ELEC 372
Engineering Software Systems

Week 6
Design Patterns

T.C. Nicholas Graham
Diane Kelly
http://www.cs.queensu.ca/~elec372

Review of Quality Attributes

- Recall
  - Requirements analysis determines desired quality attributes of system
  - When designing system, attempt to
    - Identify areas of risk where quality attributes may not be met
    - Reduce risk through design choices
  - Some attributes amenable to mathematical expression (performance, availability), others less so (modifiability)

Example Quality Attributes

- Performance
- Availability
- Security
- Modifiability
- Usability
- Testability

Design Choices Trade Off Quality Attributes

- E.g., to obtain *performance*, may need to use a very complex algorithm, reducing *modifiability*
- To obtain *availability*, may need to replicate data sources, reducing *security*
- Therefore, design choices rely on knowing required targets for quality attributes
  - At what point is security sufficient?
  - At what point is system sufficiently fast?
  - … Etc.

Design Patterns

- Software architecture normally covers the large-scale design of software systems
- Within architectural components, it can be useful to apply *micro-architectures* to help design these components
- Sometimes called *design patterns*

Design Patterns

- A design pattern is a set of rules of how to structure code in order to solve a particular class of problem
- Provides vocabulary to discuss design
- Particularly suited to OO design as patterns can be expressed in terms of classes, associations
- Design patterns normally help in improving at least one quality attribute, perhaps at the expense of others

Example: the Publish and Subscribe Design Pattern

- Problem
  - A data source is shared by a number of clients
  - Multiple clients depend on the value of the data source
  - *Modifiability* attribute important

Concrete Example: Expense Report

Item	*$US Cost*	*$Cdn Cost*	*Total cost ($Cdn)*
Air		$670	$670
Hotel	$980		$1470
Meals	$70		$105
TOTAL	$1050	$670	$1245

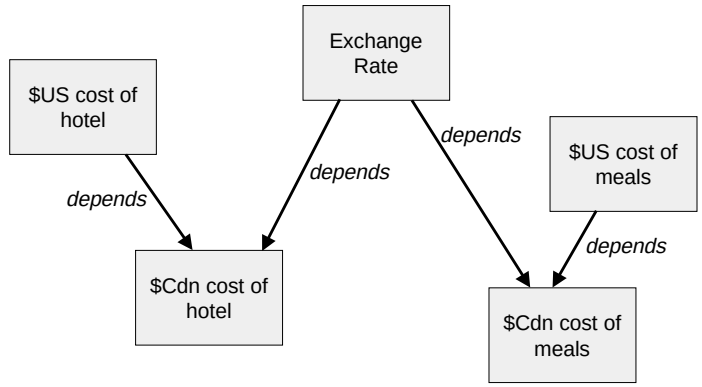
Forms-based interface for calculating expense reports.

Concrete Example: Expense Report

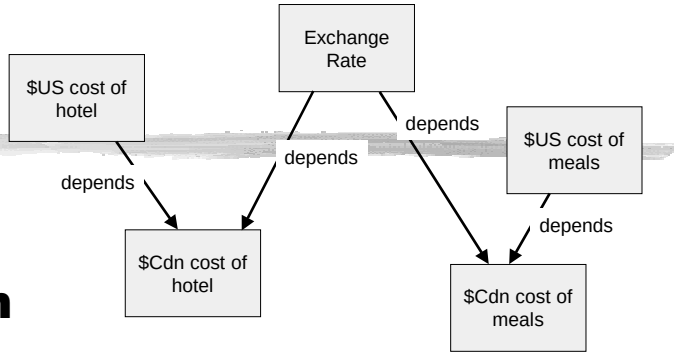
Item	*$US Cost*	*$Cdn Cost*	*Total cost ($Cdn)*
Air		$670	$670
Hotel	$980		$1470
Meals	$70		$105
TOTAL	$1050	$670	$1245

Items depend on value of exchange rate. If exchange rate changes, these values should change too.

Dependencies



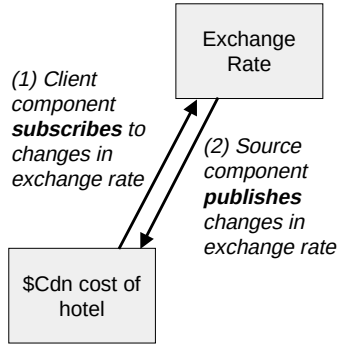
Problem



- When exchange rate modified, must update values of all components that depend on it
- Therefore must know which components depend on exchange rate
- This causes great problems in modifiability

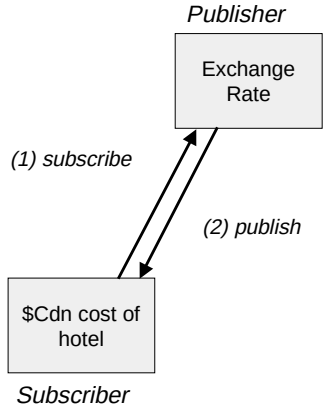
Solution: Publish and Subscribe Design Pattern

- A source component contains data on which a set of client components depend (e.g., current exchange rate)
- The client components *subscribe* to changes in the data
- The source component *publishes* any changes in the data



Solution: Publish and Subscribe Design Pattern

- Advantages
  - Publisher code does not depend on who subscribers are
  - Therefore can add arbitrary subscribers at runtime
  - For example, any entries in expense report that are expressed in $US become subscribers to the exchange rate

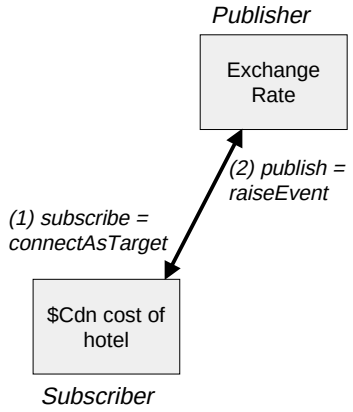


Implementing Design Patterns

- Design patterns represents strategies for organizing code, not explicit implementations
- Actual implementation used must be determined on a case-by-case basis

Event-Connector Implementation

- Define new event type *ExchangeRateChangedEvent* representing change in exchange rate
- Connect subscribers to publisher via event connector
  - Subscribers call connectAsTarget to connect
- Publish changes by firing *ExchangeRateChangedEvent* events



Code for New Event Type

```
import java.util.EventObject;

public class ExchangeRateChangedEvent extends EventObject {
    private float newExchangeRate;

    public int getNewExchangeRate () {
       return newExchangeRate;
    }
    public ExchangeRateChangedEvent (
                Object source, float newRate) {
       super (source);
       newExchangeRate = newRate;
    }
}
```

Where do Design Patterns Come From?

- Actually mined from code
  - General rule -- design pattern must be seen in at least three real systems before it is considered to be a design pattern

Adapter Design Pattern

[Gamma et al., pp 139-150]

- Motivation
  - Sometimes a class implements functionality similar to what an application requires, but not the correct interface for that application
  - E.g.: In wish to implement a *BookList* class implementing a list of text book used in a course
    - `int getNumBooks()`
    - `Book getBook (int n)`
    - `void addBook (Book newBook)`
  - These operations are very similar to the operations in the predefined *Vector* class

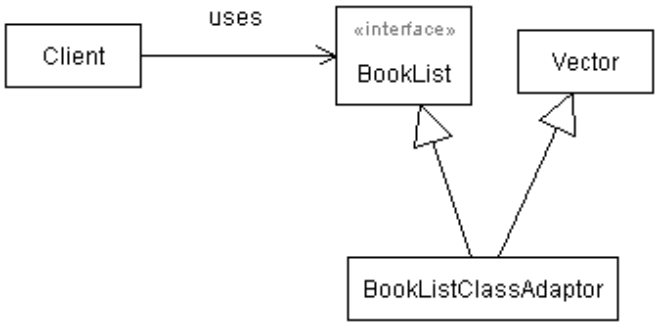
Adapter Design Pattern

- Options
  - Implement *BookList* from scratch
    - Expensive; error prone
  - Use *Vector*
    - Don't get desired interface for *BookList*
  - Adapt *Vector* to *BookList* interface

Adapter

- Two ways of adapting the *Vector*
  - Class adapter
    - Based on inheritance
  - Object adapter
    - Based on aggregation

Class Adapter



Class Adapter

BookList.java

```
public interface BookList {
      public int getNumBooks();
      public Book getBook(int n);
      public void addBook(Book newBook);
}
```

```
import java.util.Vector;
public class BookListClassAdapter
             extends Vector implements BookList {
    public int getNumBooks() {
        return size();
    }

    public Book getBook(int n) {
        return (Book) elementAt(n);
    }
    public void addBook(Book newBook) {
        for (int i=0; i<getNumBooks(); i++) {
            Book b=getBook(i);
            if (b.isSameBook (newBook)) {
                b.setQuantity (b.getQuantity()
                    + newBook.getQuantity());
                return;
            }
        }
        add (newBook);
    }
}
```

BookListClassAdapter.java

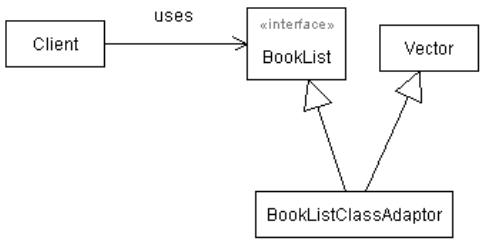
SampleClient.java

```
// This code simply shows how a book list is instantiated

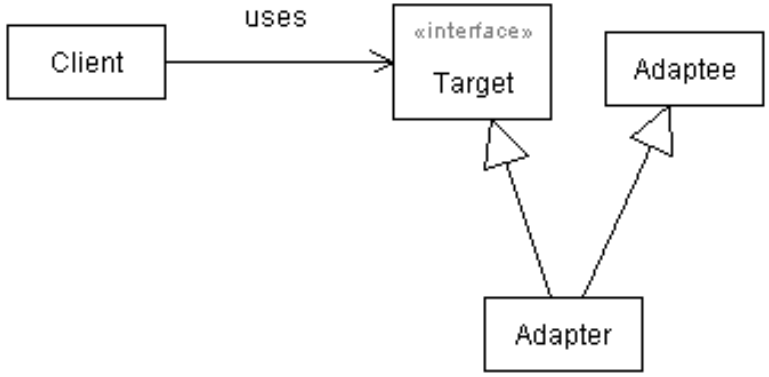
public class SampleClient {
    public static void main (String [] args) {
        Book b = new Book (…);
        BookList bl = new BookListClassAdapter ();
        bl.addBook ();
        System.out.println (“Book list has ” +
            bl.getNumBooks() + “ books”);
    }
}
```

Class Adapter

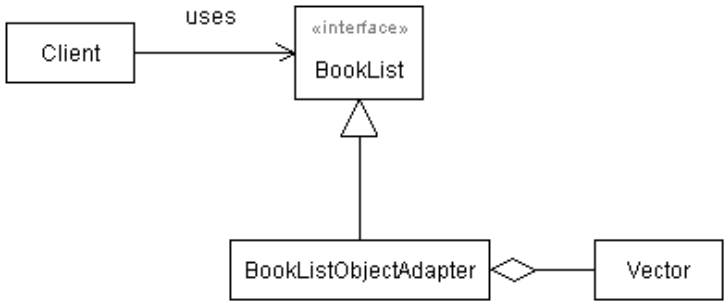
- Adapter simply inherits operations of Vector
- Clients use BookList interface, not BookListClassAdapter directly
- Therefore hides public methods of *Vector* from client



General Form of Class Adapter



Object Adapter



BookListObjectAdapter.java

```
import java.util.Vector;
public class BookListObjectAdapter implements BookList {
    private Vector books = new Vector();

    public int getNumBooks() {
        return books.size();
    }

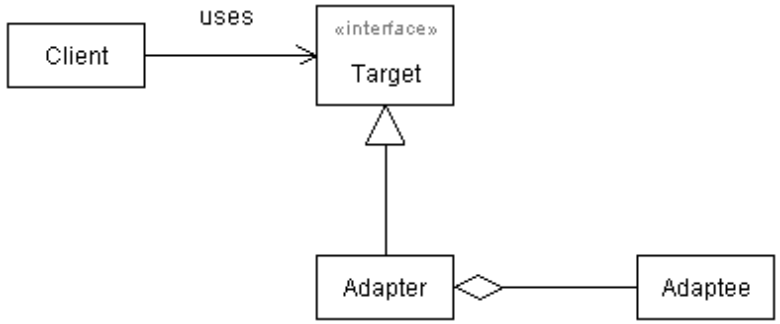
    public Book getBook(int n) {
        return (Book) books.elementAt(n);
    }

    public void addBook(Book newBook) {
        for (int i=0; i<getNumBooks(); i++) {
            Book b=getBook(i);
            if (b.isSameBook (newBook)) {
                b.setQuantity (b.getQuantity()
                    + newBook.getQuantity());
                return;
            }
        }
        books.add (newBook);
    }
}
```

SampleClient.java

```
// This code simply shows how a book list is instantiated
public class SampleClient {
    public static void main (String [] args) {
        Book b = new Book (…);
        BookList bl = new BookListObjectAdapter ();
        bl.addBook ();
        System.out.println ("Book list has " +
            bl.getNumBooks() + " books");
    }
}
```

General Form of Object Adapter



Relation to Quality Attributes

- Modifiability
  - Adapter class provides clean interface to clients of adapted object
- Reusability
  - Takes advantage of adapted class for reuse
- Correctness
  - Takes advantage of tested code

The Façade Pattern

[Gamma et al., pp 185-193]

- Motivation
  - Sometimes part of a system is itself implemented as a subsystem, consisting of a set of objects
  - Rather than revealing subsystem structure to rest of system, use *façade* object to provide interface to subsystem as a whole

The Façade Pattern

- Example
  - A student information system provides access to functions on
    - What students are enrolled in a given course
    - What courses a given student is enrolled in
    - Where a given course is being held
  - Implemented via three classes
    - Courses: information on course enrollments
    - Students: information on students' courses
    - Facilities: information on location of courses

Student Information System Classes

Courses.java

```
class Courses {
   …
   public StudentList getEnrollment (CourseId c) { … }
   …
}
```

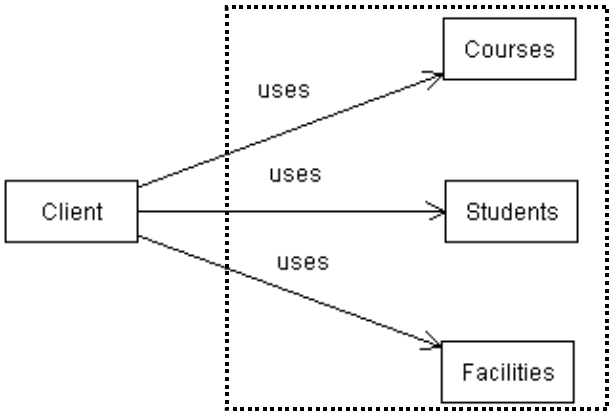
Students.java

```
class Students {
   …
   public CourseList getCourses (StudentId s) { … }
}
```

Facilities.java

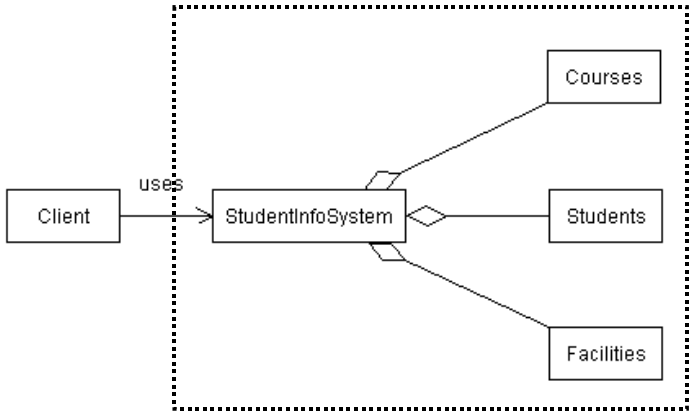
```
class Facilities {
   …
   public LectureHallId getLectureHall (CourseId c) { … }
}
```

Student Information System without Façade



Student Information Subsystem

Student Information System with Façade



Student Information Subsystem

Façade Implementation

StudentInformationSystem.java

```
public class StudentInformation System {
   Courses cs;
   Students ss;
   Facilities fs;
   …
   public StudentList getEnrollment (CourseId c) {
      return cs.getEnrollment (c);
   }

   public CourseList getCourses (StudentId s) {
      return ss.getCourses (s);
   }

   public LectureHallId getLectureHall (CourseId c) {
      return fs.getLectureHall (c);
   }
}
```

Relation to Quality Attributes

- Modifiability
  - By removing external links to subsystem components, impact of changes to subsystem components limited to subsystem itself
- Performance
  - Slight performance cost of extra call indirection through façade
    - But… smart runtime systems such as Javasoft HotSpot may unroll (optimize away) these extra calls