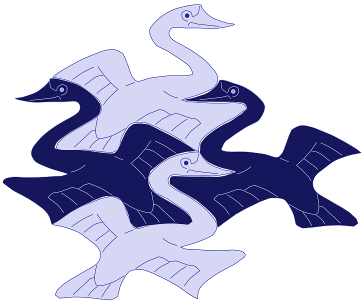


Design Patterns

Object Oriented Programming

http://softeng.polito.it/courses/09CBI

SoftEng http://softeng.polito.it

Version 3.8.1
© Marco Torchiano, 2021

Pattern

A reusable solution

to a known problem

in a well defined context

...just one of the possible definitions

Pattern

- Context
  - A (design) situation giving rise to a (design) problem
- Problem
  - Set of forces repeatedly arising in the context
    - Force: any relevant aspect of the problem (E.g., requirements, constraints, desirable properties)
- Solution
  - A proven resolution of the problem
  - Configuration to balance forces
    - Structure with components and relationships
    - Run-time behaviour

Example

SOCIAL PATTERN

- Context:
  - At the supermarket several customers crowd the gastronomy desk to get their fresh cut of ham
- Problem:
  - Customers quarrel to have their turn first
  - Order of arrival should be obeyed
  - It is hard to spot who arrived earlier or later
- Solution:
  - Provide numbered tickets the customer take as soon as they arrive and which they are called by

History

- Initially proposed by Christopher Alexander
- He described patterns for architecture (of buildings)
  - *The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing and when we create it. It is both a process and a thing …*

Types of Software Patterns

- Architectural Patterns
  - Address system wide structures
- Design Patterns
  - Leverage higher level mechanisms
- Idioms
  - Leverage language specific features

Architectural pattern

- Expresses a fundamental structural organization schema for software systems
- Provides a set of predefined components with their responsibilities
- Defines the rules and guidelines for organizing the relationships between the components

Example

ARCHITECTURAL PATTERN

- Context:
  - several programs that are used in sequence read from input and write sequentially to output
- Problem:
  - there are a lot of intermediate files used for communication between programs
- Solution:
  - adopt a pipe & filter architecture feeding a program with the result of the previous one

Design pattern

- Provides a scheme for refining components of a software system or their relationships
- Describes a commonly recurring structure of communicating components

Example

DESIGN PATTERN

- Context:
  - A class library providing few functionalities contains a lot of classes
- Problem:
  - The user is exposed to the internal complexity of the library
- Solution:
  - Create a new façade class that interacts with the user and hide all the details

Idiom

- Is a low-level pattern specific to a programming language
- Describes how to implement particular aspects of components or the relationships between them
- Leverages the features of a programming language

Example

IDIOM

- Context:
  - An attribute is constant and should be globally available to many classes
- Problem:
  - Opening access would allow unauthorized modifications
  - The attribute is repeated in every object
- Solution:
  - Make it **`public static final`**

Pattern Description

- Name
- Problem
- Context
- Forces
- Solution
- Force Resolution
- Design Rationale

Coplien

- Name
- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Related Patterns

GoF

Pattern language

- Pattern do not exist in isolation
  - Two or more patterns are applied together
  - A pattern is used to implement part of another pattern
  - A pattern can introduce a problem solved by another
- We have Pattern Languages
  - Or pattern systems

Pattern Language

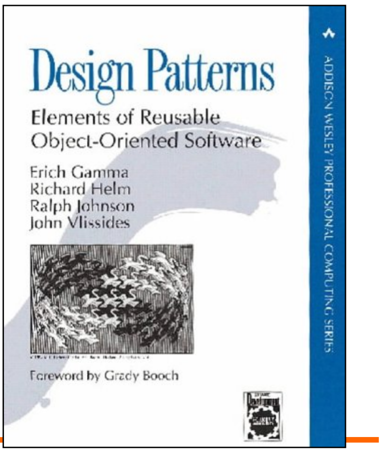
- Collection of patterns together with guidelines for
  - Implementation
  - Combination
  - Practical use
- Should
  - Count enough patterns
  - Describe patterns uniformly
  - Present relationships

Example

- MVC is implemented using
  - Observer
  - Iterator

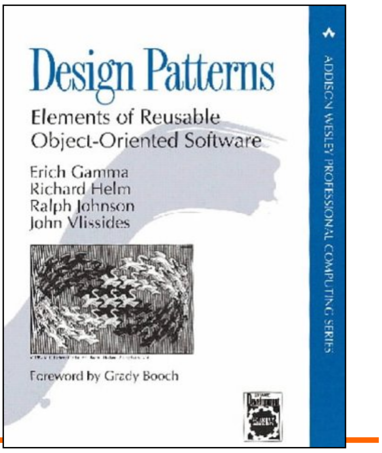
Design Patterns (GoF)

- Describe the structure of components
- Most widespread category of pattern
- First category of patterns proposed for software development


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns (GoF)

- Creational
  - E.g. Abstract Factory, Singleton
- Structural
  - E.g. Façade, Composite
- Behavioral
  - *Class:* e.g. Template Method
  - *Object:* e.g. Observer


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON WESLEY PROFESSIONAL COMPUTING SERIES

Design patterns

- Description of communicating objects and classes that are customized to solve a general design problem in a particular context
- A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design

Description

- Name and classification
- Intent
  - Also known as
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Pattern classification

- Purpose
  - Creational
  - Structural
  - Behavioral
- Scope
  - Class
  - Object

Pattern classification

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	1	1	2
	Object	4	6	10

Pattern selection

- Consider how patterns solve problems
- Scan intent sections
- Study how pattern interrelate
- Study patterns of like purpose
- Examine a cause of redesign
- Consider what should be variable in your design

Using a pattern

- Read through the pattern
- Go back and study
  - Structure
  - Participants
  - Collaborations
- Look at the sample code

Using a pattern

- Choose names for participants
  - Meaningful in the application context
- Define the classes
- Choose operation names
  - Application specific
- Implement operations

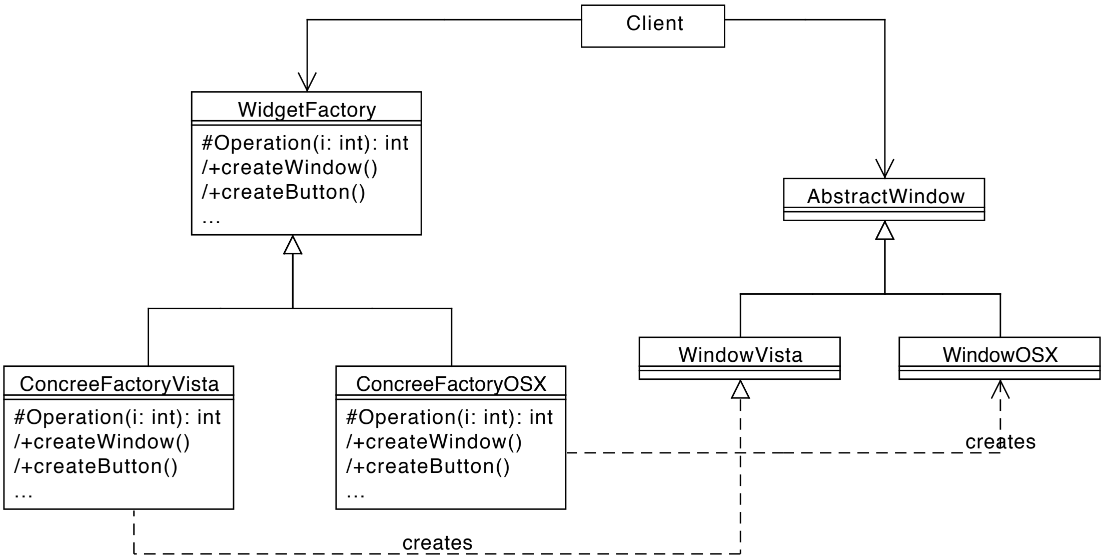
Creational patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Abstract Factory

- Context
  - A family of related classes can have different implementation details
- Problem
  - The client should not know anything about which variant they are using / creating

Abstract Factory Example



Client

WidgetFactory

#Operation(i: int): int
/+createWindow()
/+createButton()
...

AbstractWindow

ConcreeFactoryVista

#Operation(i: int): int
/+createWindow()
/+createButton()
...

ConcreeFactoryOSX

#Operation(i: int): int
/+createWindow()
/+createButton()
...

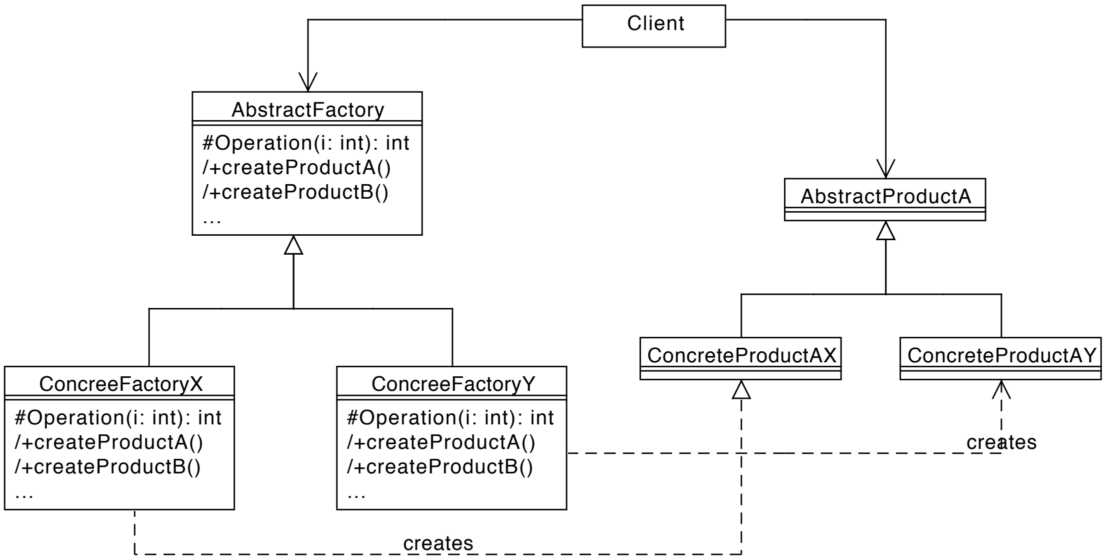
WindowVista

WindowOSX

creates

creates

Abstract Factory



Client

AbstractFactory

#Operation(i: int): int
/+createProductA()
/+createProductB()
...

AbstractProductA

ConcreeFactoryX

#Operation(i: int): int
/+createProductA()
/+createProductB()
...

ConcreeFactoryY

#Operation(i: int): int
/+createProductA()
/+createProductB()
...

ConcreteProductAX

ConcreteProductAY

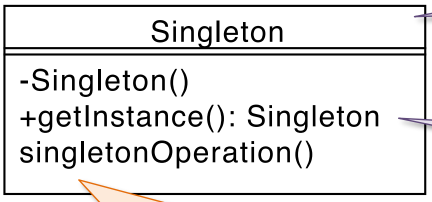
creates

creates

Singleton

- Context:
  - A class represents a concept that requires a single instance
- Problem:
  - Clients could use this class in an inappropriate way

Singleton Pattern



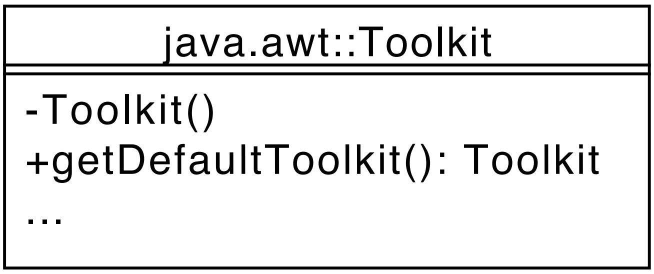
Singleton class

Instantiation static method

```
private Singleton() { }
private static Singleton instance;
public static Singleton getInstance(){
  if(instance==null)
       instance = new Singleton();
  return instance;
}
```

Singleton Example

- **`java.awt.Toolkit`**
  - Singleton + FactoryMethod



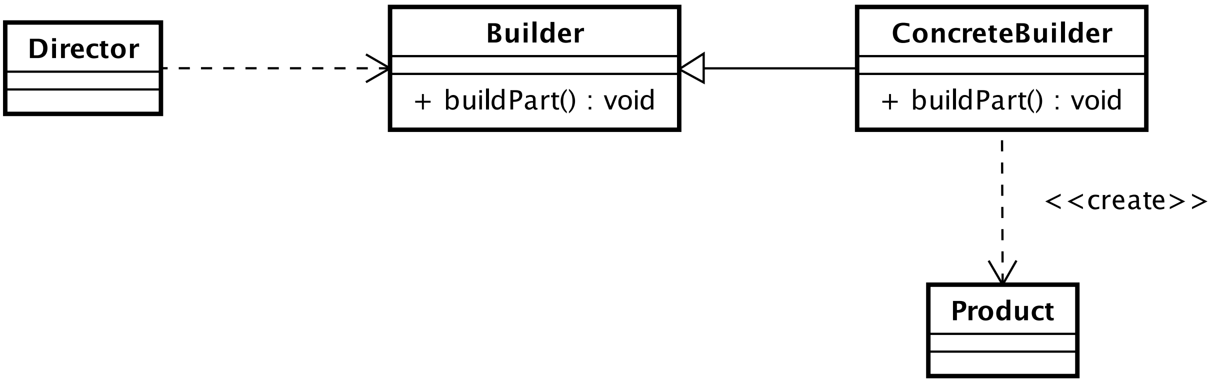
java.awt::Toolkit

-Toolkit()
+getDefaultToolkit(): Toolkit
...

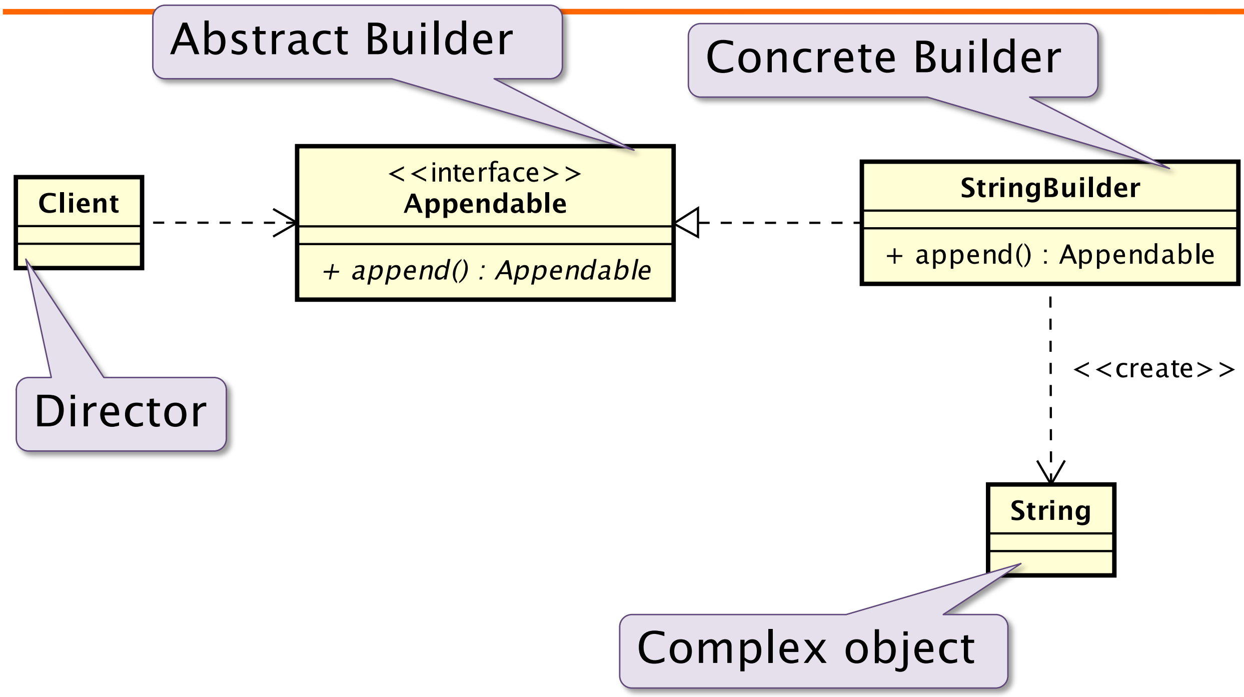
Builder object

- Context
  - An object of a complex class has to be created
- Problem
  - The creation entails complex interaction with the object
  - Different variation of the target object might be created

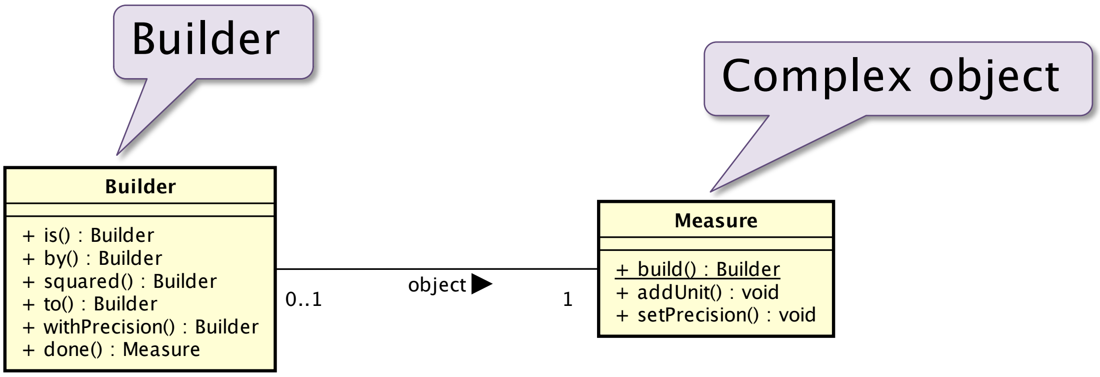
Builder Pattern



Builder example



Example Measure builder



Note: Simplified version w.r.t. GoF

Structural patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.

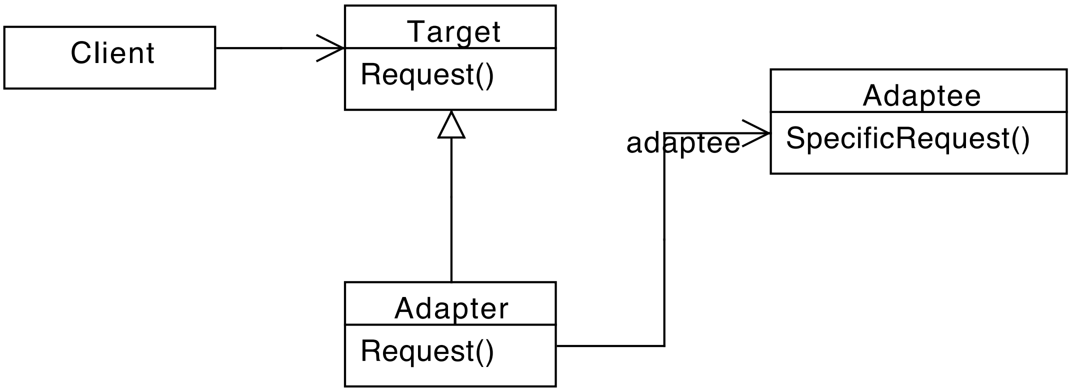
GoF structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

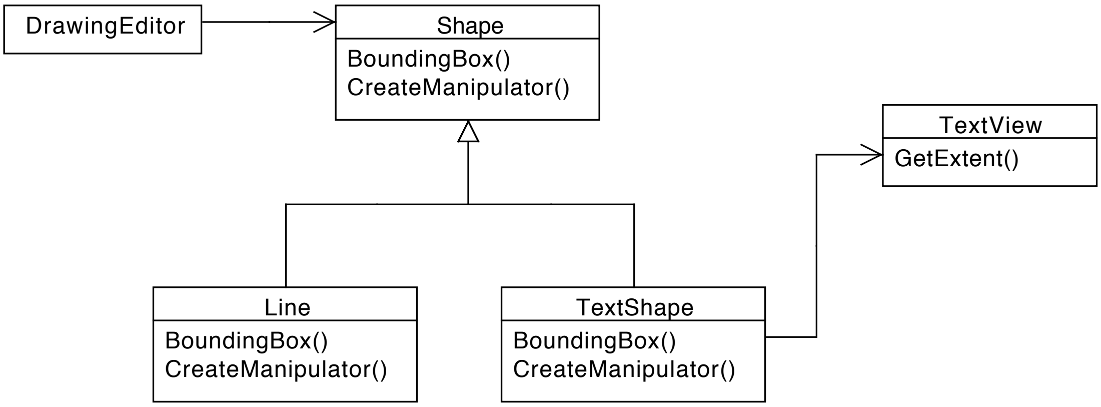
Adapter

- Context:
  - A class provides the features required by another class but its interface is not the one expected
- Problem:
  - The integration of the provider class should be possible without modifying it
    - Its source code could be not available
    - It is already used as it is somewhere else

Adapter



Adapter example



Java Listener Adapter

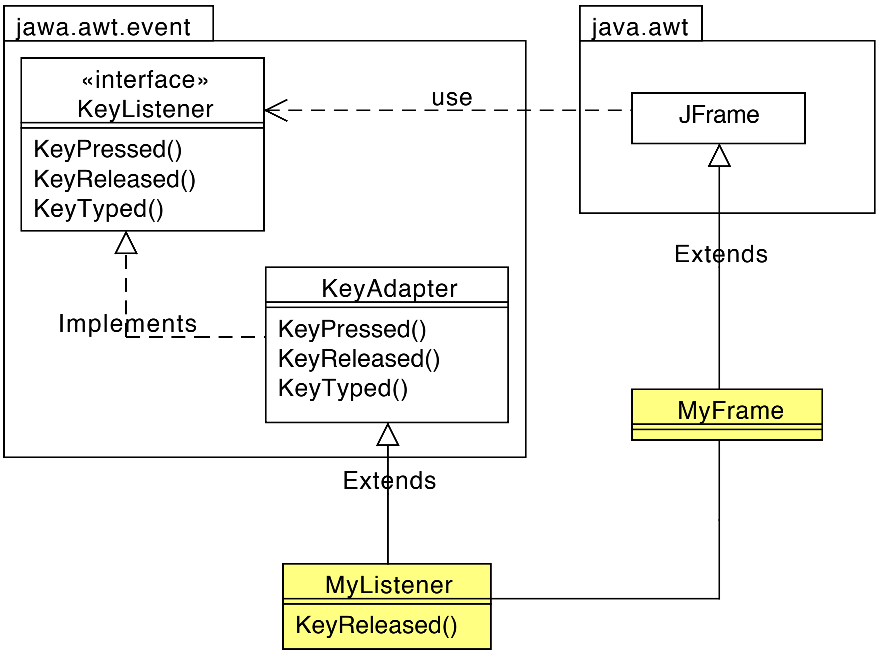
- In Java GUI, events are handled by Listeners
- Listener classes need to implement Listener interfaces
  - Include several methods
  - They all should be implemented

Java Listener Adapter

```
class MyListener{
  public void KeyPressed(..){}
  public void KeyReleased(..){
    // … handle event
  }
  public void KeyTyped(..){} }
```

```
class MyListener{
  public void KeyReleased(..){
    // … handle event
  }
}
```

Java Listener Adapter



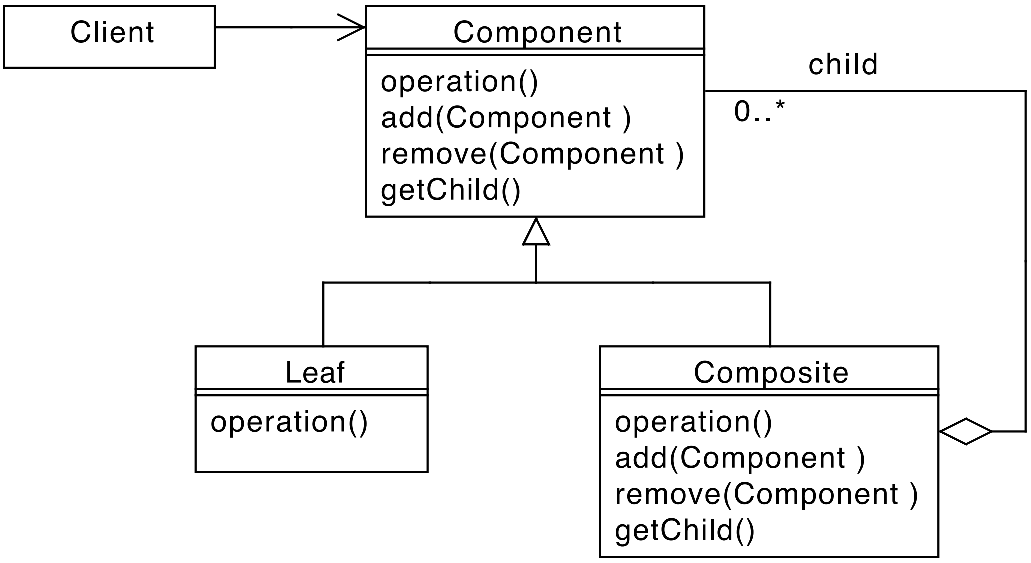
Structural Class Patterns

- Adapter pattern
  - Inheritance plays a fundamental role
  - Only example of structural class pattern

Composite

- Context:
  - You need to represent part-whole hierarchies of objects
- Problem
  - Clients are complex
  - Difference between composition objects and individual objects.

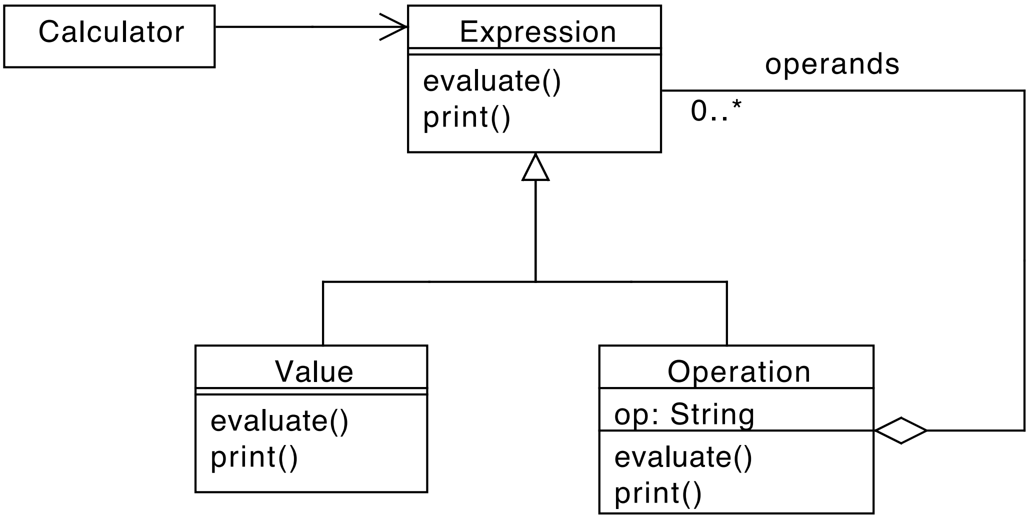
Composite



Composite Example

- Arithmetic expressions representation
  - Operators
  - Operands
- Evaluation of expressions

Composite Example



Composite Example

```
abstract class Expression {
  public abstract int evaluate();
  public abstract String print();
}
```

Composite Example

```
class Value {
  private int value;

  public Value(int v){
    value = v;
  }
  public int evaluate(){
  return value;
  }
  public String print(){
    return new String(value);
  }
}
```

Composite Example

```
class Operation {
  private char op; // +, -, *, /
  private Expression left, right

  public Operation(char op,
  Expression l, Expression r){
    this.op = op;
    left = l;
    right= r;
  }
…
```

Composite Example

```
class Operation {
…
  public evaluate(){
    switch(op){
      case ‘+’: return
       left.evaluate() +
         right.evaluate();
    break;
      …
    }
  }
…
```

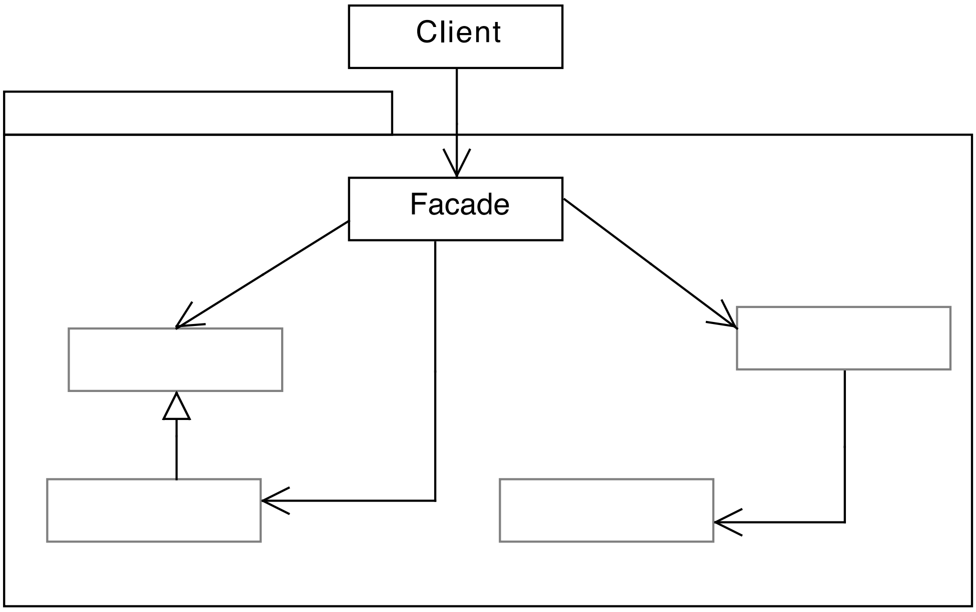
Composite Example

```
class Operation {
…
  public print(){
    return left.print() + op +
         right.print();
  }
}
```

Facade

- Context
  - A functionality is provided by a complex group of classes (interfaces, associations, etc.)
- Problem
  - How is it possible to use the classes without being exposed to the details

Facade



Behavioral patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Not just patterns of objects or classes but also the patterns of communication.
  - Complex control flow that's difficult to follow at run-time.
  - Shift focus away from flow of control to let concentrate just on the way objects are interconnected.

GoF behavioral patterns

Object-level

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

Class-level

- Template Method
- Interpreter

Mechanisms

- Encapsulating variation
- Objects as arguments
- Information circulation policies
- Sender and Receiver decoupling

Encapsulating Variation

- A varying aspect of a program
- Captured by an object
  - Other delegate operations to the “variant” object

Argument Objects

- Often an object is passed as argument
  - Hides complexity from clients
  - Concentrate the “active” code in one class

Information circulation

- Responsibility of how to circulate information may be:
  - Distributed among different parties.
  - Encapsulated in a single object.

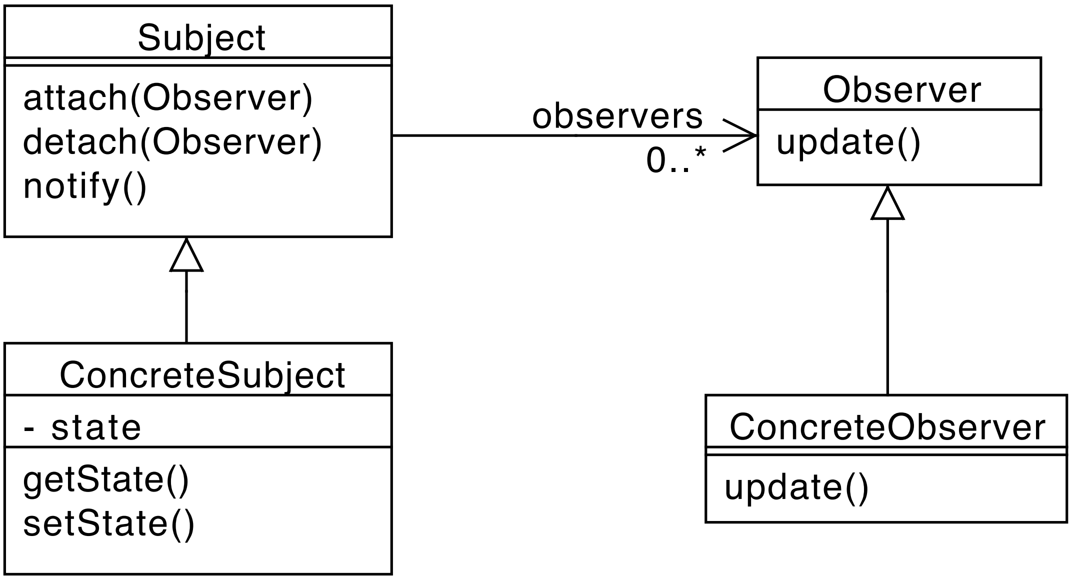
Communication decoupling

- Decoupling senders and receivers is a key to:
  - Reduce coupling
  - Improve reusability
  - Enforce layering and structure

Observer

- Context:
  - The change in one object may influence one or more other objects
- Problem
  - High coupling
  - Number and type of objects to be notified may not be known in advance

Observer



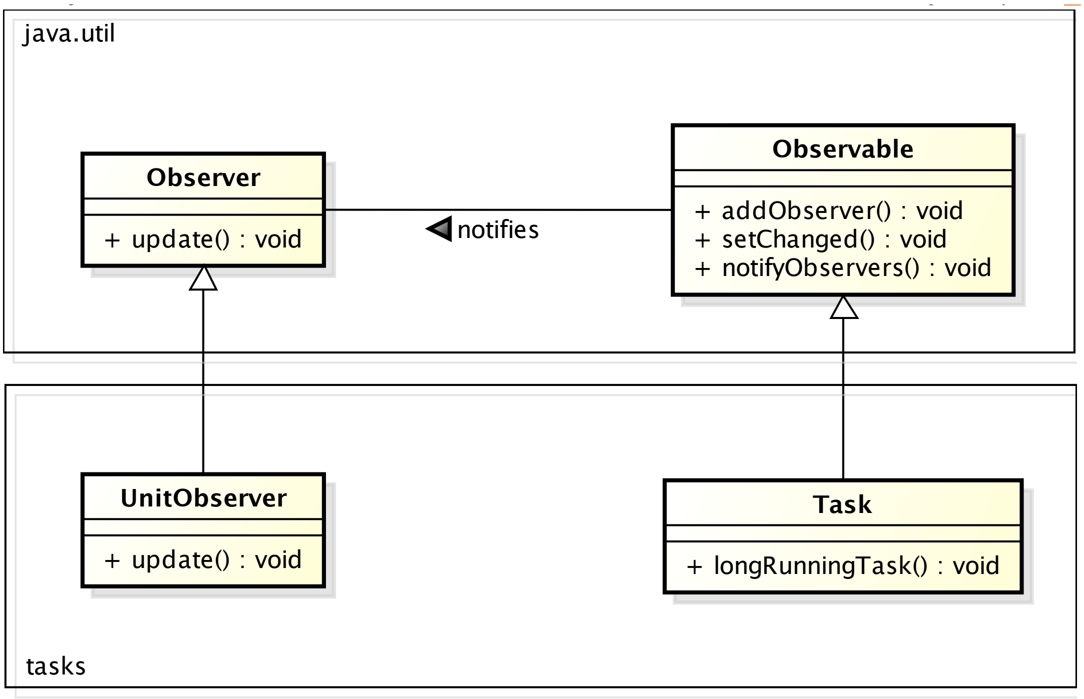
Observer – Consequences

+ Abstract coupling between Subject and Observer
+ Support for broadcast communication
– Unanticipated updates

Observer–Observable

- Allow a standardized interaction between an objects that needs to notify one or more other objects
- Defined in package **`java.util`**
- Class **`Observable`**
- Interface **`Observer`**

Observer-Observable



Java Observer-Observable

```
class Observable{
 void addObserver(..){}
 void deleteObserver(..){}
 void deleteObservers(){}
 int countObservers() {}
 void setChanged() {}
 void clearChanged() {}
 boolean hasChanged() {}
 void notifyObservers() {}
 void notifyObservers(..) {}
}
```

Observer-Observable

- Class **`Observable`** manages:
  - registration of interested observers by means of method **`addObserver()`**
  - sending the notification of the status change to the observer(s) together with additional information concerning the status (event object).
- Interface **`Observer`** allows:
  - Receiving standardized notification of the observer change of state through method **`update()`** accepts two arguments:
    - Observable object that originated the notification
    - additional information (the event object)

Observer-Observable

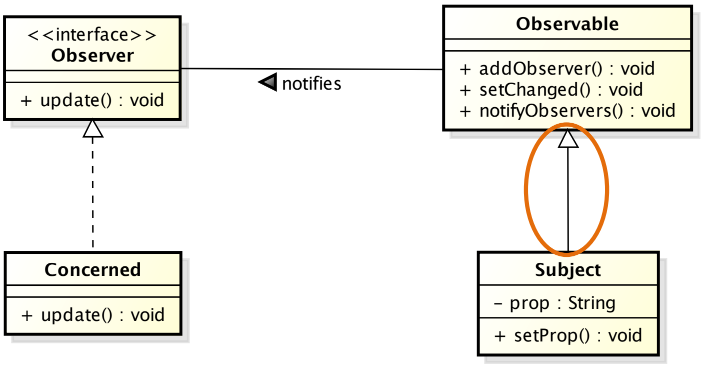
- Sending a notification from an observable element involves two steps:
  - record the fact the the status of the Observable has changed, by means of method **`setChanged()`**,
  - send the actual notification while providing the additional information (the event object), by means of method **`notifyObservers()`**

Inheritance vs. composition

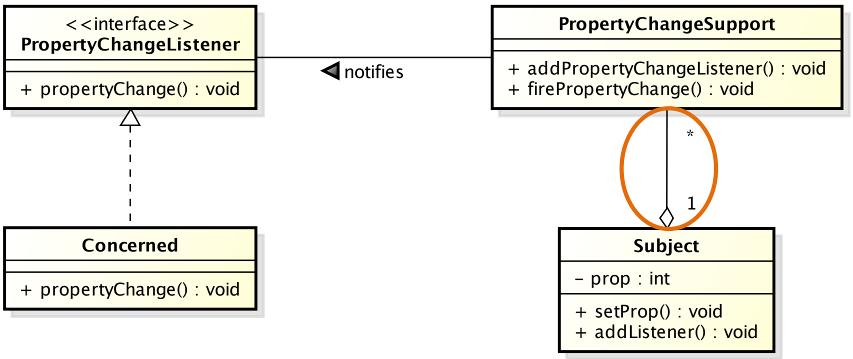
Reuse can be achieved via:

- Inheritance
  - The reusing class has the reused methods available as own methods.
  - Clients can invoke directly inherited methods
- Composition
  - The reusing class has the reused methods available in an included object (attribute)
  - The reusing class must provide methods that accept clients requests and delegate to the included object

Inheritance vs. Composition

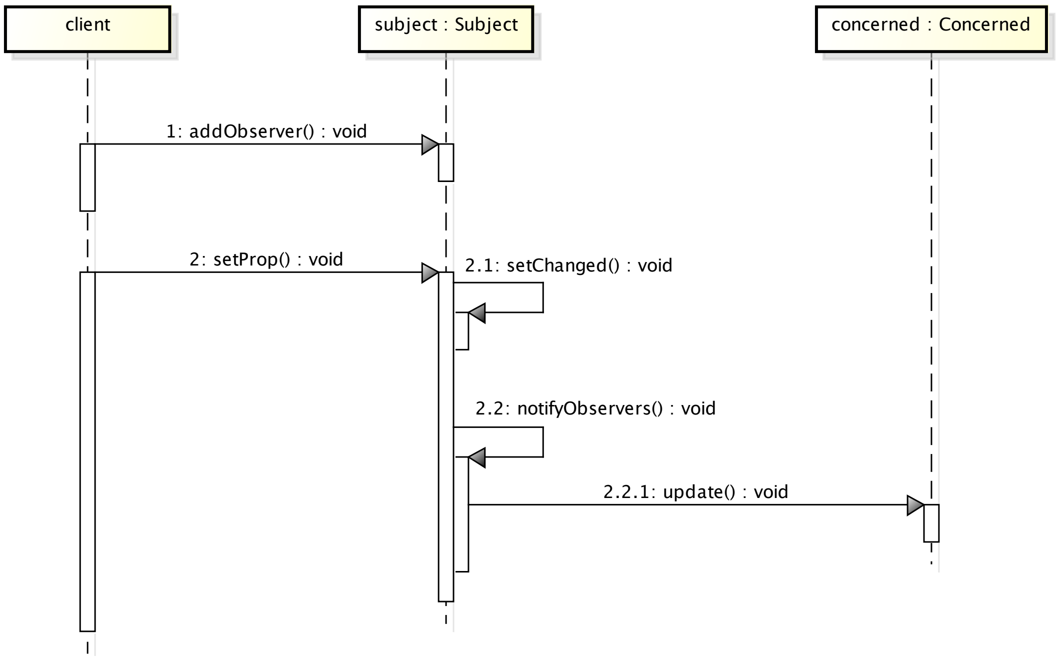


Inheritance

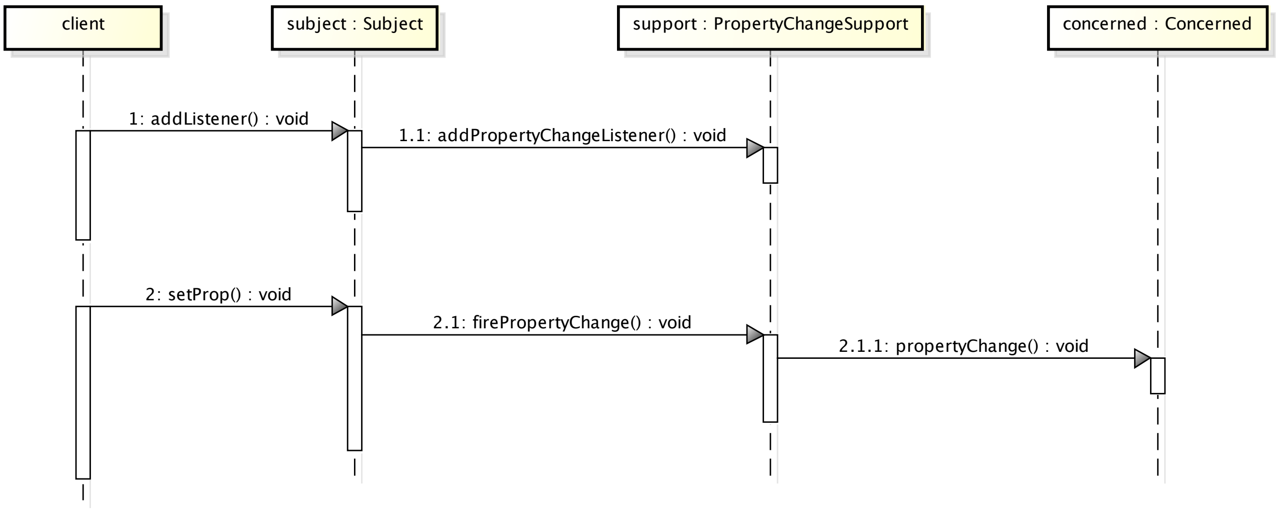


Composition

Observer w/Inheritance



Observer w/Composition



client
subject : Subject
support : PropertyChangeSupport
concerned : Concerned
1: addListener() : void
1.1: addPropertyChangeListener() : void
2: setProp() : void
2.1: firePropertyChange() : void
2.1.1: propertyChange() : void

Observer subject w/inheritance

```
public class Subject
      extends Observable {

  String  prop="ini";

  public void setProp(String val){
    setChanged();
    property = val;
    notifyObservers("theProp");
  }




}
```

Observer subject w/composition

```
public class Subject {
  PropertyChangeSupport pcs =
        new PropertyChangeSupport(this);
  String  prop=”ini";

  public void setProp(String val) {
    String old = property;
    property = val;
    pcs.firePropertyChange("theProp”,old,val);
  }
  // delegation:
  public void addObs(PropertyChangeListener l){
    pcs.addPropertyChangeListener("theProp”,l);
} }
```

Observer with inheritance

```
public class Concerned
     implements Observer {

  @Override
  public void update(Observable src,
                    Object arg) {
    System.out.println("Variation of " +
                     arg);
}

}
```

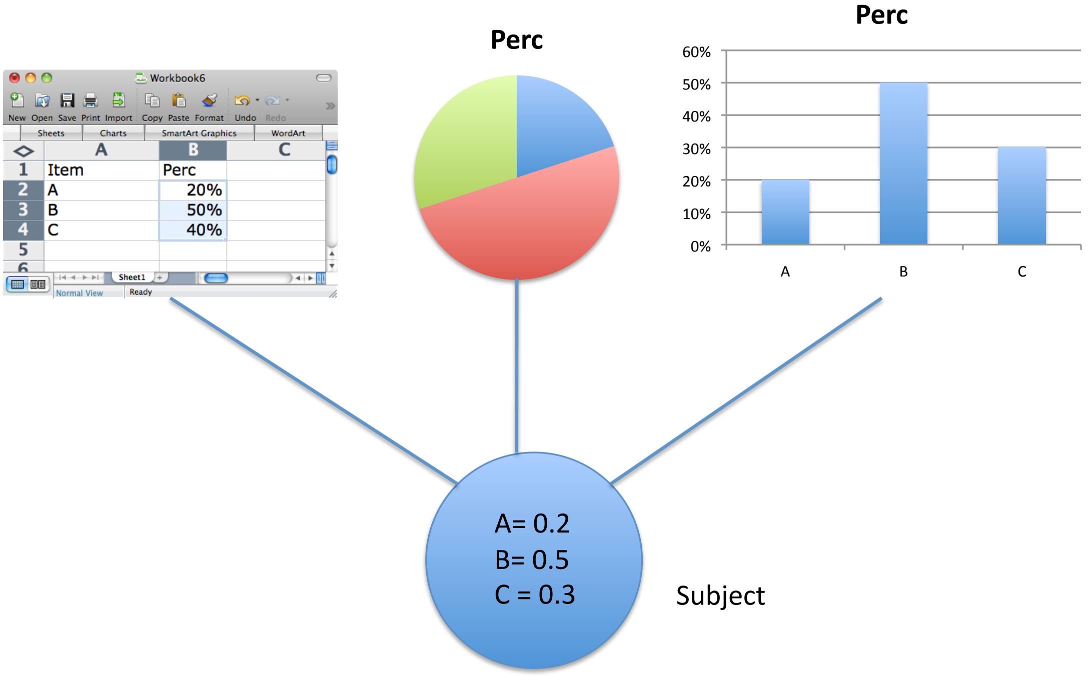
Observer with composition

```
public class Concerned
       implements PropertyChangeListener {

  @Override
  public void propertyChange(
               PropertyChangeEvent evt) {
    System.out.println("Variation of " +
                      evt.getPropertyName());
}

}
```

Observer Example


Workbook6
New Open Save Print Import Copy Paste Format Undo Redo
Sheets Charts SmartArt Graphics WordArt
Sheet1
Normal View Ready
Perc
Perc
60%
50%
40%
30%
20%
10%
0%
A
B
C

	A	B
1	Item	Perc
2	A	20%
3	B	50%
4	C	40%

Observers

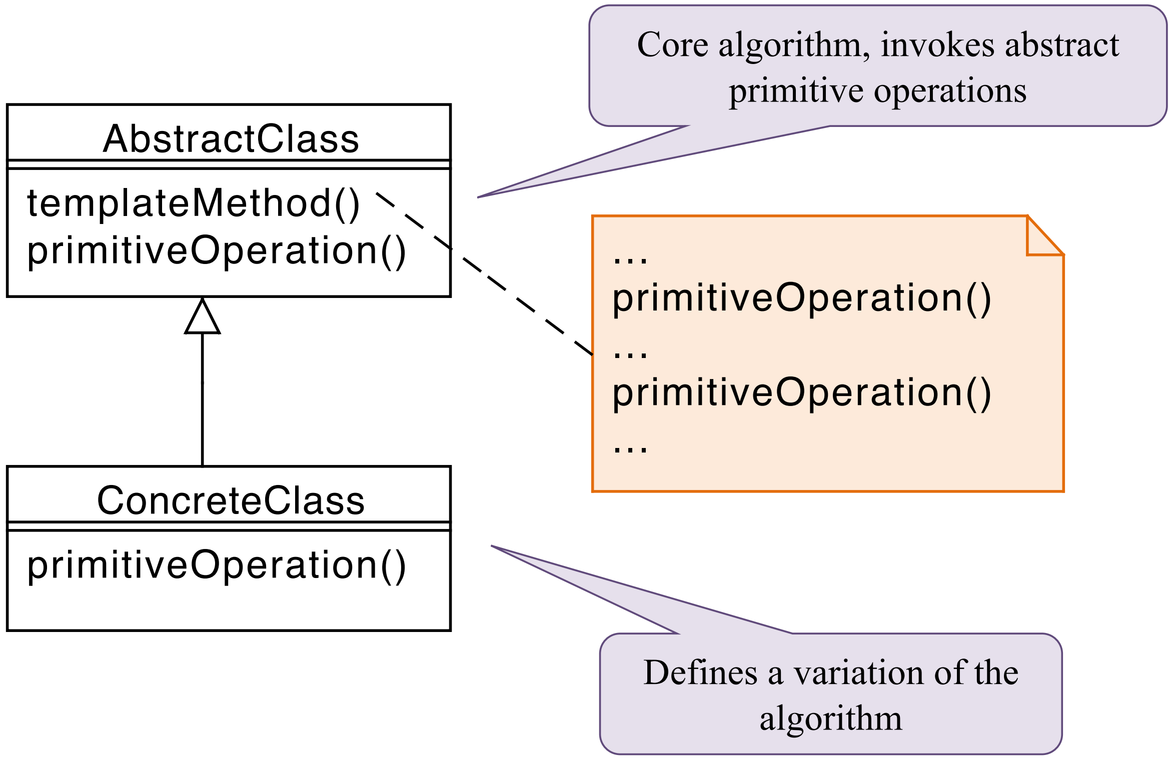
A= 0.2
B= 0.5
C = 0.3

Subject

Template Method

- Context:
  - An algorithm/behavior has a stable core and several variation at given points
- Problem
  - You have to implement/maintain several almost identical pieces of code

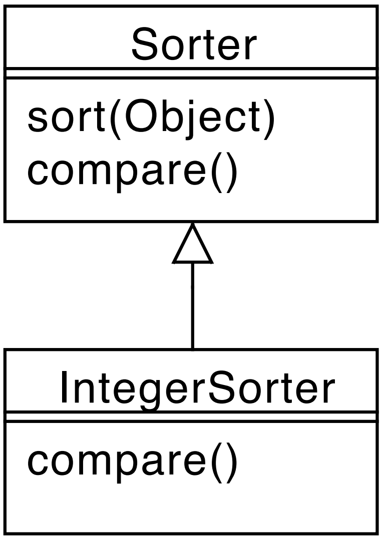
Template Method



Core algorithm, invokes abstract primitive operations

Defines a variation of the algorithm

Template Method Example



Example: Sorter

```
public abstract class Sorter {
  public void sort(Object v[]){
    for(int i=1; i<v.length; ++i)
      for(int j=0; j<v.length-i; ++j){
        if(compare(v[j],v[j+1])>0){
          Object o=v[j];
          v[j]=v[j+1]; v[j+1]=o;
    } } }
  }
  abstract int compare(Object a, Object b);
}
```

Example: StringSorter

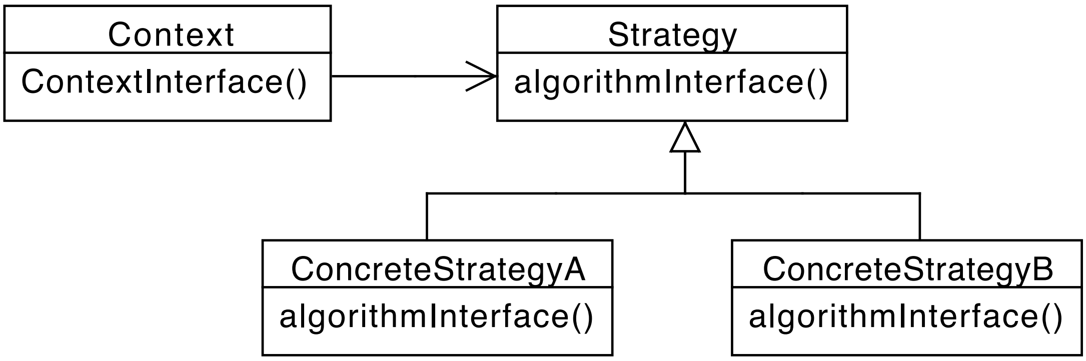
```
class StringSorter extends Sorter {
  int compare(Object a, Object b){
    String sa=(String)a;
    String sb=(String)b;
    return sa.compareTo(sb);
  }
}
```

```
Sorter ssrt = new StringSorter();
String[] v={"g","t","h","n","j","k"};
ssrt.sort(v);
```

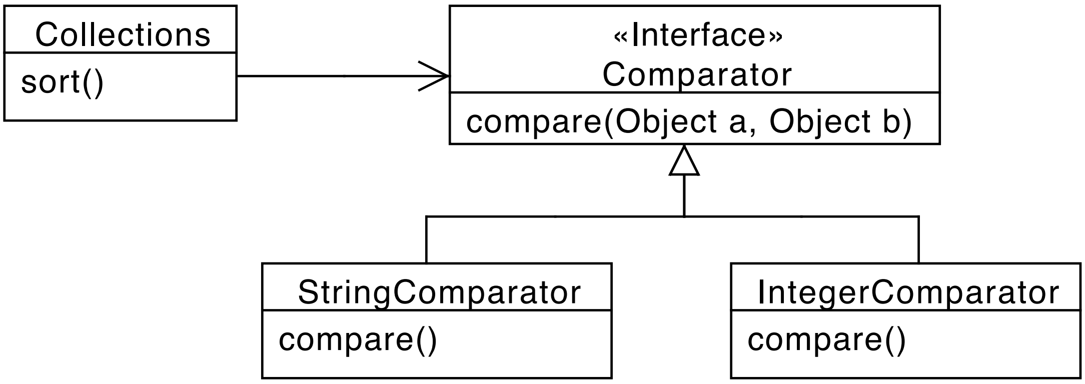
Strategy

- Context
  - Many classes or algorithm has a stable core and several behavioral variations
- Problem
  - Several different implementations are needed.
  - Multiple conditional constructs tangle the code.

Strategy



Strategy example: **Comparator**



Comparator

- Interface **java.util.Comparator**

```
public interface Comparator<T>{
  int compare(T a, T b);
}
```

- Semantics (as comparable): returns
  - a negative integer if **a** precedes **b**
  - 0, if **a** equals **b**
  - a positive integer if **a** succeeds **b**

Comparator

```
class StudentCmp
      implements Comparator<Student>{
public int compare(Student a,Student b){
    return a.id - b.id;
  }
}
```

```
Student[] sv = { new Student(11),
                 new Student(3),
                 new Student(7)};
Arrays.sort(sv, new StudentCmp());
```

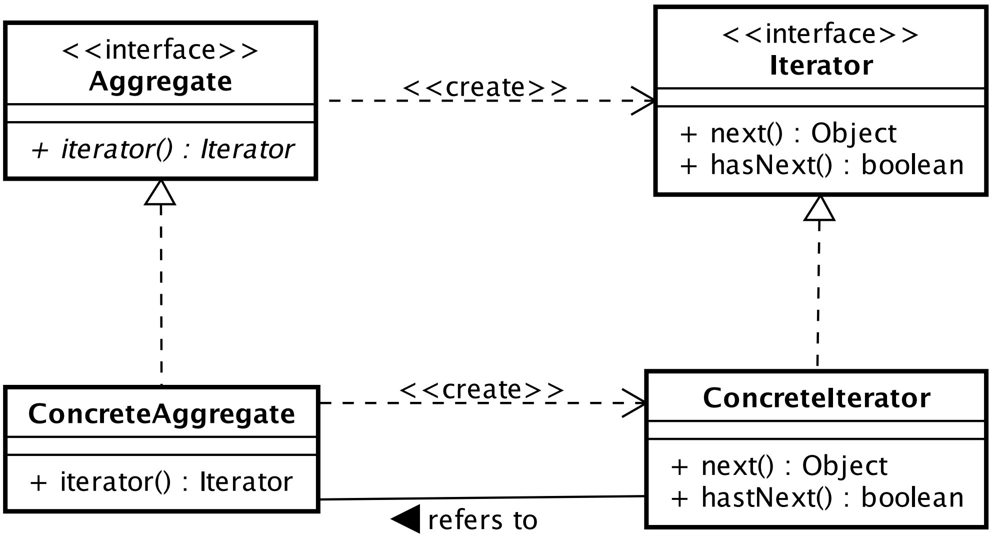
Strategy Consequences

+ Avoid conditional statements
+ Algorithms may be organized in families
+ Choice of implementations
+ Run-time binding
- Clients must be aware of different strategies
- Communication overhead
- Increased number of objects

Iterator pattern

- Context
  - A collection of objects must be iterated
- Problem
  - Multiple concurrent iterations are possible
  - The internal storage must not be exposed
- Solution
  - Provide an iterator object, attached to the collection, that can be advanced independently

Iterator pattern



<<interface>>
Aggregate
+ iterator() : Iterator

<<create>>

<<interface>>
Iterator
+ next() : Object
+ hasNext() : boolean

ConcreteAggregate
+ iterator() : Iterator

<<create>>

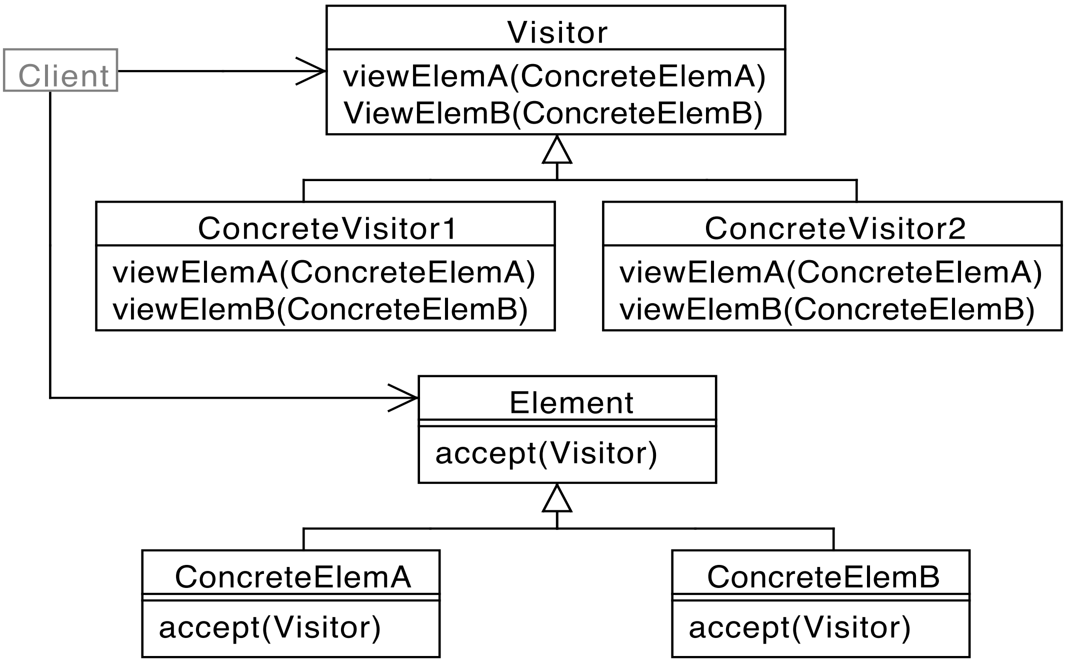
ConcreteIterator
+ next() : Object
+ hastNext() : boolean

◀ refers to

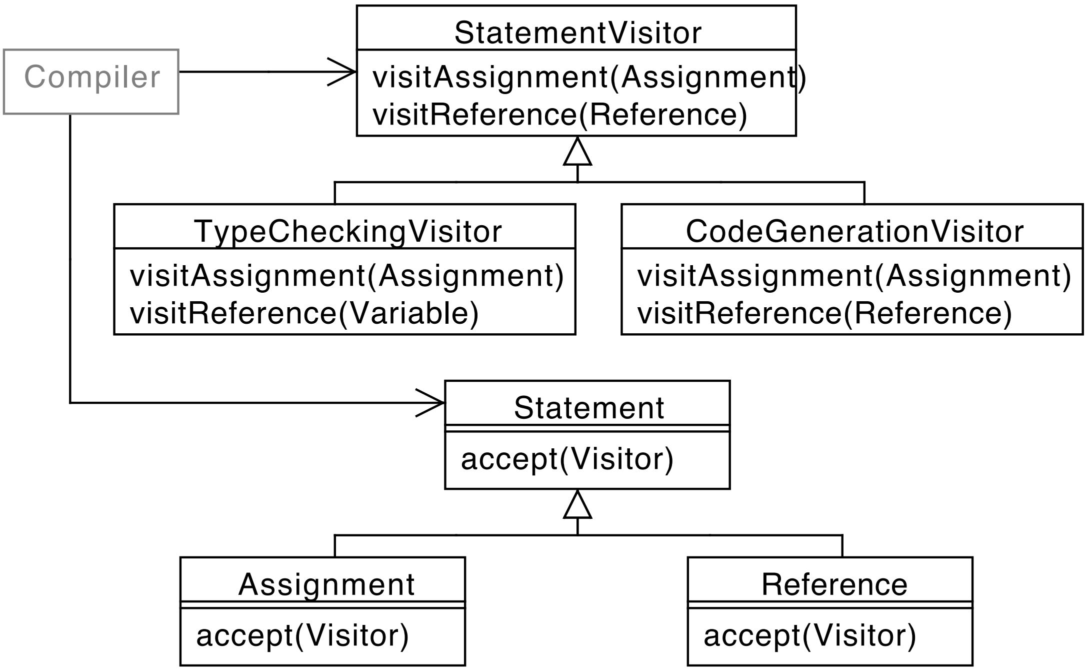
Visitor

- Context
  - An object structure contains many classes with differing interfaces.
  - Many different operations need to be performed on the objects
- Problem
  - The operations on the objects depend on their concrete classes
  - Classes could be polluted with several operations

Visitor



Visitor Example



Visitor Consequences

+ Adding new operations is very easy
+ Behavior is partitioned
+ Can visit class hierarchies
+ State can be accumulated

– Difficult to add new concrete elements
– Break of encapsulation

References

- M.Fowler, K. Scott, UML Distilled, 3rd ed. Addison-Wesley, 2003.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- E.Freeman, E.Freeman, K.Sierra, B.Bates. Head First Design Patterns, O'Reilly, 2004