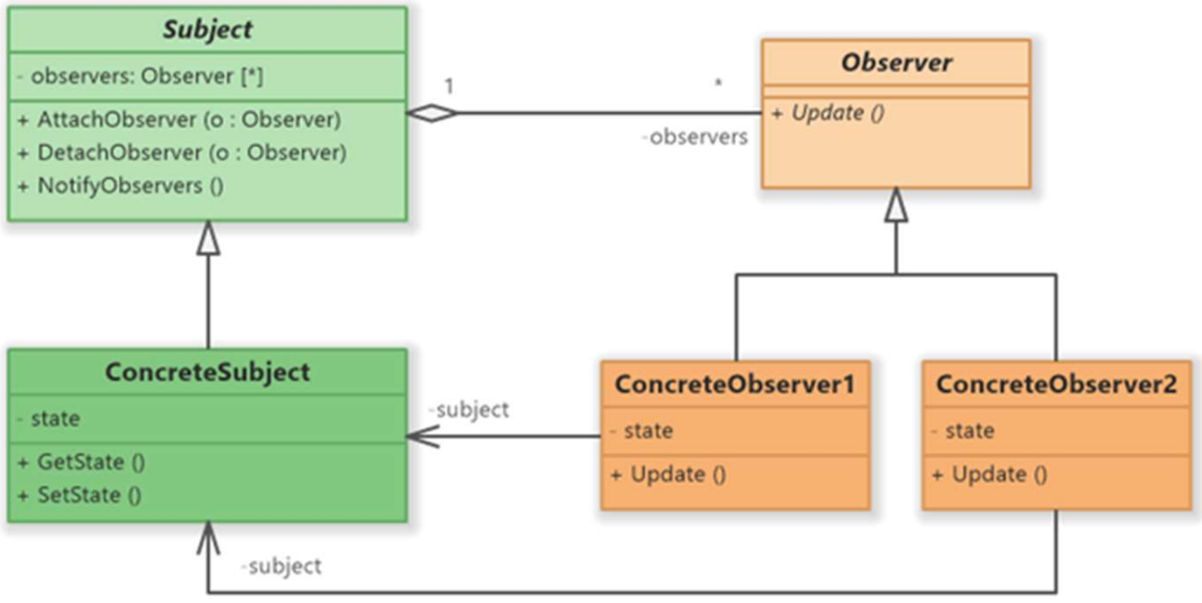
- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

ConcreteObserver

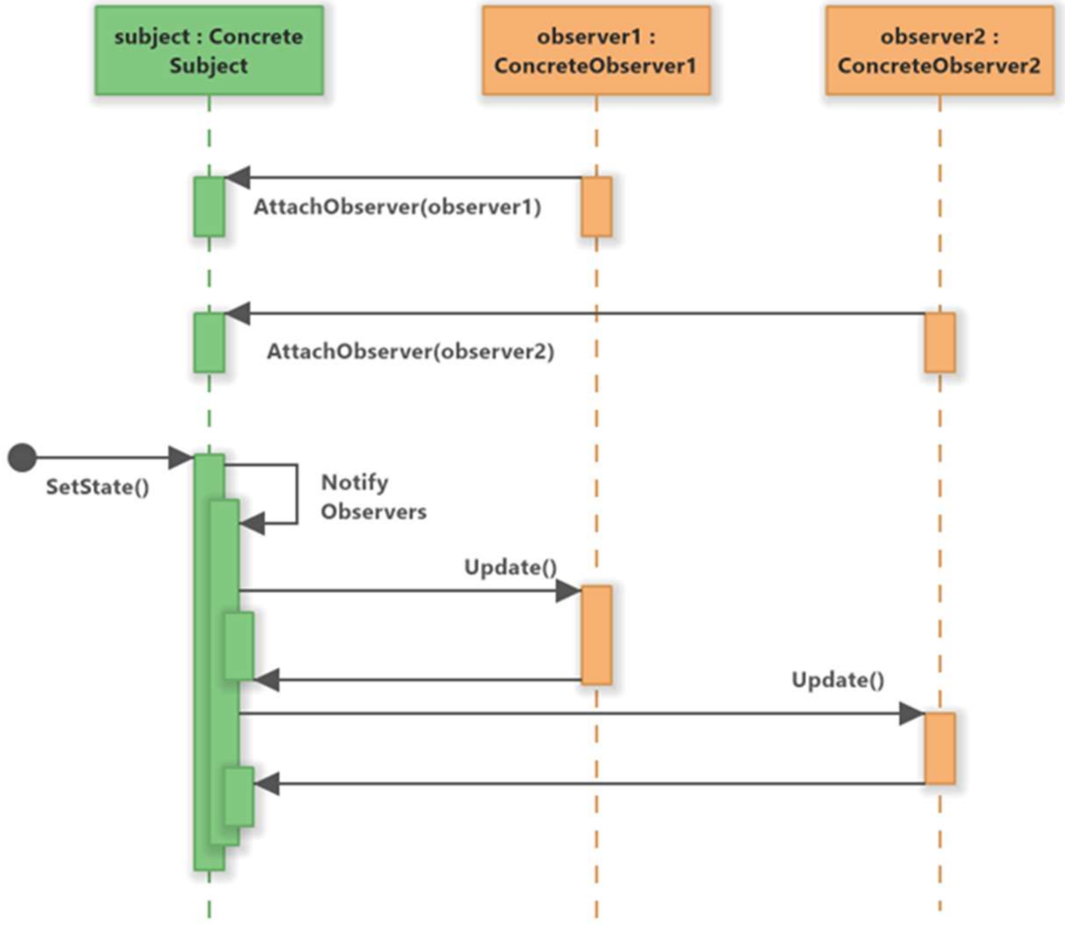
- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.



Observer



Structure

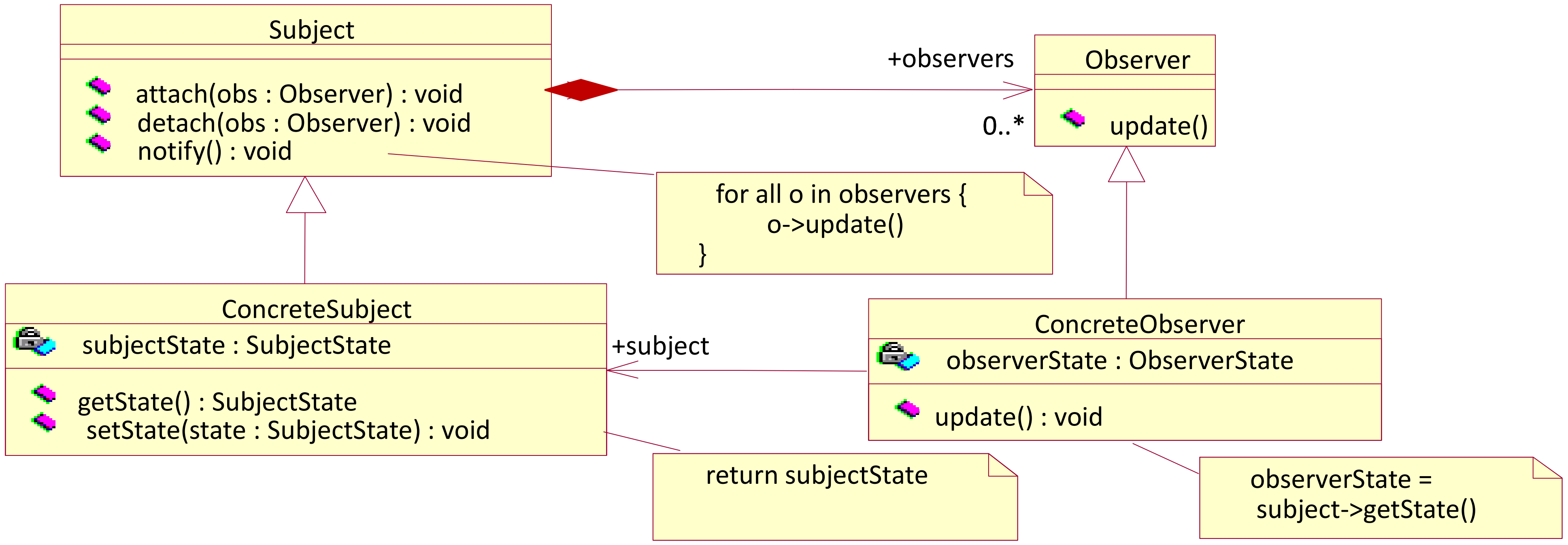

subject : Concrete Subject
observer1 : ConcreteObserver1
observer2 : ConcreteObserver2
AttachObserver(observer1)
AttachObserver(observer2)
SetState()
Notify Observers
Update()
Update()

Collaborations

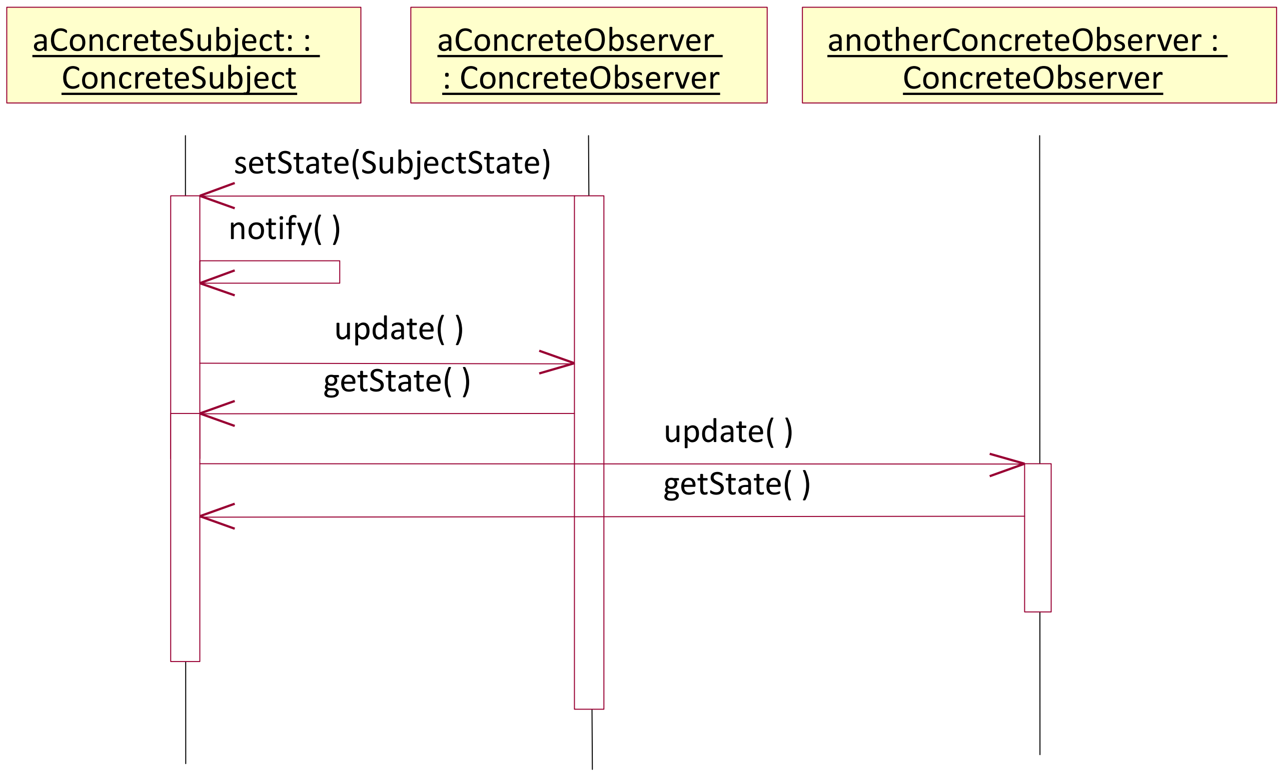
Observer (III)

Applicability

1. When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
2. When a change to one object requires changing others, and you don't know how many objects need to be changed.
3. 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.



Observer (III)


aConcreteSubject: : ConcreteSubject
aConcreteObserver : ConcreteObserver
anotherConcreteObserver : ConcreteObserver
setState(SubjectState)
notify( )
update( )
getState( )
update( )
getState( )

Collaborations:

1. ConcreteObserver *may* set the state of ConcreteSubject
2. ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
3. 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.

The Observer object that initiates the change request postpones its update until it gets a notification from the subject. Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely.

Observer (IV)

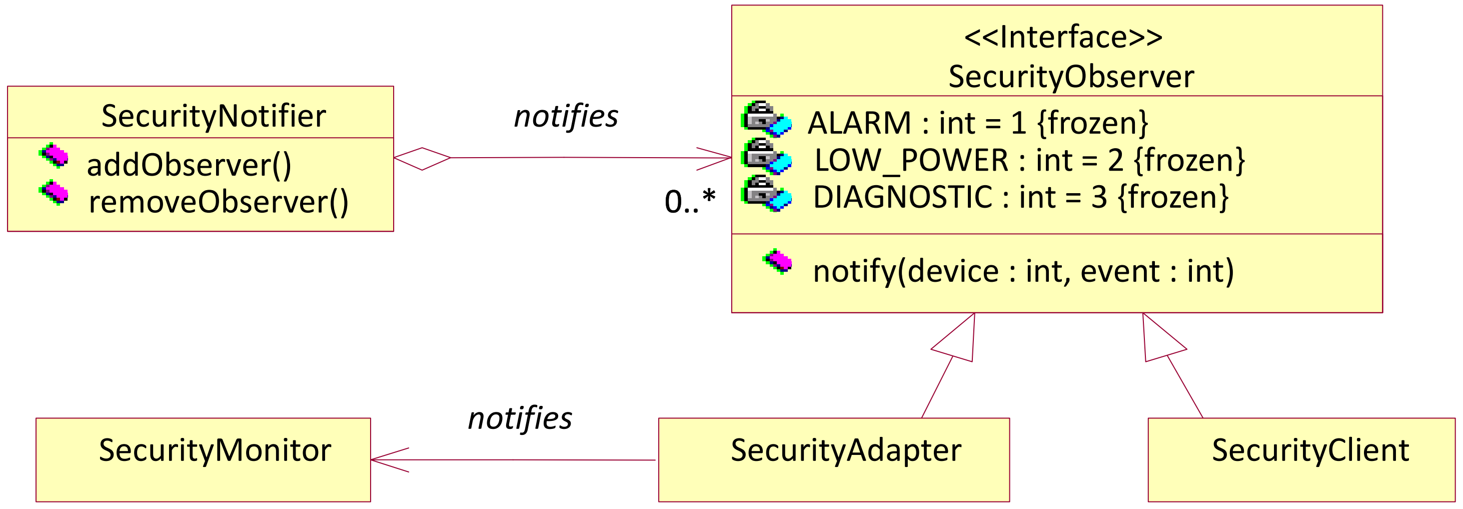
Consequencies

- *Abstract coupling between Subject and Observer.* A subject knows that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal. Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).
- *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.
- *Unexpected updates.* Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.

Observer (V)

Motivation

- Suppose that you are working for a company that manufactures smoke detectors, motion sensors, and other security devices. To take advantage of new market opportunities, your company plans to introduce a new line of devices. These devices will be able to send a signal to a security card that can be installed in most computers. The hope is that companies that make security-monitoring systems will integrate these devices and cards with their systems. To make it easy to integrate the cards with monitoring systems, you have been given the task of creating an easy-to-use API.
- The API must allow your future customers to easily integrate their programs with it so their programs will receive notifications from the security card. It must work without forcing the customers to alter the architecture of their existing software. All that the API may assume about the customer's software is that at least one, and possibly more than one, object will have a method that should be called when a notification is received from a security device.



Observer (VI)

Solution

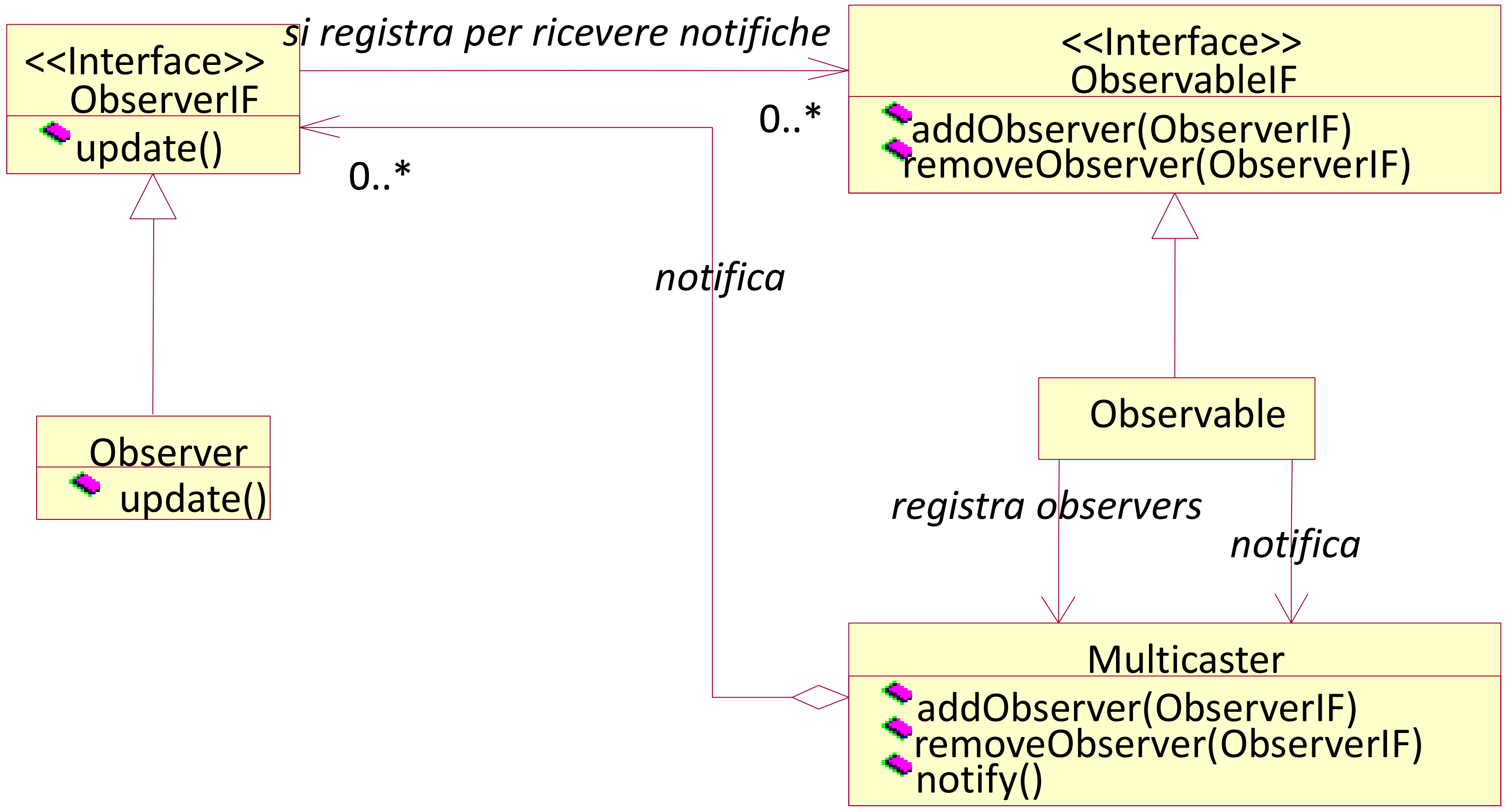
Instances of the SecurityNotifier class receive notifications from the security card. They, in turn, notify objects that previously requested to receive notifications. Only objects that implement the SecurityObserver interface can be registered with a SecurityNotifier object to receive notifications from it. A SecurityObserver object becomes registered to receive notifications from a SecurityNotifier object when it is passed to the SecurityNotifier object's addObserver method. Passing it to the Security Notifier object's removeObserver method ends the SecurityObserver object's registration to receive notifications.

A SecurityNotifier object passes a notification to a SecurityObserver object by calling its notify method. The parameters it passes to its notify method are a number that uniquely identifies the security device that the original notification came from and a number that specifies the type of notification.

Java API Usage

Java's delegation event model is a specialized form of the Observer pattern. Classes whose instances can be event sources participate in the Observable role. Event listener interfaces participate in the ObserverIF role. Classes that implement event listener interfaces participate in the Observer role. Because there are a number of classes that deliver various subclasses of java.awt.AwtEvent to their listeners, there is a Multicaster class that they use called java.awt.AWTEventMulticaster.

Observer(VII)



Structure

ObserverIF. An interface in this role defines a method that is typically called notify or update. An Observable object calls the method to provide a notification that its state has changed, passing it whatever arguments are appropriate. In many cases, a reference to the Observable object is one of the arguments that allow the method to know what object provided the notification.

Observer(VIII)

Structure (cont.d)

- **Observer.** Instances of classes in this role implement the ObserverIF interface and receive state change notifications from Observable objects.
- **ObservableIF.** Observable objects implement an interface in this role. The interface defines two methods that allow Observer objects to register and unregister to receive notifications.
- **Observable.** A class in this role implements the ObservableIF interface. Its instances are responsible for managing the registration of ObserverIF objects that want to receive notifications of state changes. Its instances are also responsible for delivering the notifications. The Observable class does not directly implement these responsibilities. Instead, it delegates these responsibilities to a Multicaster object.
- **Multicaster.** Instances of a class in this role manage registration of ObserverIF objects and deliver notifications to them on behalf of an Observable object. The purpose of this role is to increase reuse of code. Delegating these responsibilities to a Multicaster class allows their implementation to be reused by all Observable classes that implement the same ObservableIF interface or deliver notifications to objects that implement the same ObserverIF interface.

Observer (IX)

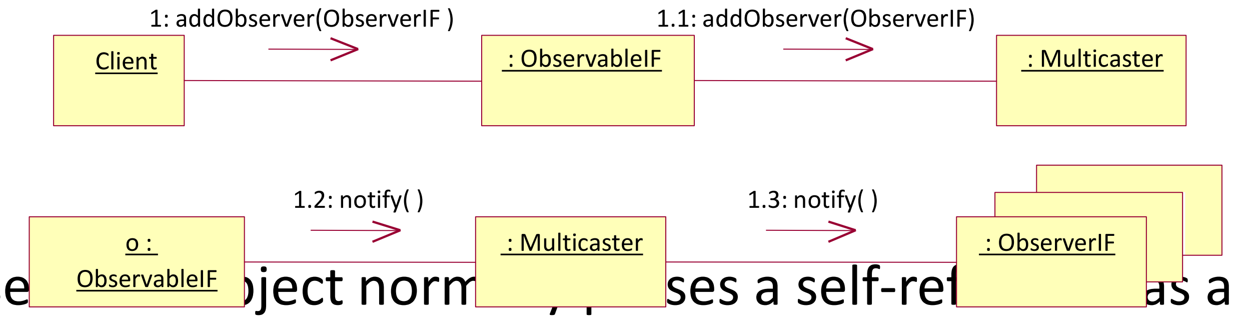
Collaborations

1. Objects that implement an ObserverIF interface are passed to the addObserver method of an ObservableIF object.

1.1 The ObservableIF object delegates the addObserver call to its associated Multicaster object. The Multicaster object adds the ObservableIF object to the collection of ObserverIF objects that it maintains.

2. The ObservableIF object labeled o needs to notify other objects that its state has changed. It initiates the notification by calling the notify method of its associated Multicaster object.

2.1 The Multicaster object calls the notify method of each one of the ObserverIF objects in its collection.



An Obse bject norm ses a self-ref as a parameter to an Observer object’s notify method. In most cases, the Observer object needs access to the Observable object’s attributes in order to act on the notification. Here are some ways to provide that access:

Observer (X)

1. Add methods to the ObservableIF interface for fetching attribute values. This is usually the best solution. However, it works only if all the classes that implement the ObservableIF interface have a common set of attributes sufficient for Observer objects to act on notifications.
2. You can have multiple ObservableIF interfaces, with each providing access to enough attributes for an Observer object to act on notifications. To make that work, ObserverIF interfaces must declare a version of their notify method for each one of the ObservableIF interfaces. However, requiring observer objects to be aware of multiple interfaces removes much of the original motivation for having ObservableIF interfaces. Requiring a class to be aware of multiple interfaces is not much better than requiring it to be aware of multiple classes, so this is not a very good solution.
3. You can pass attributes that ObserverIF objects need as parameters to their notify methods. The main disadvantage of this solution is that it requires Observable objects to know enough about ObserverIF objects to provide them with the correct attribute values. If the set of attributes required by ObserverIF objects changes, then you must modify all of the Observable classes accordingly.
4. You can dispense with the ObservableIF interface and pass the Observable objects to ObserverIF objects as instances of their actual class. This implies overloading the ObserverIF interface’s notify method, so that there is a notify method for each Observable class that will deliver notifications to ObserverIF objects.

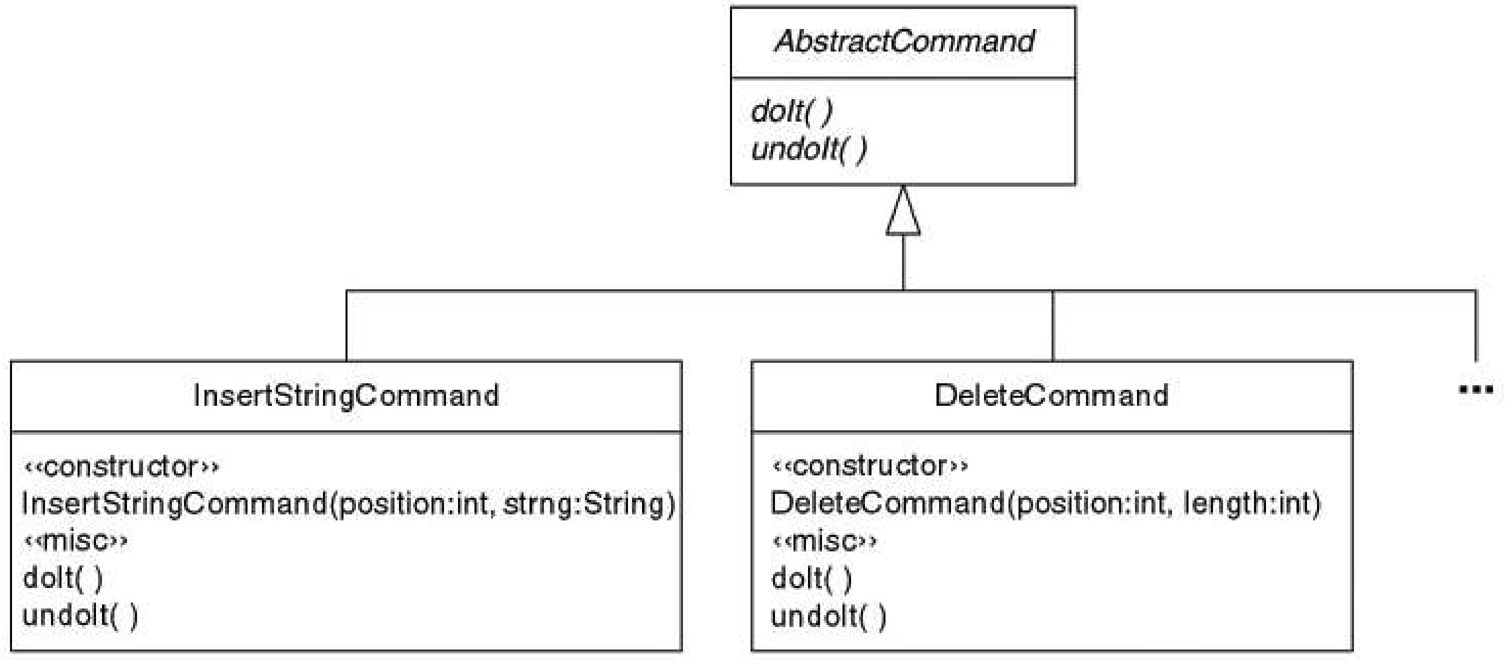
Command (I)

Intent

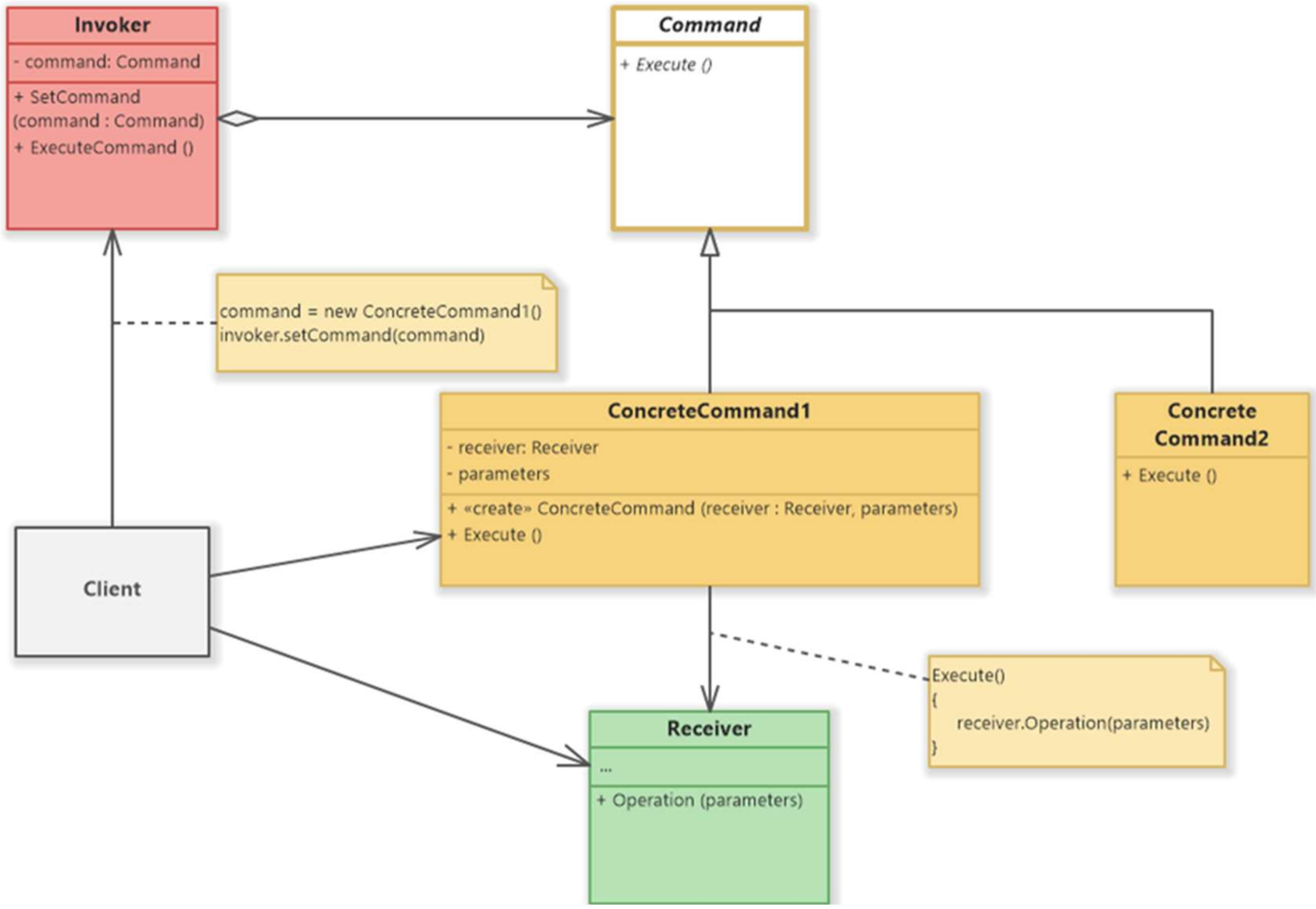
- Encapsulate commands in objects so that you can control their selection and sequencing, queue them, undo them, and otherwise manipulate them.
- An object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Motivation (I)

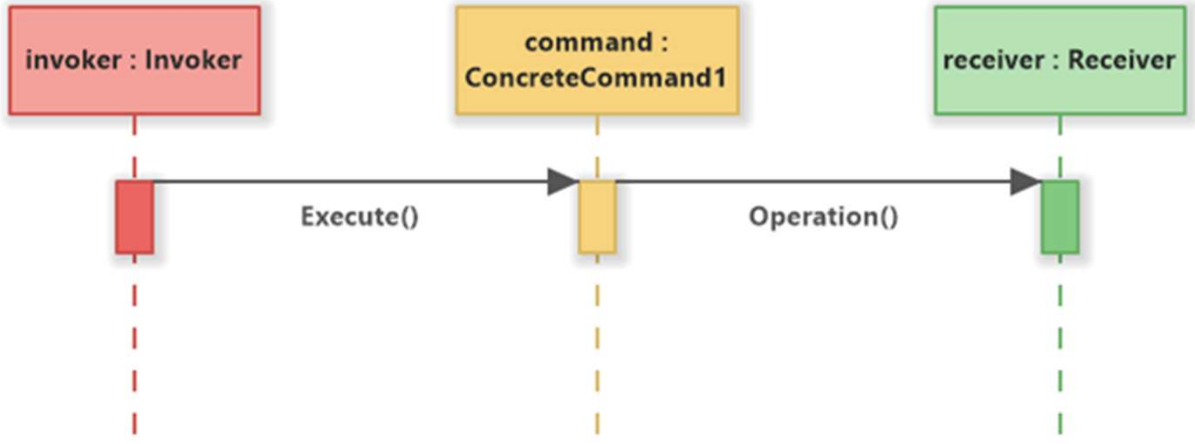
Suppose you want to design a word processing program so that it can undo and redo commands. A way to accomplish this is to materialize each command as an object with do and undo methods.



Command



Structure



Collaborations

Command (II)

1. When you tell the word processor to do something, instead of directly performing the command, it creates an instance of the subclass of AbstractCommand corresponding to the command. It passes all necessary information to the instance's constructor.
2. Once the word processor has materialized a command as an object, it calls the object's doIt method to execute the command.
3. The word processor also puts the command object in a data structure that allows the word processor to maintain a history of the commands that have been executed. Maintaining a command history allows the word processor to undo commands in the reverse order that they were issued by calling their undo methods.

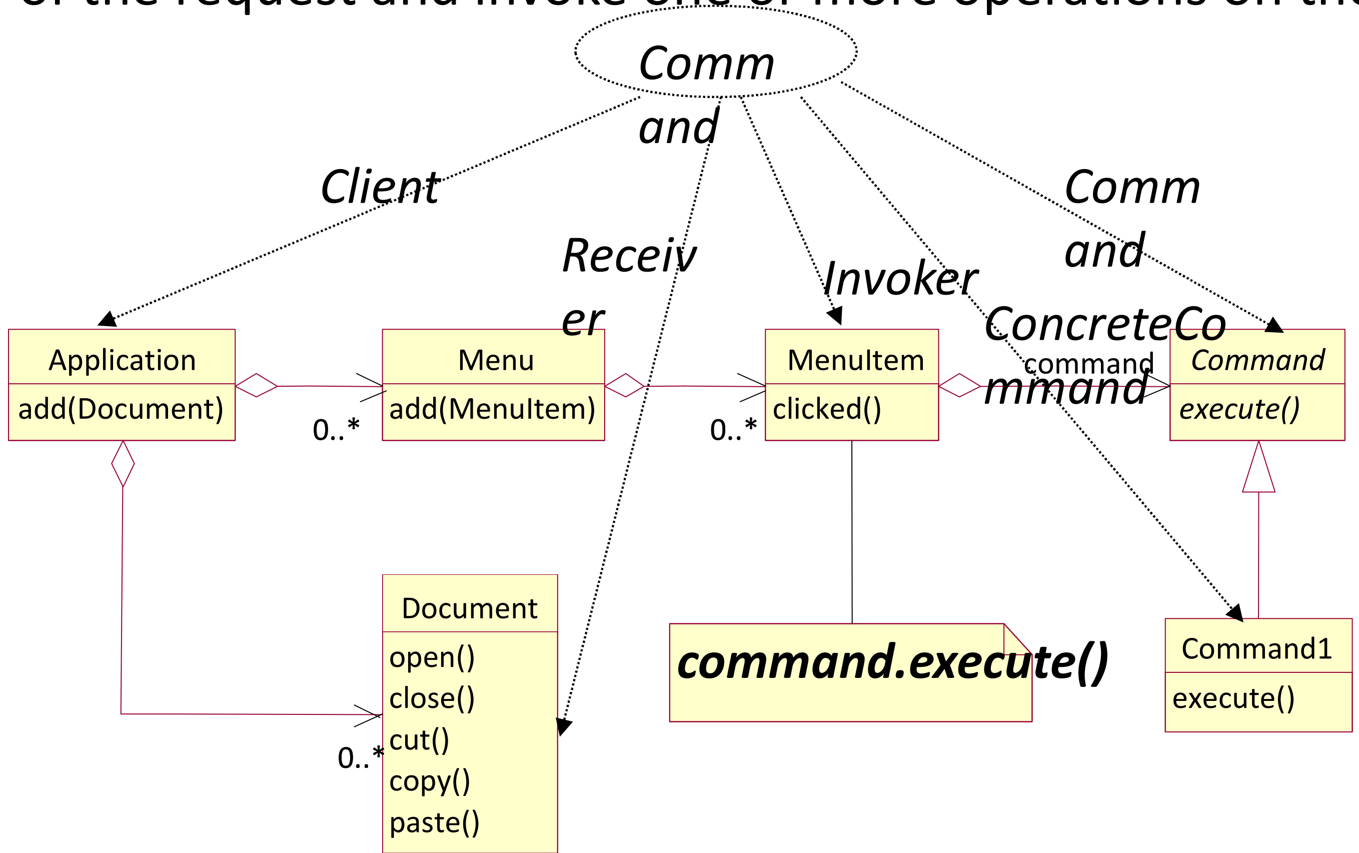
Command (III)

Motivation (II)

- User interface toolkits include objects like buttons and menus that carry out a request in response to user input. But the toolkit can't implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which object. As toolkit designers we have no way of knowing the receiver of the request or the operations that will carry it out.
- The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects. *The key to this pattern is an abstract Command class, which declares an interface for executing operations.* In the simplest form this interface includes an abstract `execute` operation. Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing `execute` to invoke the request. The receiver has the knowledge required to carry out the request.

Command (IV)

- Menus can be implemented easily with Command objects. Each choice in a Menu is an instance of a MenuItem class.
- Consider a word processor where an Application class creates the menus and their menu items along with the rest of the user interface. Application also keeps track of Document objects that a user has opened.
- The application configures each MenuItem with an instance of a concrete Command subclass. When the user selects a MenuItem, the MenuItem calls `execute` on its command, and `execute` carries out the operation. MenuItems don't know which subclass of Command they use. Command subclasses store the receiver of the request and invoke one or more operations on the receiver.



Command - Motivation

- In each of these examples, notice how the Command pattern *decouples the object that invokes the operation from the one having the knowledge to perform it*. This gives us a lot of flexibility in designing our user interface.
  - For instance, an application can provide both a menu and a push button interface to a feature just by making the menu and the push button share an instance of the same concrete Command subclass.
  - We can replace commands dynamically, which would be useful for implementing context-sensitive menus.
  - We can also support command scripting by composing commands into larger ones.
- All of this is possible because the object that issues a request only needs to know how to issue it; it doesn't need to know how the request will be carried out.

Applicability

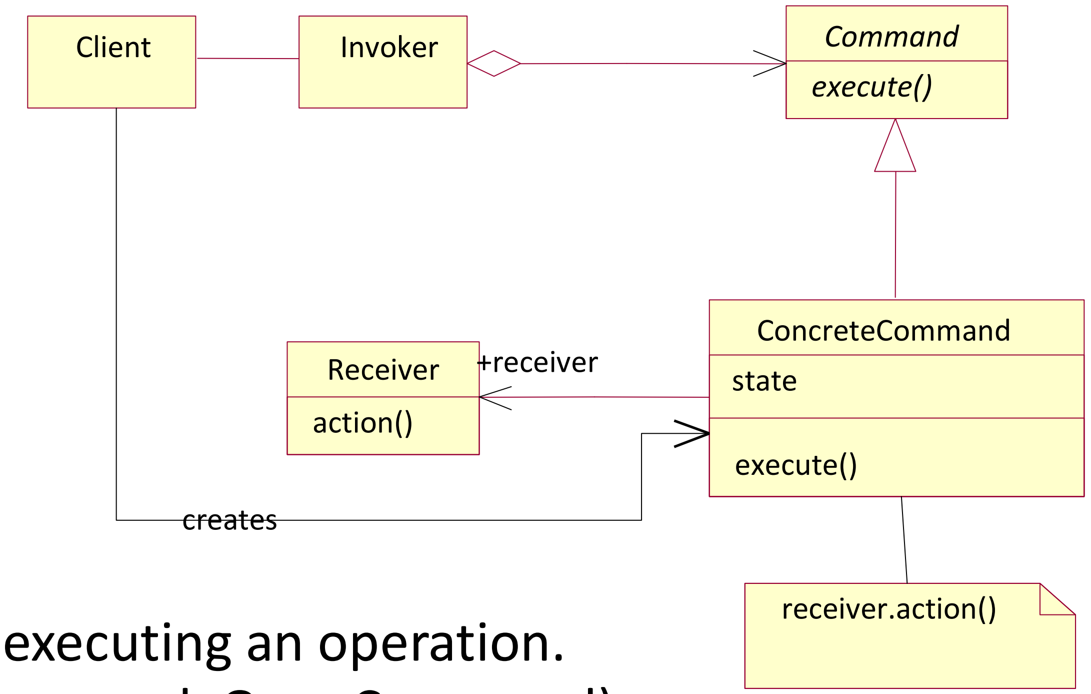
Use the Command pattern when you want to:

- *parameterize objects by an action to perform*, as MenuItem objects. You can express such parameterization in a procedural language with a **callback** function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
- *have a more versatile and sofisticated reference to an object* than a simple identifier.

Command - Applicability

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

Command - Structure



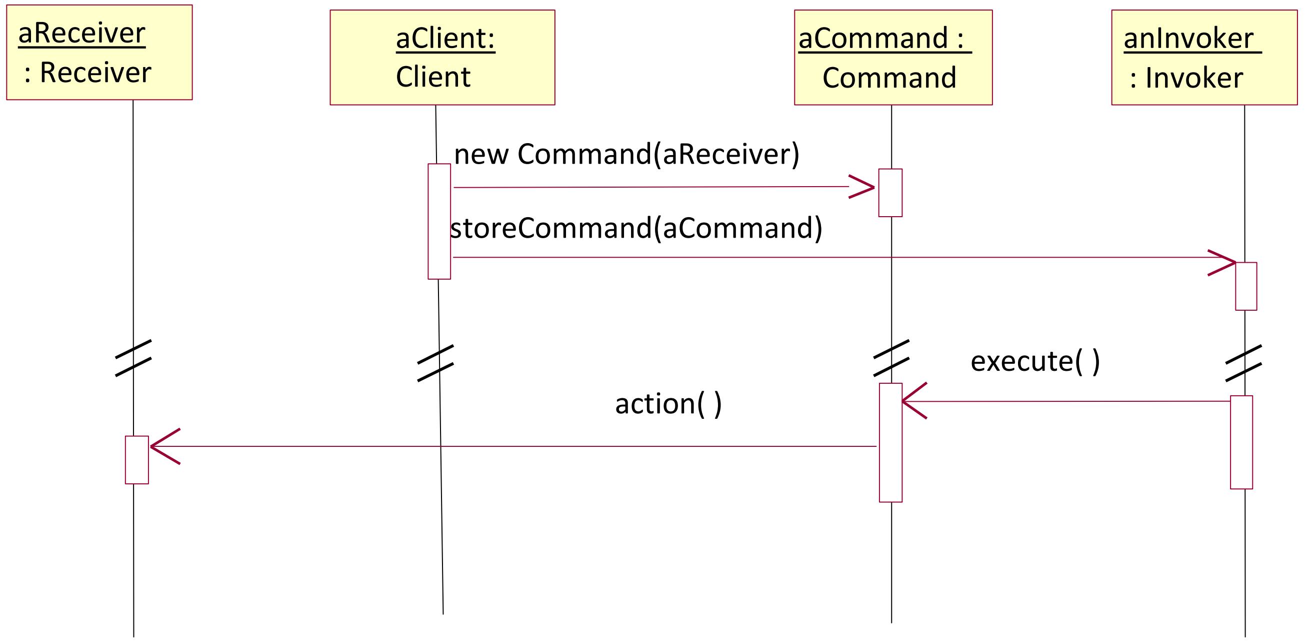
Structure

- **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.

Command - Collaborations

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.



Command - Applicability

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. In general, a composite command is an instance of the Composite pattern.
- It's easy to add new Commands, because you don't have to change existing classes.

Implementation

Consider the following issues when implementing the Command pattern:

- *How intelligent should a command be?* A command can have a wide range of abilities. At one extreme it merely defines a binding between a receiver and the actions that carry out the request. At the other extreme it implements everything itself without delegating to a receiver at all. The latter extreme is useful when you want to define commands that are independent of existing classes, when no suitable receiver exists, or when a command knows its receiver implicitly. For example, a command that creates another application window may be just as capable of creating the window as any other object. Somewhere in between these extremes are commands that have enough knowledge to find their receiver dynamically.

Command - Implementation

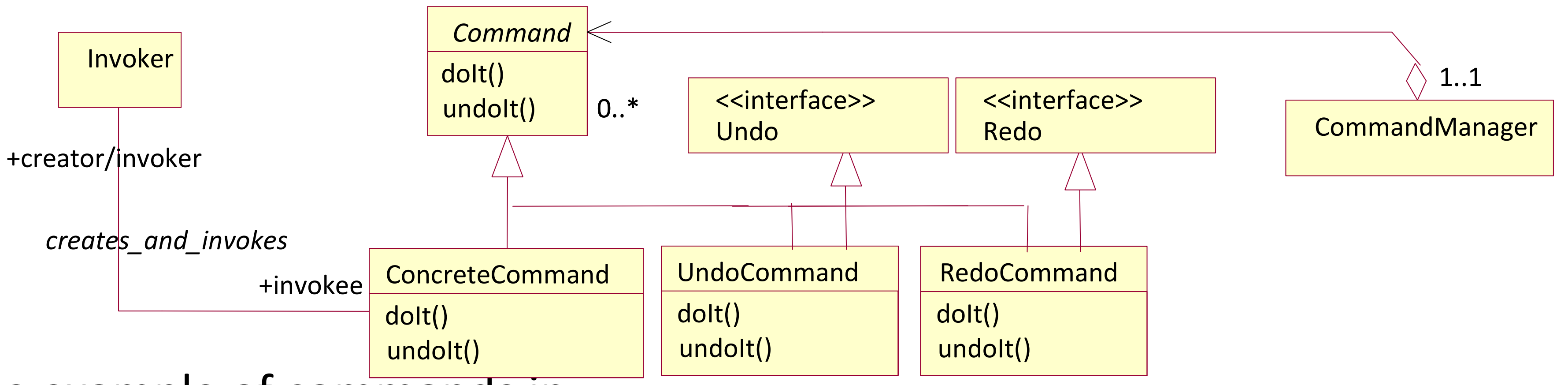
- *Supporting undo and redo.* Commands can support undo and redo capabilities. A ConcreteCommand class might need to store additional state to do so. This state can include
  - the Receiver object, which actually carries out operations in response to the request,
  - the arguments to the operation performed on the receiver, and
  - any original values in the receiver that can change as a result of handling the request.

  The receiver must provide operations that let the command return the receiver to its prior state.
- To support one level of undo, an application needs to store only the command that was executed last. For multiple-level undo and redo, the application needs a **history list** of commands that have been executed, where the maximum length of the list determines the number of undo/redo levels. The history list stores sequences of commands that have been executed. Traversing backward through the list and reverse-executing commands cancels their effect; traversing forward and executing commands reexecutes them.

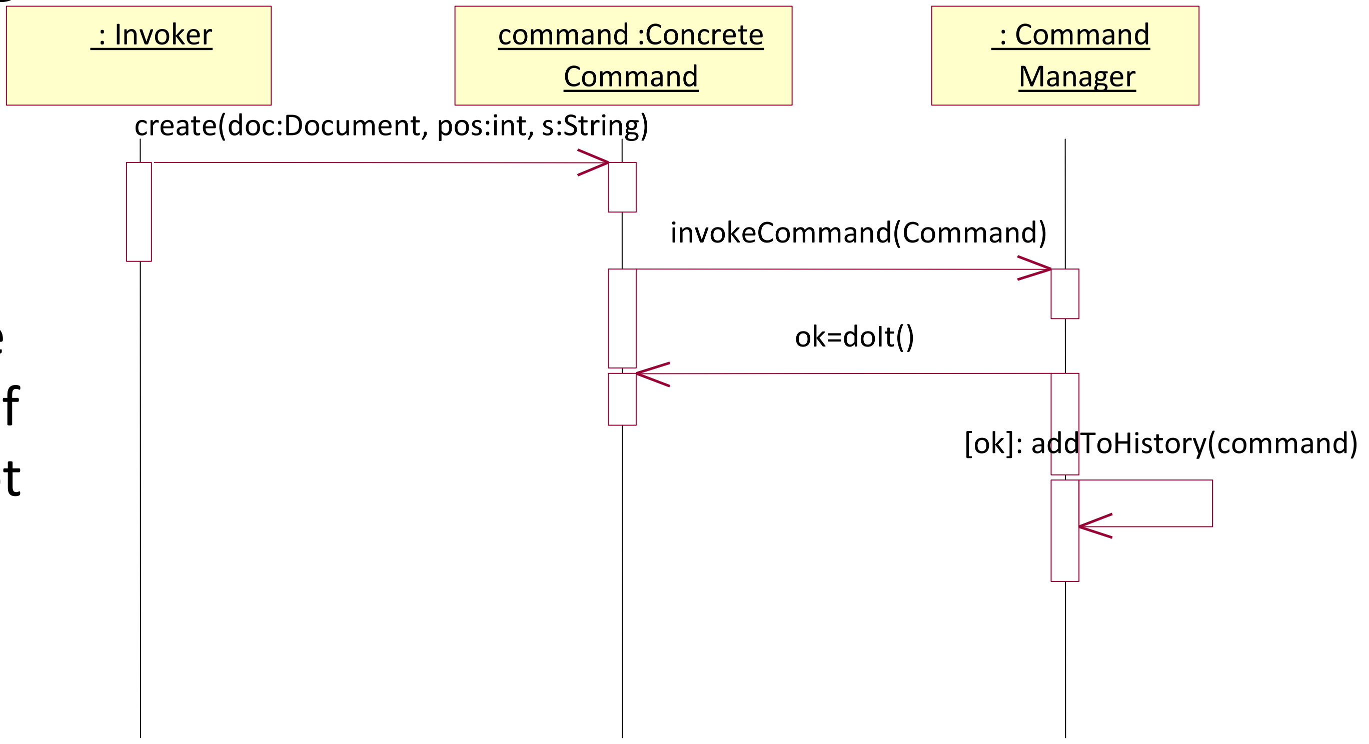
Command - Implementation

- An undoable command might have to be copied before it can be placed on the history list. That's because the command object that carried out the original request, say, from a MenuItem, will perform other requests at later times. Copying is required to distinguish different invocations of the same command if its state can vary across invocations.
- For example, a DeleteCommand that deletes selected objects must store different sets of objects each time it's executed. Therefore the DeleteCommand object must be copied following execution, and the copy is placed on the history list. If the command's state never changes on execution, then copying is not required—only a reference to the command need be placed on the history list. Commands that must be copied before being placed on the history list act as prototypes.
- *Avoiding error accumulation in the undo process.* Hysteresis can be a problem in ensuring a reliable, semantics-preserving undo/redo mechanism. Errors can accumulate as commands are executed, unexecuted, and reexecuted repeatedly so that an application's state eventually diverges from original values. It may be necessary therefore to store more information in the command to ensure that objects are restored to their original state.

Command - Undo/Redo Sample



The example of commands in a word processor that can be undone and redone. CommandManager is responsible with the management of a collection of commands created by the Invoker. The responsability of the CommandManager is not only to manage undo and redo commands, but also to order commands execution when a command should be executed.


: Invoker
command :Concrete Command
: Command Manager
create(doc:Document, pos:int, s:String)
invokeCommand(Command)
ok=doIt()
[ok]: addToHistory(command)

Command - Code

```
public abstract class AbstractCommand {
   public final static CommandManager manager = new
          CommandManager();
   public abstract boolean doIt();
   public abstract boolean undoIt();
} // class AbstractCommand
 /* Comando concreto. */
class ConcreteCommand extends AbstractCommand {
    private Document document;
    private String strng;
    private int position;
    ConcreteCommand(Document doc, int pos, String s) {
        this.document = doc;
        this.position = pos;
        this.strng = s;
        manager.invokeCommand(this);
    } // Constructor(Document, int, String)
    public boolean doIt() {
        try {
            document.insertStringCommand(position, strng);
        } catch (Exception e) {
            return false;
        } // try
        return true;
    } // doIt()
   public boolean undoIt() {
        try {
            document.deleteCommand(position, strng.length());
        } catch (Exception e) {
            return false;
        } // try
        return true;
    } // undoIt()
} // class ConcreteCommand
interface UnDo {
} // interface Undo
class UndoCommand implements Undo {
    public boolean doIt() {
        throw new NoSuchMethodError();
    }  // doIt()
    public boolean undoIt() {
        throw new NoSuchMethodError();
    } // undoIt()
} // UndoCommand
```

```
import java.util.LinkedList;
class CommandManager {
    private int maxHistoryLength = 100;
    private LinkedList history = new LinkedList();
    private LinkedList redoList = new LinkedList();
    public void invokeCommand(AbstractCommand command) {
        if (command instanceof UndoCommand) {// se implementa Undo
            undo();
            return;
        } // if undo
        if (command instanceof RedoCommand) { //se implementa Redo
            redo();
            return;
        } // if redo
        if (command.doIt()) {
            addToHistory(command);
        } else { // command cannot be undone
            history.clear();
        } // if
        if (redoList.size() > 0)
          redoList.clear();
    } // invokeCommand(AbstractCommand)
    private void undo() {
        if (history.size() > 0) {
            AbstractCommand undoCommand;
            undoCommand = (AbstractCommand)history.removeFirst();
            undoCommand.undoIt();
            redoList.addFirst(undoCommand);
        } // if
    } // undo()
    private void redo() {
        if (redoList.size() > 0) {
            AbstractCommand redoCommand;
            redoCommand = (AbstractCommand)redoList.removeFirst();
            redoCommand.doIt();
            history.addFirst(redoCommand);
        } // if
    } // redo()
    private void addToHistory(AbstractCommand command) {
        history.addFirst(command);
        if (history.size() > maxHistoryLength)
          history.removeLast();
    } // addToHistory(AbstractCommand)
} // class CommandManager
```

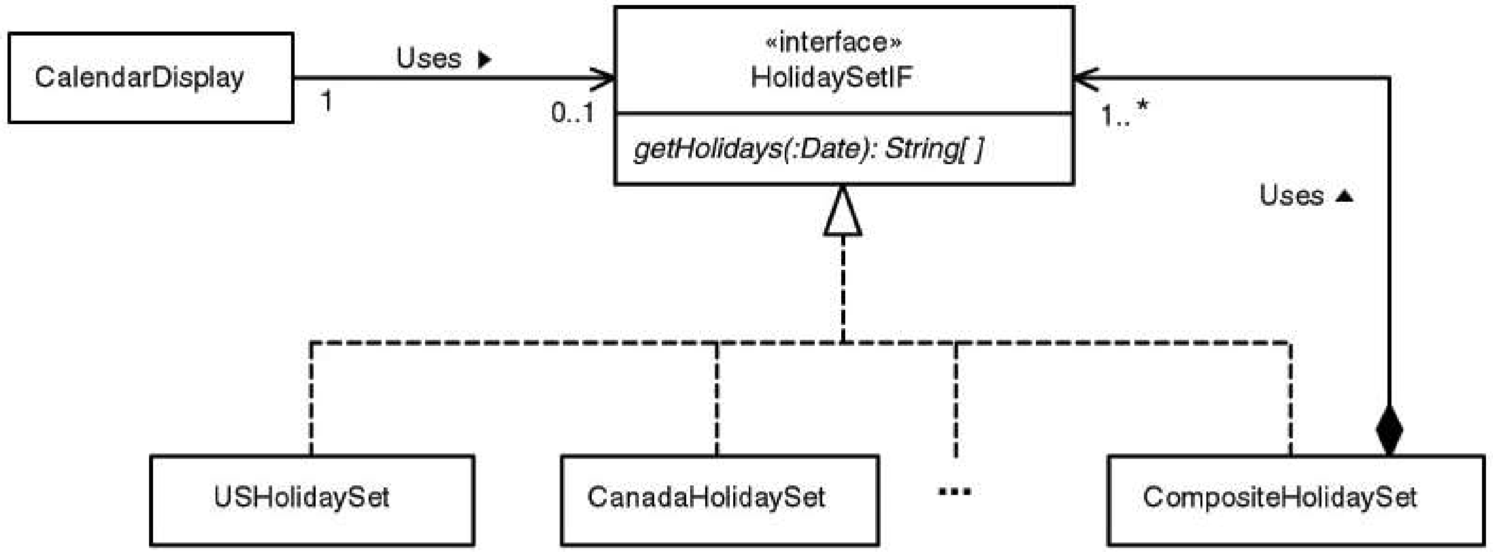
Strategy - Motivation

Intent

Encapsulate related algorithms in classes that implement a common interface. This allows the selection of algorithm to vary by object. It also allows the selection of algorithm to vary over time.

Motivation

Suppose you have to write a program that displays calendars. One of the requirements for the program is that it be able to display sets of holidays celebrated by different nations and different religious groups. The user must be able to specify which sets of holidays to display. You would like to satisfy the requirement by putting the logic for each set of holidays in a separate class. This will give you a set of small classes to which you could easily add more classes. You want classes that use these holiday classes to be unaware of any specific set of holidays.



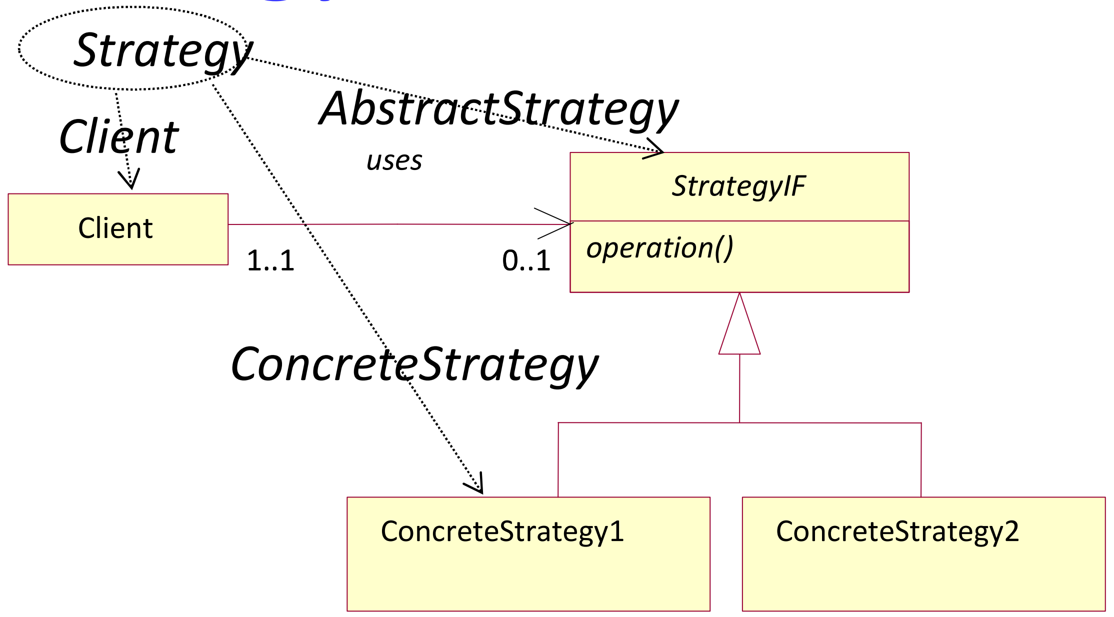
Strategy (II)

- If a Calendar Display object has a HolidaySetIF object to work with, it consults with that object about each day it displays, in order to find out if that day is a holiday. Such objects are either an instance of a class like USHoliday that identifies a single set of dates or they are an instance of CompositeHoliday. The CompositeHoliday class is used when the user requests the display of multiple sets of holidays. It is instantiated by passing an array of Holiday objects to its constructor.
- This arrangement allows a CalendarDisplay object to find out what holidays fall on a particular date by just calling a HolidaySetIF object's getHolidays method.

Applicability

- A program must provide multiple variations of an algorithm or behavior.
- You need to vary the behavior of each instance of a class.
- You need to vary the behavior of objects at runtime.
- Delegating behavior to an interface allows classes that use the behavior to be unaware of the classes that implement the interface and the behavior.
- If a behavior of a class's instances does not vary from instance to instance or over time, then it is simplest for the class to directly contain the behavior or directly contain a static reference to the behavior.

Strategy - Structure



Strutura

- **Client.** A class in the Client role delegates an operation to an interface. It does so without knowing the actual class of the object it delegates the operation to or how the class implements the operation.
- **StrategyIF.** An interface in this role provides a common way to access operations encapsulated by its subclasses.
- **ConcreteStrategy1, ConcreteStrategy2, and so on.** Classes in this role provide alternative implementations of the operation that the client class delegates.
- The Strategy pattern always occurs with a mechanism for determining the actual ConcreteStrategy object that the client object will use. The selection of a strategy is often driven by configuration information or events. However, the actual mechanism varies greatly. For this reason, no particular strategy-selecting mechanism is included in the pattern.

Strategy - Code

```java
import java.util.Date;

public abstract class Holiday {
   // constant to indicate there are no holidays
   protected final static String[] noHoliday = new String[0];
   /* The method returns a n array of strings which describe the
        holidays in a given date. Il the date is not holiday the method
        returns an array of  length 0     */
   abstract public String[] getHolidays(Date d) ;
} // class Holiday

import java.util.Date;
class PrintCalendar{
   private Holiday holiday;
   private static final String[] noHoliday = new String[0];
   //...
   /*
    in this class information on the dates to be displayed  is stored.
    */
   private class DateCache {
      private Date date;
      private String[] holidayStrings;

      DateCache(Date dt) {
         date = dt;
         //...
         if (holiday == null) {
            holidayStrings = noHoliday
         } else {
            holidayStrings = holiday.getHolidays(data);
         } // if
         //...
      } // construttore(Date)
   } // class DateCache
} // class PrintCalendar
```

```java
import java.util.Date;
/*  This class determines if the date is a holiday for a combination of
        caledars . */
public class CompositeHoliday extends Holiday {
   private Holiday[] arrayHolidays;

   public CompositeHoliday (Holidays[] h) {
      arrayHolidays= new Holiday[h.length];
      System.arraycopy(f, 0, arrayHolidays, 0, h.length);
   } // CompositeHoliday

   public String[] getHolidays(Date dt) {
      return getHolidays0(dt, 0, 0);
   } // getHolidays(Date)

   private String[] getHolidays0(Date dt, int offset, int ndx) {
      if (ndx >= arrayHoliday.length) {
         return new String[offset];
      } // if
      String[] holidays= arrayHolidays[ndx].getHolidays(dt);
      String[] result = getHolidays0(dt, offset+holidays.length, ndx+1);
      System.arraycopy(holidays, 0, result, offset, holidays.length);
      return result;
   } // getHolidays0(Date, int, int)
} // class CompositeHoliday

import java.util.Date;
public class HolidayRomania extends Holiday{
   public String[] getHolidays(Date dt) {
      String[] holidays = noHoliday;
      //...
      return holidays;
   } // getHolidays(Date)
} // class HolidayRomania
```

Template Method - Intent

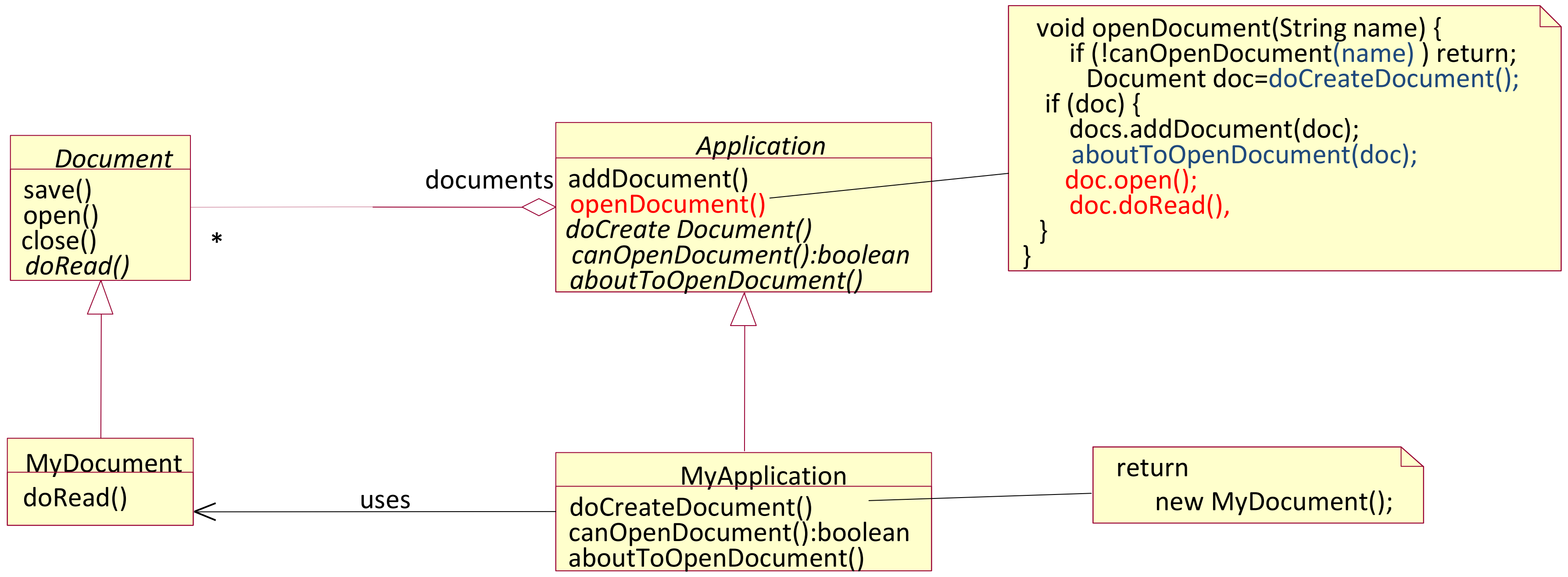
Intent

Include an abstract class that contains only part of the logic needed to accomplish its purpose. Organize the class so that its concrete methods call an abstract method where the missing logic would have appeared. Provide the missing logic in the subclass’s methods that override the abstract methods. In other words, 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

- Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.
- Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines DrawApplication and DrawDocument subclasses; a spreadsheet application defines SpreadsheetApplication and SpreadsheetDocument subclasses.
- The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation.
- OpenDocument defines each step for opening a document. It checks if the document can be opened, creates the application-specific Document object, adds it to its set of documents, and reads the Document from a file.

Template Method



```
void openDocument(String name) {
     if (!canOpenDocument(name) ) return;
       Document doc=doCreateDocument();
  if (doc) {
     docs.addDocument(doc);
     aboutToOpenDocument(doc);
    doc.open();
     doc.doRead(),
  }
}
```

- We call OpenDocument a **template method**. A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. Application subclasses define the steps of the algorithm that check if the document can be opened (CanOpenDocument) and that create the Document (DoCreateDocument). Document classes define the step that reads the document (DoRead). The template method also defines an operation that lets Application subclasses know when the document is about to be opened (AboutToOpenDocument), in case they care.
- By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering, but it lets Application and Document subclasses vary those steps to suit their needs.

Template Method (III)

Motivation(II)

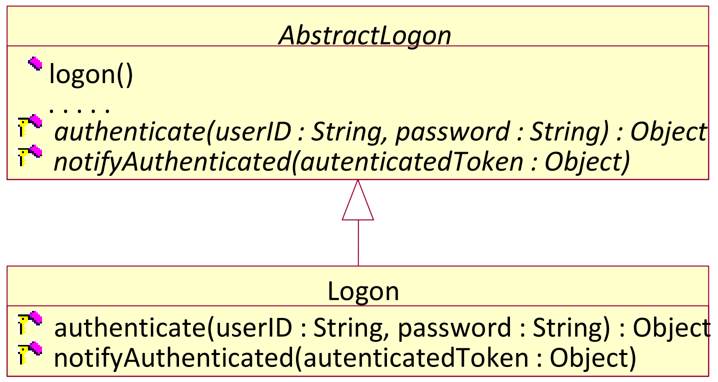
Suppose that you have the task of writing a reusable class for logging users into an application or applet. In addition to being reusable and easy to use, the tasks of the class will be to:

1. Prompt the user for a user ID and password.
2. Authenticate the user ID and password. The result of the authentication operation should be an object. If the authentication operation produces some information needed later as proof of authentication, then the object produced by the authentication operation should encapsulate the information.
3. While the authentication operation is in progress, the user should see a changing and possibly animated display that tells the user that authentication is in progress and all is well.
4. Notify the rest of the application or applet that login is complete and make the object produced by the authentication operation available to the rest of the application.

- Two of these tasks-prompting the user and assuring the user that authentication is in progress-are application independent. Though the strings and images displayed to the user may vary with the application, the underlying logic will always be the same.
- The other two tasks-authenticating the user and notifying the rest of the application-are application specific. Every application or applet will have to provide its own logic for these tasks.

Template Method (IV)

- The way that you organize your Logon class will be a large factor in how easy it is for developers to use. Delegation is a very flexible mechanism. You could simply organize a Logon class so that it delegates the tasks of authenticating the user and notifying the rest of the application. Though this approach gives a programmer a lot of freedom, it does not help guide a programmer to a correct solution.
- You can achieve a fill-in-the-blanks organization by defining the Logon class to be an abstract class that defines abstract methods that correspond to the application-dependent tasks that the programmer must supply code for. To use the Logon class, a programmer must define a subclass of the Logon class.
  Solution: The AbstractLogon class has a method called logon that contains the top-level logic for the top-level task of logging a user on to a program. It calls the abstract methods authenticate and notifyAuthentication to perform the program-specific tasks of authenticating a user and notifying the rest of the program when the authentication is accomplished.


AbstractLogon
logon()
. . . . .
authenticate(userID : String, password : String) : Object
notifyAuthenticated(autenticatedToken : Object)
Logon
authenticate(userID : String, password : String) : Object
notifyAuthenticated(autenticatedToken : Object)

Template Method - Applicability

Applicability

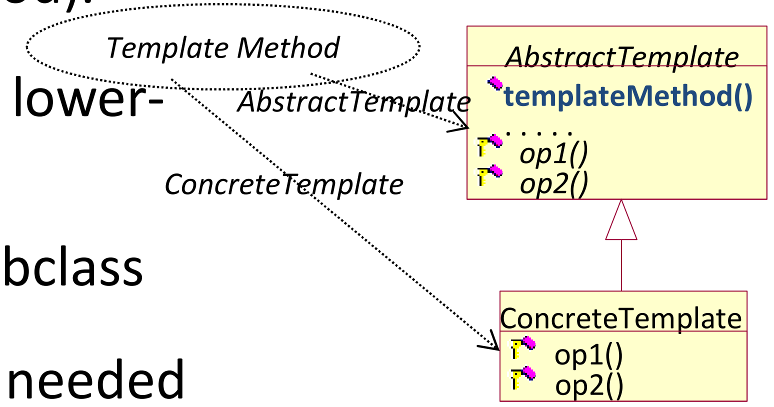
The Template Method pattern should be used:

- 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“: 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

AbstractTemplate. A class in this role has a concrete method that contains the class’s top-level logic (templateMethod). This method calls other methods, defined in the AbstractTemplate class as abstract methods, to invoke lower-level logic that varies with each subclass of the AbstractTemplate class.

ConcreteTemplate. A class in this role is a concrete subclass of an AbstractTemplate class. It overrides the abstract methods defined by its superclass to provide the logic needed to complete the logic of the templateMethod method.



Template Method

Consequences

A programmer writing a subclass of an AbstractTemplate class is forced to override those methods that must be overridden to complete the logic of the superclass. A well-designed template method class has a structure that provides a programmer with guidance in providing the basic structure of its subclasses.

Implemention

Clasa AbstractTemplate provides the templateMethod() method and some abstract operations which have to be implemented by subclasses. Such methods are called hook methods and are named with the –hook suffix.

Sometimes, the subclasses would not implement these operations. It is better to implement them as no-op operations.

Code Sample

First is the AbstractLogon class. It participates in the Template Method pattern in the AbstractTemplate role. Its template method is called logon. The logon method puts up a dialog that prompts the user for a user ID and password. After the user supplies a user ID and password, the logon method pops up a window telling the user that authentication is in progress. The window stays up while the logon method calls the abstract method authenticate to authenticate the user id and password. If the authentication is successful, it takes down the dialog boxes and calls the abstract method notifyAuthentication to notify the rest of the program that the user has been authenticated.

Template Method - Code

```
public abstract class AbstractLogon {
public void logon(Frame frame, String programName) {
        Object authenticationToken;
       LogonDialog logonDialog;
        logonDialog = new LogonDialog(frame, "Log on to "+programName);
        JDialog waitDialog = createWaitDialog(frame);
        while(true) {
          waitDialog.setVisible(false);
           logonDialog.setVisible(true);
           waitDialog.setVisible(true);
           try {
              String userID = logonDialog.getUserID();
              String password = logonDialog.getPassword();
             authenticationToken = authenticate(userID,  password);
              break;
           } catch (Exception e) {
             // Tell user that Authentication failed
              JOptionPane.showMessageDialog(frame,  e.getMessage(), "Authentication Failure", JOptionPane.ERROR_MESSAGE);
           }  // try
       } // Authentication successful
        waitDialog.setVisible(false);
        logonDialog.setVisible(false);
       notifyAuthentication(authenticationToken);
     } // logon() ...
    abstract protected Object authenticate(String userID, String password)   throws Exception;
    abstract   protected void notifyAuthentication(Object authToken) ;
} // class AbstractLogon
public class Logon extends AbstractLogon {
 ...
   protected Object authenticate(String userID, String password) throws Exception {
        if (userID.equals("abc") && password.equals("123"))
         return userID;
        throw new Exception("bad userID");
     } // authenticate
     protected void notifyAuthentication(Object authToken) { ...     } // notify(Object)
} // class Logon
```

Comments on Behavioral Patterns

Encapsulation variants

Behavioral Patterns frequently encapsulate an aspect, dividing the functionality: Strategy is an algorithm, Mediator is a collaboration protocol between objects, Iterator a way of moving components in an aggregate. These patterns have two types of objects: new objects that encapsulate the layout and existing objects that use them. The functionality of the new objects must belong to the existing objects if they are not patterns.
Chain of Responsibility is instead different: it uses an arbitrary number of objects that already existed in the system.

Objects as arguments

Some behavioral patterns introduce an object that is always used as an argument. Visitor uses this object for a polymorphic *accept()* operation on visited objects. Other patterns (Command) define objects that act like a ball that circulates internally and is invoked later. In Command the *command* operation is polymorphic.

Separation of senders from recipients

Command, Mediator and Chain of Responsibility promote a separation between senders and receivers. Command introduces the separation by means of a Command object that will represent the coupling between sender (invoker) and recipient (receiver). This fact allows the use of several recipients with the same sender, which can be easily reused. The recipient can be used as a parameter in the Command object and with different senders. In these cases, the sub-classification is used for each sender-recipient link.
Mediator separates the objects forcing them to communicate exclusively through the Mediator object. It directs the requests to the CollegueX objects and centralizes the communication. As the interface is fixed, Mediator must implement its own triage scheme to add flexibility to the pattern: the request can be coded and the arguments packaged so that CollegueX can receive requests for an unlimited number of operations. But the centralized triage scheme can be a source of execution insecurity.

HINTS FOR YOUR FIRST PROJECT

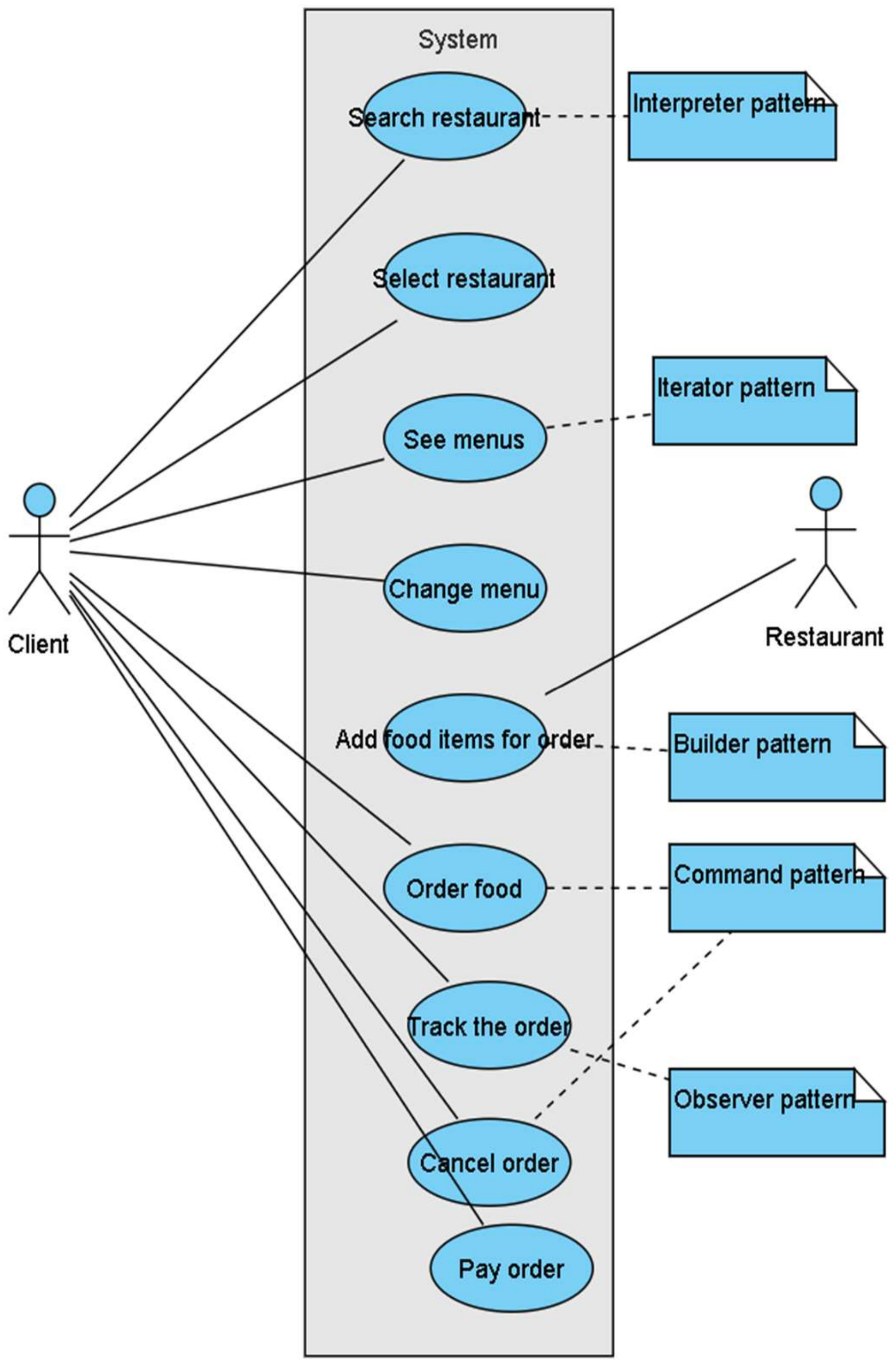
Hints in the Choice of Your Design Patterns

Problem: Design an online order food delivery application driven by the use cases.

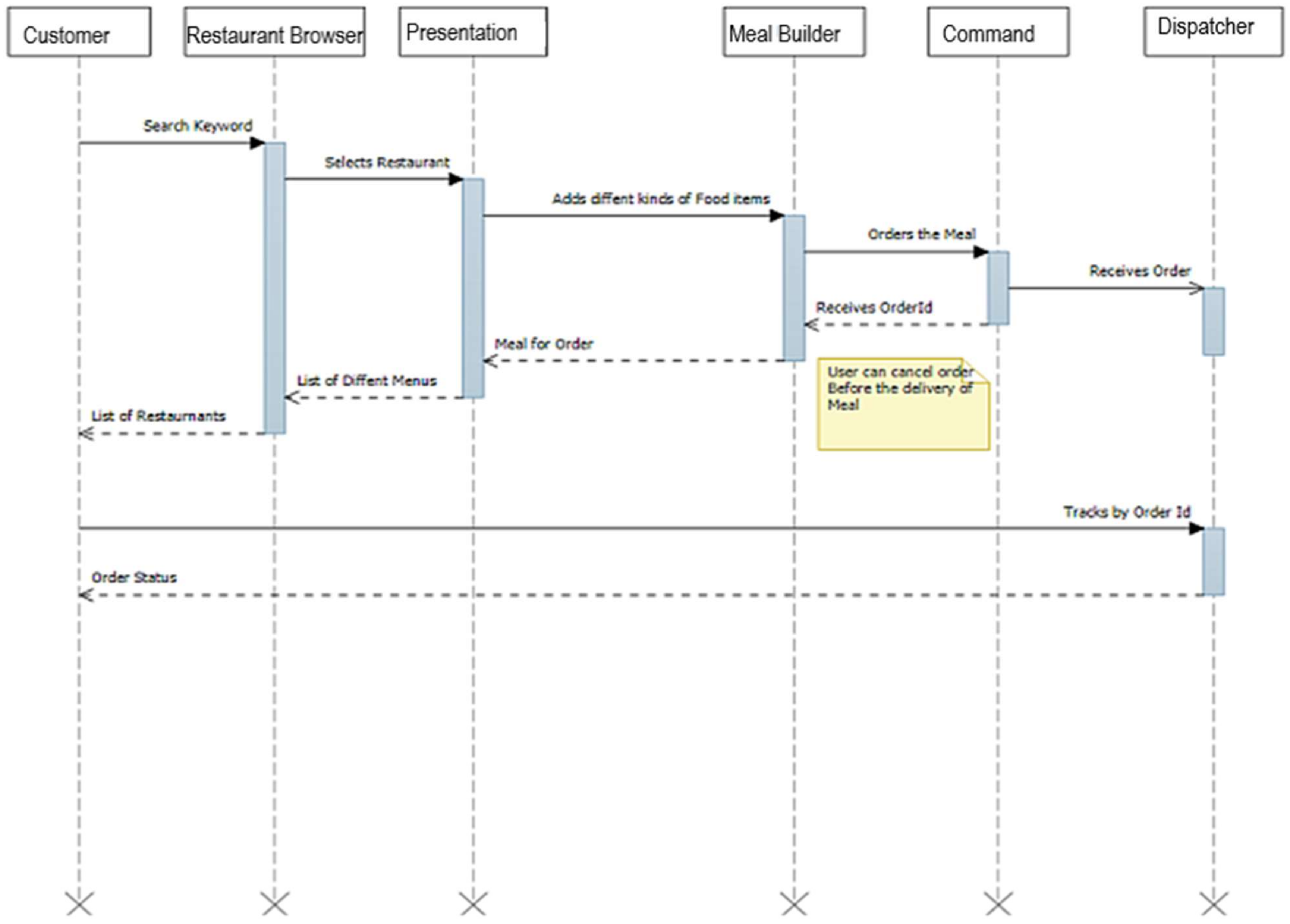
Solution: You may choose for each use case, according to its behavior and functionality, a dominant pattern, or more inter-related patterns.

For instance:

1. *Search restaurant* can be modeled using an **Interpreter** pattern that defines a grammatical representation for a language and provides an interpreter to deal with this grammar In our case, it should specify the way to interpret expression based on location, name of restaurant, food items, PIN, etc.
2. *See menus* can be modeled using an **Iterator** pattern that provides a way to access elements of an aggregate object sequentially without exposing the underlying structure of the object.
3. *Add food items to order* can be easily modeled with a **Builder** pattern used to configure and assemble complex objects. It provides a way to create the same object from different kinds of objects.
4. *Order food* and *Cancel order* can be modeled with the **Command** pattern. It wraps the request under an object that is called Command. That command object is being passed to invoker object.
5. Finally, Track the order can be modeled with the **Observer** pattern that can notify the user for each change in the state of the pack.



Sequence Diagram of a Food Order Delivery



END OF THE FIRST PART