More Design Patterns

EECS 3311

Ilir Dema

demailir@eecs.yorku.ca

Acknowledge

- Some of the covered materials are based on SOEN 344 at Concordia, SE463 at UW, SENG321 at UVic, CSE331 at University of Washington and previous EECS3311 offerings:
  - Nikolaos Tsantalis, Jo Atlee, Marty Stepp, Mike Godfrey, Jonathan S. Ostroff, M. Ernst, S. Reges, D. Notkin, R. Mercer, Davor Svetinovic, Jack Jiang, **Jackie Wang**

YORK UNIVERSITÉ UNIVERSITY

Outlines

- **Java Design Patterns**
  - Bridge Pattern
  - Visitor Design Pattern
  - Composite Pattern
  - Observer design patterns

Pattern: Bridge

What’s It All About?

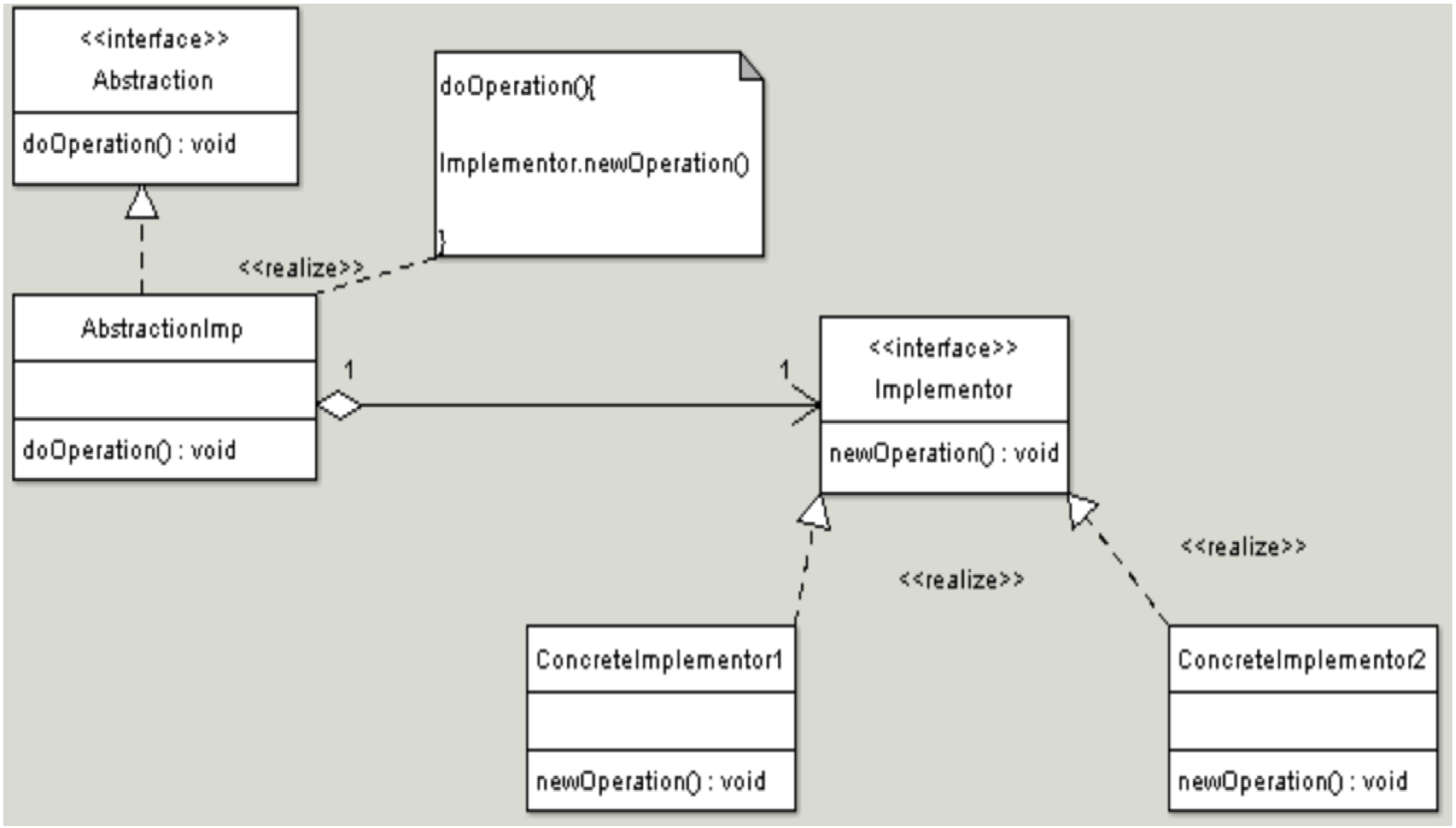
- The intent of this pattern is to decouple abstraction from implementation so that the two can vary independently.

- Sometimes an abstraction should have different implementations; for example, a GUI app may have a Windows implementation and a Unix implementation.

Applicability

- The bridge pattern applies when there is a need to avoid permanent binding between an abstraction and an implementation and when the abstraction and implementation need to vary independently.

- Using the bridge pattern would leave the client code unchanged with no need to recompile the code.
  - Thus, helps make the design open to extension, and closed to modification.


<<interface>>
Abstraction
doOperation() : void
doOperation(){
Implementor.newOperation()
}
<<realize>>
AbstractionImp
doOperation() : void
1
1
<<interface>>
Implementor
newOperation() : void
<<realize>>
<<realize>>
ConcreteImplementor1
newOperation() : void
ConcreteImplementor2
newOperation() : void

Implementation

- The participants classes in the bridge pattern are:
- **Abstraction** - Abstraction defines abstraction interface.
- **AbstractionImpl** - Implements the abstraction interface using a reference to an object of type Implementor.
- **Implementor** - Implementor defines the interface for implementation classes. This interface does not need to correspond directly to abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementor interface.
- **ConcreteImplementor1, ConcreteImplementor2** - Implements the Implementor interface.

Consequences of the Bridge Pattern

- Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured and even switched at run-time.
- Abstraction and Implementor hierarchies can be extended independently.

Pattern: Composite

What’s It All About?

- There are times when a program needs to manipulate a tree data structure and it is necessary to treat both branches as well as leaf nodes uniformly.
- The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

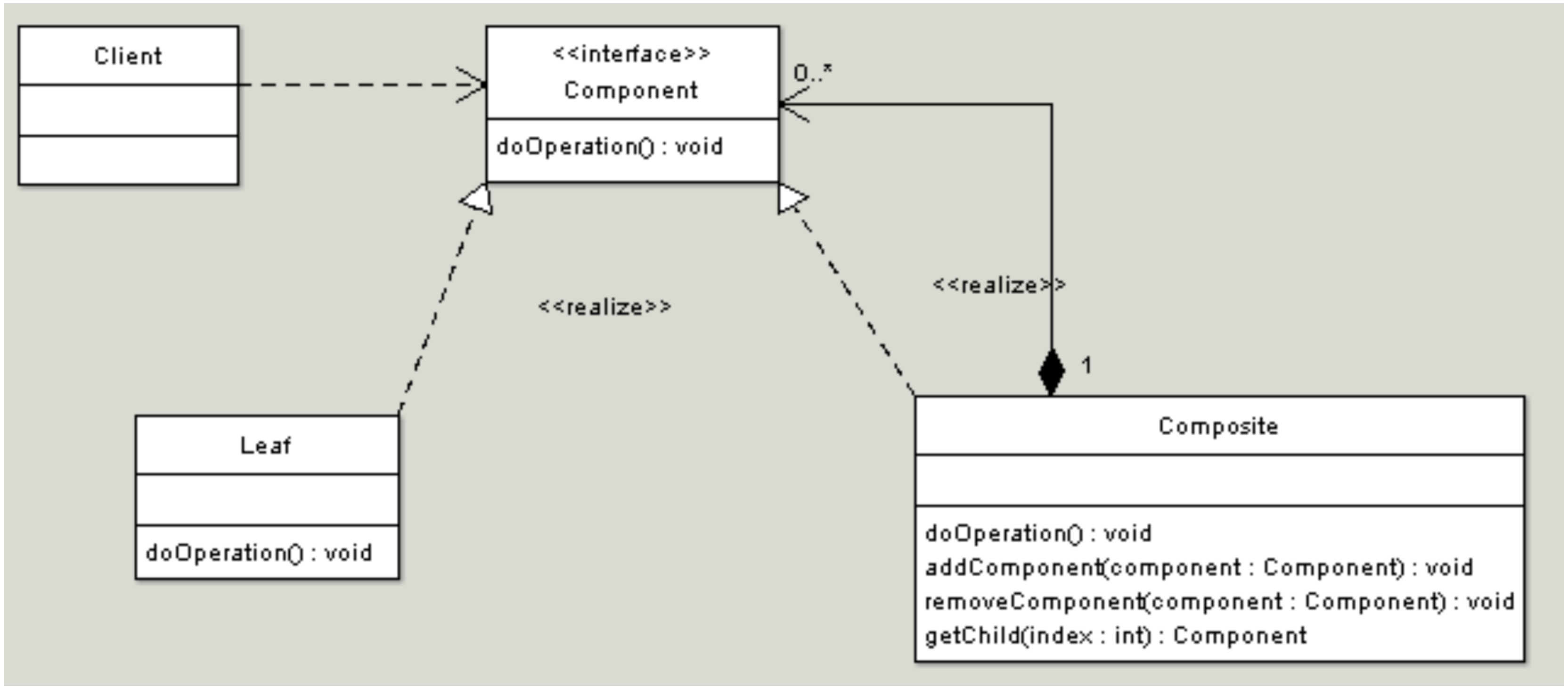
Where Applicable

- When there is a part-whole hierarchy of objects, and a client needs to deal with objects uniformly, regardless of the fact that an object might be a leaf or a branch.

Example – Graphics Drawing Editor

- A shape can be basic or complex.
    - Basic shape: line
    - Complex shape: a rectangle which is made of four line objects.
- Since shapes have many operations in common such as rendering the shape to screen, and since shapes follow a part-whole hierarchy, composite pattern can be used to enable the program to deal with all shapes uniformly.

UML of Composite Pattern



Pattern: Visitor

What’s It All About?

- Allows for new operations to be defined and used on elements of an object structure without changing the contents of those elements.

- The Key is **Double Dispatch**
    - choosing the method to invoke based both on receiver and argument types

Where Applicable

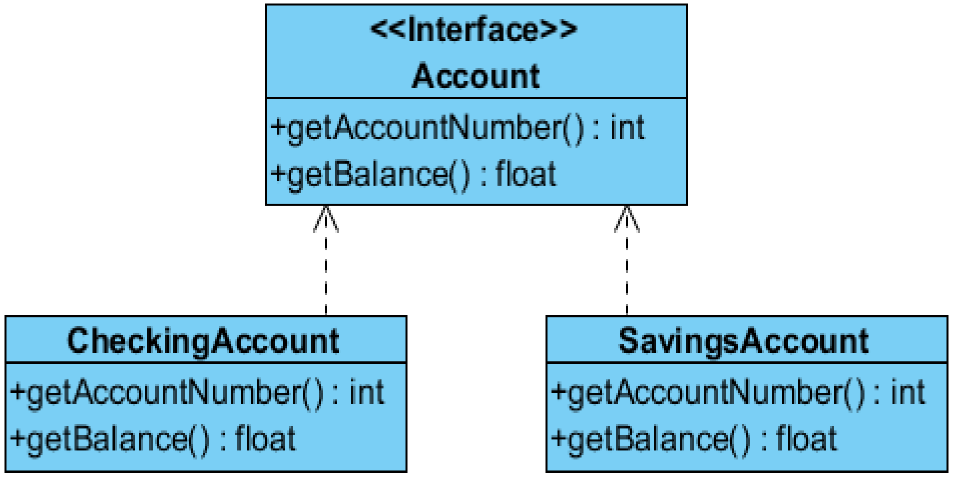
- Rarely Changing Object Structures
- Using Unrelated Operations
- Many Classes with Differing Interfaces

Visitor design pattern

- Add an accept(Visitor) method to the "element" hierarchy
- Create a "visitor" base class w/ a visit() method for every "element" type
- Create a "visitor" derived class for each "operation" to do on "elements"
- Client creates "visitor" objects and passes each to accept() calls

How it Works

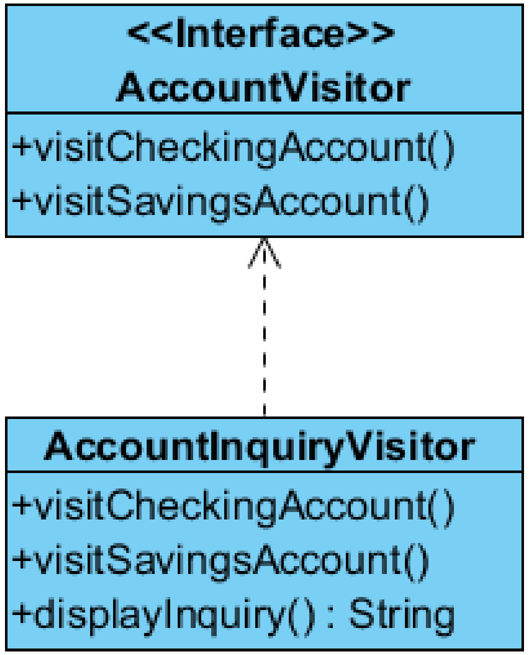
- Concrete Object Structure
- Assume Rarely Changing
- Bank Accounts



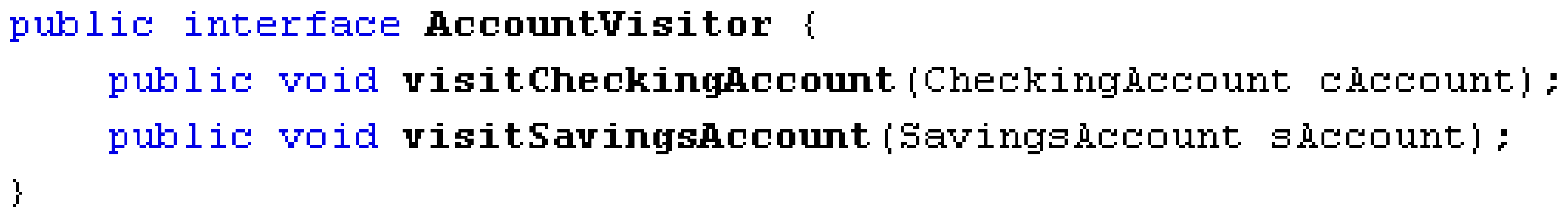
Add an Inquiry Operation

- Check balances and display account summary

- Don’t Want to Change Structure
- Create a Visitor Structure
- Account Visitor



Account Visitor Interface

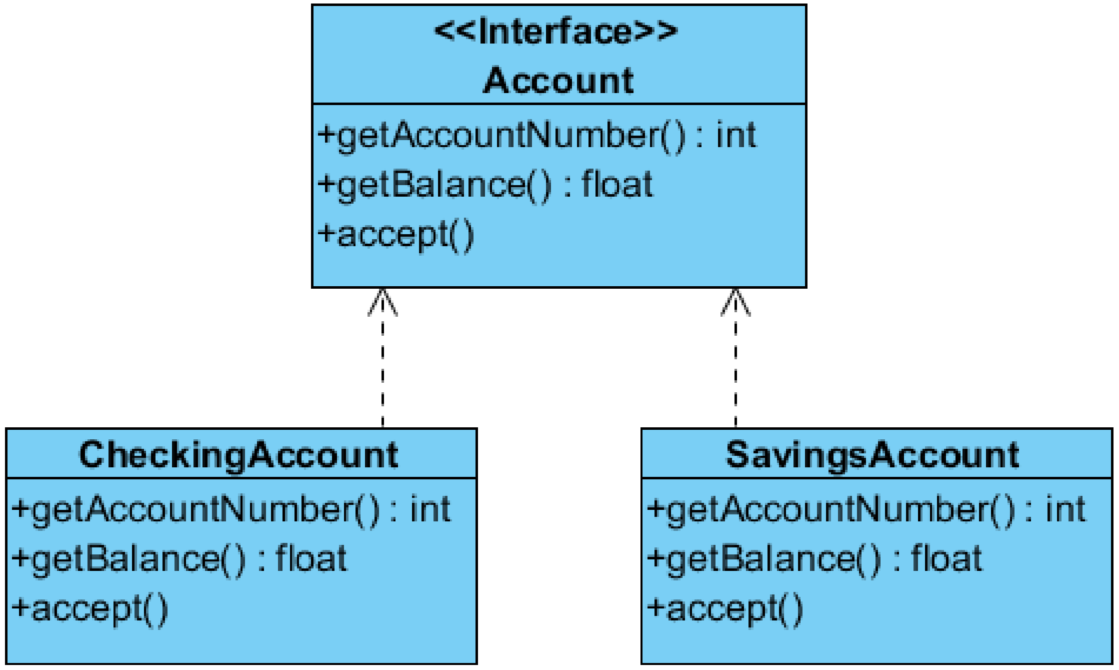


```java
public interface AccountVisitor {
    public void visitCheckingAccount(CheckingAccount cAccount);
    public void visitSavingsAccount(SavingsAccount sAccount);
}
```

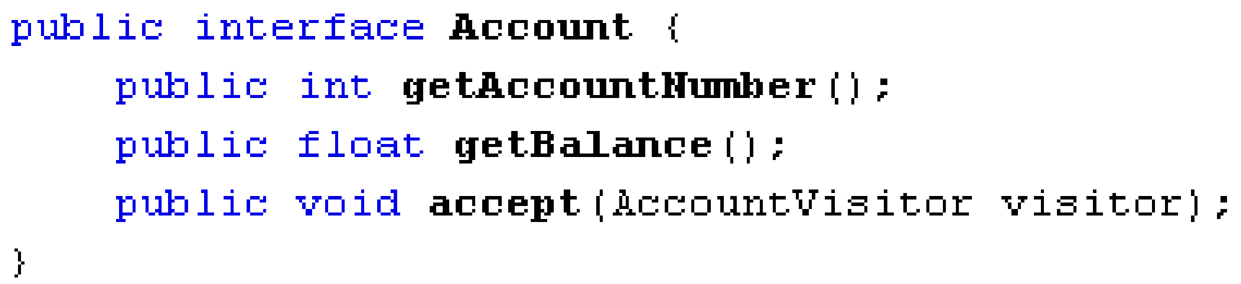
Inquiry Visitor

```java
import java.text.DecimalFormat;

public class AccountInquiryVisitor implements AccountVisitor {
    private String checkAcctBal;
    private int checkAcctNum;
    private String saveAcctBal;
    private int saveAcctNum;
    private DecimalFormat money;

    public AccountInquiryVisitor() {
        checkAcctBal = "";
        checkAcctNum = 0;
        saveAcctBal = "";
        saveAcctNum = 0;
        money = new DecimalFormat("$0.00");
    }
    public void visitCheckingAccount(CheckingAccount cAccount) {
        checkAcctBal = money.format(cAccount.getBalance());
        checkAcctNum = cAccount.getAccountNumber();
    }
    public void visitSavingsAccount(SavingsAccount sAccount) {
        saveAcctBal = money.format(sAccount.getBalance());
        saveAcctNum = sAccount.getAccountNumber();
    }
    public String displayInquiry() {
        String inquiry = "";
        inquiry = inquiry + "Balance Inquiry for All Accounts:\n";
        inquiry = inquiry + "\n";
        inquiry = inquiry + "  Checking " + checkAcctNum + "\n";
        inquiry = inquiry + "  Current Balance: " + checkAcctBal + "\n";
        inquiry = inquiry + "\n";
```

Account Structure Change



Account Interface



```
public interface Account {
    public int getAccountNumber();
    public float getBalance();
    public void accept(AccountVisitor visitor);
}
```

Checking Account

```java
public class CheckingAccount implements Account {
    private int accountNumber;
    private float currentBalance;

    public CheckingAccount(int accountNumber, float initialAmount) {
        this.accountNumber = accountNumber;
        currentBalance = initialAmount;
    }
    public int getAccountNumber() {
        return this.accountNumber;
    }
    public float getBalance() {
        return currentBalance;
    }
    public void accept(AccountVisitor visitor) {
        visitor.visitCheckingAccount(this);
    }
}
```

Savings Account

```
public class SavingsAccount implements Account {
    private int accountNumber;
    private float currentBalance;

    public SavingsAccount(int accountNumber, float initialAmount) {
        this.accountNumber = accountNumber;
        currentBalance = initialAmount;
    }
    public int getAccountNumber() {
        return this.accountNumber;
    }
    public float getBalance() {
        return currentBalance;
    }
    public void accept(AccountVisitor visitor) {
        visitor.visitSavingsAccount(this);
    }
}
```

Main Method

```java
public class Main {
    public static void main(String[] args) {
        CheckingAccount myChecking = new CheckingAccount(10253, 1000);

        SavingsAccount mySavings = new SavingsAccount(10334, 2000);

        AccountInquiryVisitor inquiry = new AccountInquiryVisitor();

        myChecking.accept(inquiry);

        mySavings.accept(inquiry);

        System.out.println(inquiry.displayInquiry());
    }
}
```

Pattern: Observer

Observer Pattern

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

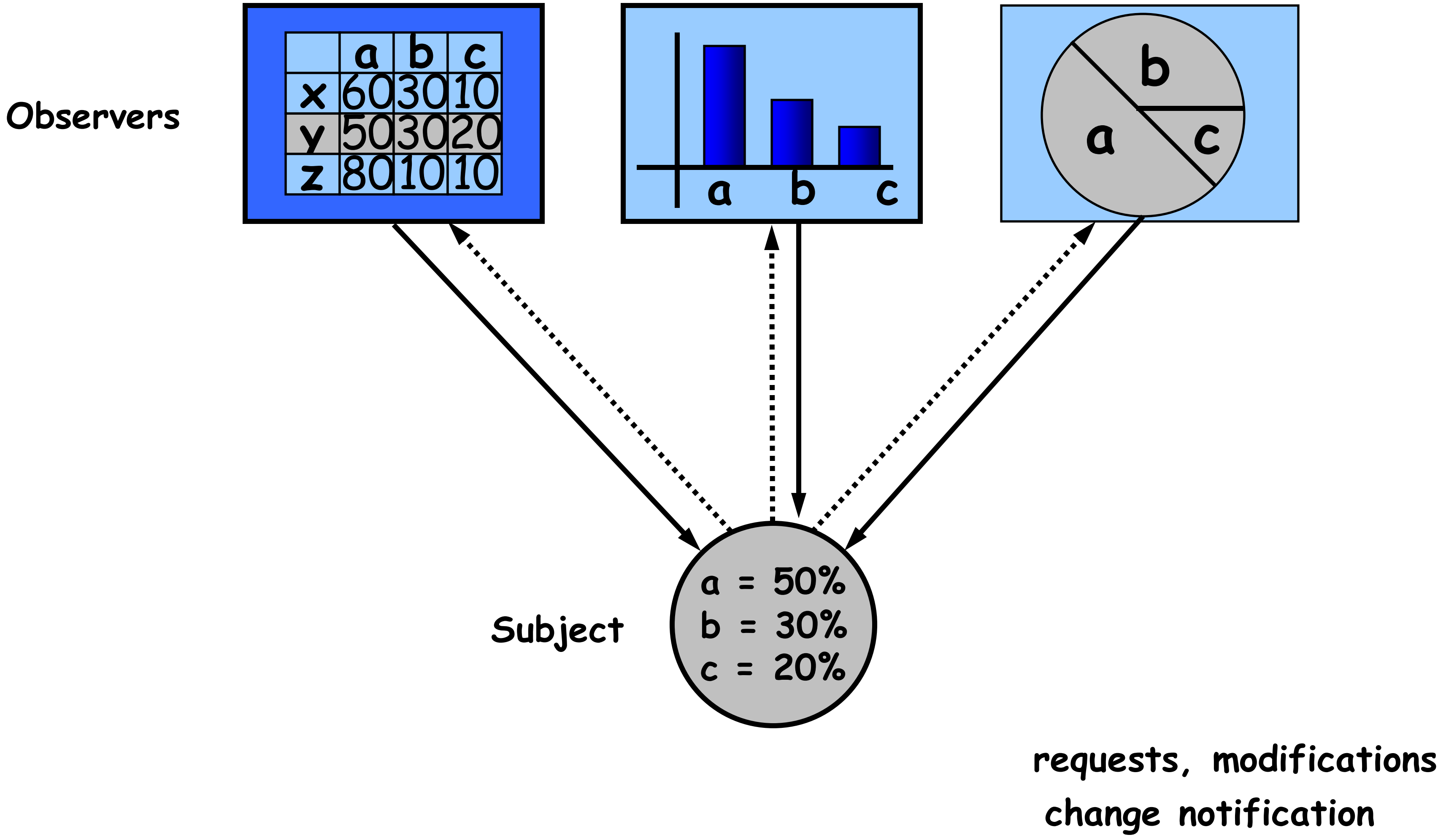
- a.k.a Dependence mechanism / publish-subscribe / broadcast / change-update

Subject & Observer

- Subject
  - the object which will frequently change its state and upon which other objects depend

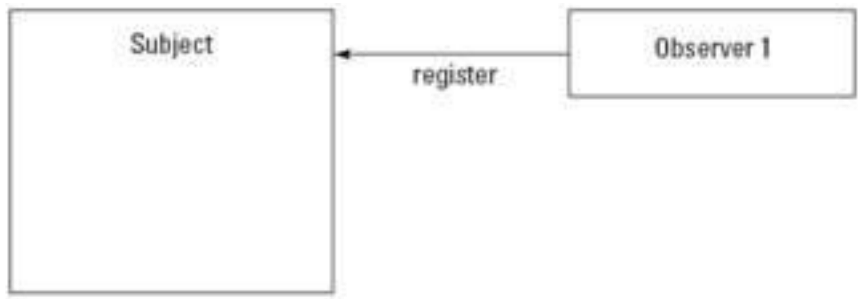
- Observer
  - the object which depends on a subject and updates according to its subject's state.

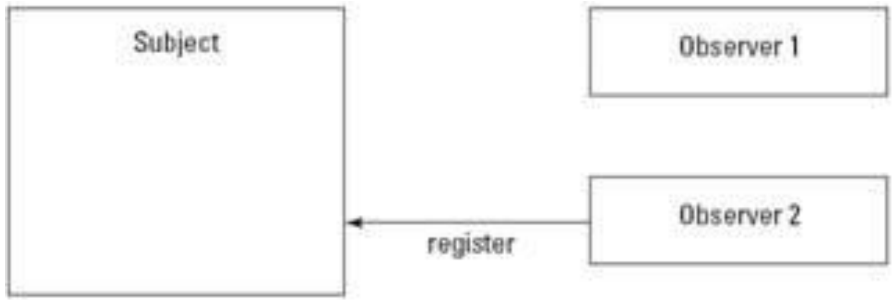
Observer Pattern - Example



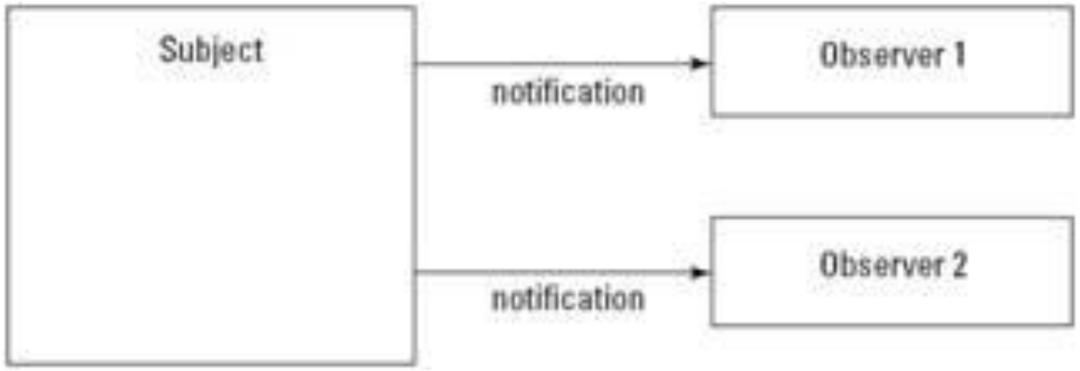
Observer Pattern - Working

A number of Observers "register" to receive notifications of changes to the Subject. Observers are not aware of the presence of each other.





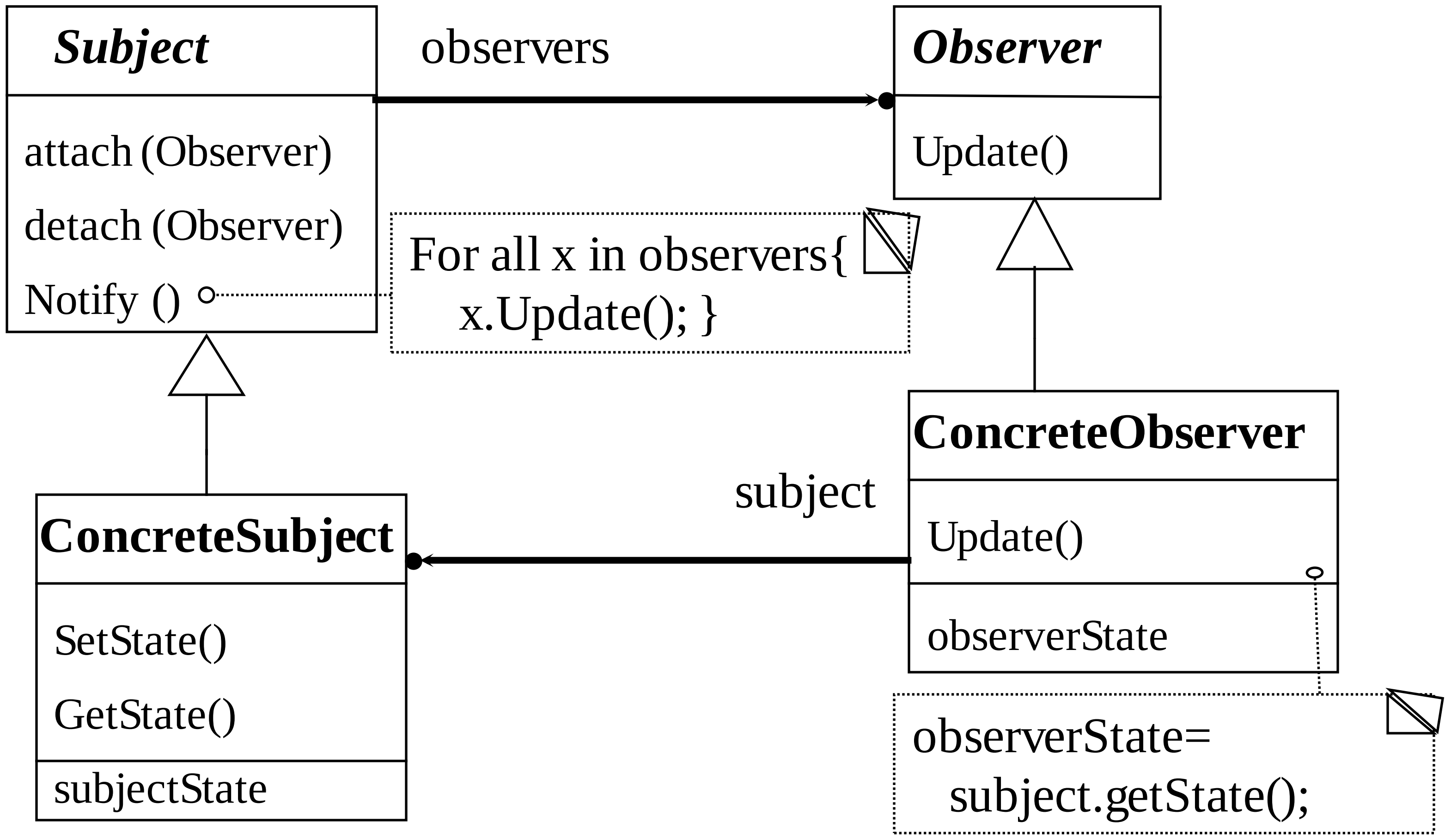
When a certain event or "change" in Subject occurs, all Observers are "notified'.



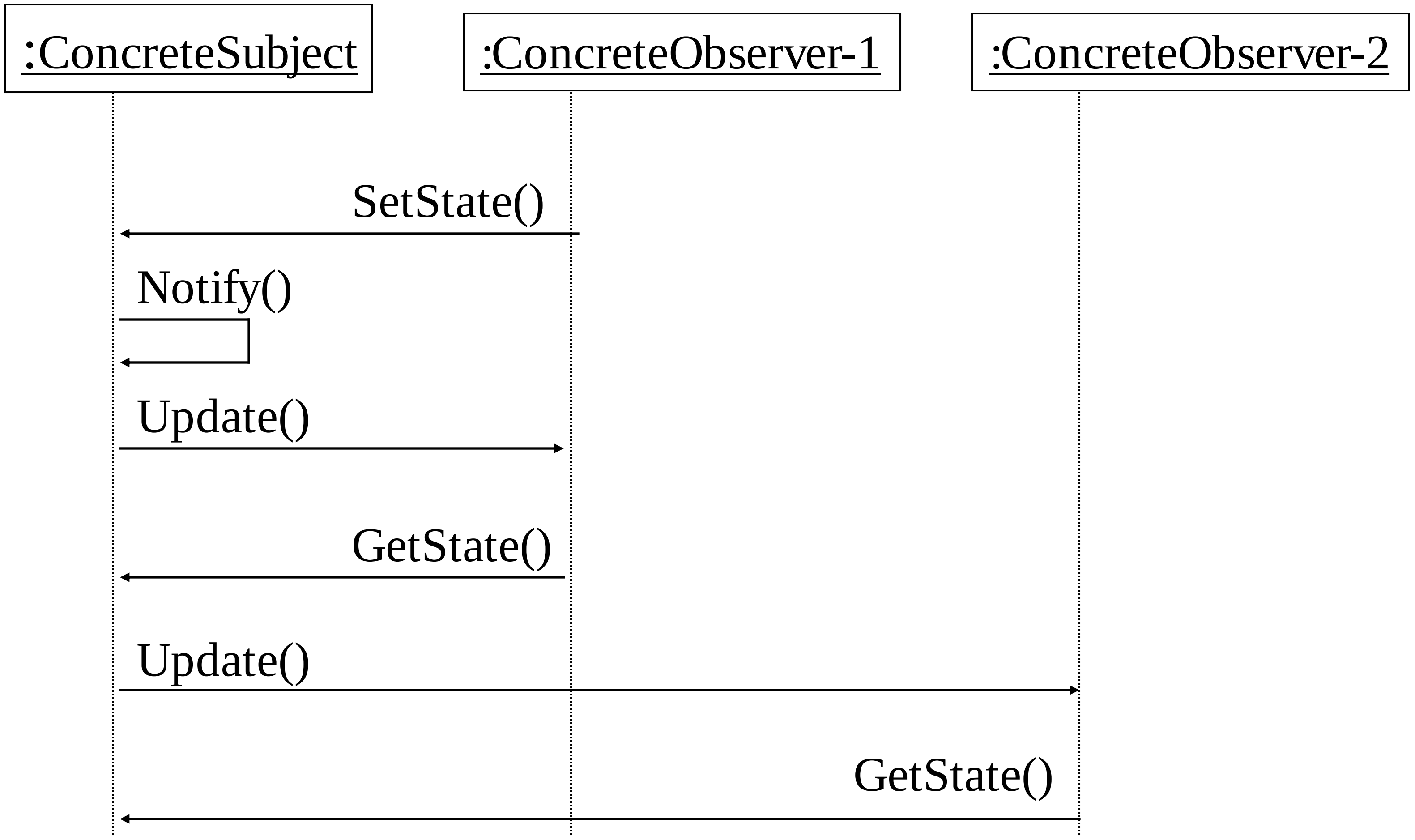
Observer Pattern - Key Players

- Subject
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that gets notified of changes in a subject
- ConcreteSubject
  - Stores “state of interest” to observers
  - Sends notification when state changes
- ConcreteObserver
  - Implements updating interface

Observer Pattern - UML



Observer Pattern - Collaborations



Observer Pattern - Implementation

```
interface Observer {
    void update (Observable sub, Object arg)
    // repaint the pi-chart
}
class Observable {

    public void addObserver(Observer o) {}
    public void deleteObserver (Observer o) {}
    public void notifyObservers(Object arg) {}
     public boolean hasChanged() {}
...
}
```

Java terminology for Subject.

Observer Pattern - Consequences

- Loosely Coupled
    - Reuse subjects without reusing their observers, and vice versa
    - Add observers without modifying the subject or other observers

- Abstract coupling between subject and observer
    - Concrete class of none of the observers is known

- Support for broadcast communication
    - Subject doesn’t need to know its receivers

- Unexpected updates
    - Can be blind to changes in the system if the subject is changed (i.e. doesn’t know “what” has changed in the subject)