Coming up

- Today
  - Exam review
  - Lecture on design patterns (not on this exam)

- Exam on Wednesday, March 11, in class

Test Review

- This is how it’s going to work:
  - Topics
  - Then I’ll open the floor to questions
  - Once we are out of questions, we’ll move on to the day’s lecture

Exam 1 Topics

- Software development lifecycle
- Teamwork
- In-field debugging (guest lecture)
- Requirements
- Architecture
- UML
- User Interface
- Big Software Data (guest lecture)

Design Patterns

Creational, Structural, Behavioral

Design patterns outline

- Introduction to design patterns
- Creational patterns (constructing objects)
- Structural patterns (controlling heap layout)
- Behavioral patterns (affecting object semantics)

What is a design pattern?

- a standard solution to a common programming problem
  - a design or implementation structure that achieves a particular purpose
  - a high-level programming idiom
- a technique for making code more flexible
  - reduce coupling among program components
- shorthand for describing program design
  - a description of connections among program components
  - the shape of a heap snapshot or object model
Example 1: Encapsulation (data hiding)

- Problem: Exposed fields can be directly manipulated
  - Violations of the representation invariant
  - Dependencies prevent changing the implementation
- Solution: Hide some components
  - Permit only stylized access to the object
- Disadvantages:
  - Interface may not (efficiently) provide all desired operations
  - Indirection may reduce performance

Example 2: Subclassing (inheritance)

- Problem: Repetition in implementations
  - Similar abstractions have similar members (fields, methods)
- Solution: Inherit default members from a superclass
  - Select an implementation via run-time dispatching
- Disadvantages:
  - Code for a class is spread out, and thus less understandable
  - Run-time dispatching introduces overhead

Example 3: Iteration

- Problem: To access all members of a collection, must perform a specialized traversal for each data structure
  - Introduces undesirable dependences
  - Does not generalize to other collections
- Solution:
  - The implementation performs traversals, does bookkeeping
  - Results are communicated to clients via a standard interface
- Disadvantages:
  - Iteration order is fixed by the implementation and not under the control of the client

Example 4: Exceptions

- Problem:
  - Errors in one part of the code should be handled elsewhere.
  - Code should not be cluttered with error-handling code.
  - Return values should not be preempted by error codes.
- Solution: Language structures for throwing and catching exceptions
- Disadvantages:
  - Code may still be cluttered.
  - It may be hard to know where an exception will be handled.
  - Use of exceptions for normal control flow may be confusing and inefficient.

Example 5: Generics

- Problem:
  - Well-designed data structures hold one type of object
- Solution:
  - Programming language checks for errors in contents
  - List<Data> instead of just List
- Disadvantages:
  - Slightly more verbose types

Tips for designing generic classes

- First, write and test a concrete version
  - Consider creating a second concrete version
- Then, generalize it by adding type parameters
  - The compiler will help you to find errors
A puzzle about generics

• Integer is a subtype of Number

• Is List<Integer> a subtype of List<Number>?

   NO!

A puzzle about generics

• Why not?
  
  is List.add(Integer) stronger than List.add(Number)?

• What goes wrong if List<Integer> is a subtype of List<Number>?

  ```java
  List<Integer> li = new ArrayList<Integer>();
  // legal if List<Integer> is subtype of List<Number>:
  List<Number> ln = li;
  ln.add(new Float());
  li.get(0); // we got a Float out of a List<Integer>!
  ```

• Integer[] is a subtype of Number[]
  
  – Can we use similar code to break the Java type system?

Why care about design patterns?

• You could come up with these solutions on your own

• You shouldn't have to!

• A design pattern is a known solution to a known problem

When not to use a design pattern

• Rule 1: delay

• Design patterns can increase or decrease understandability

  |
  |
  | add indirection | improve modularity |
  |
  | increase code size | separate concerns |
  |
  | ease description |

If your design or implementation has a problem, consider design patterns that address that problem

Design pattern references

• Canonical reference:
  
  the "Gang of Four" book

  – Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.

• Another good reference for Java


Creational patterns

• Constructors in Java are inflexible
  
  – Can't return a subtype of the class they belong to
  
  – Always return a fresh new object, never re-use one

• Sharing
  
  – Singleton
  
  – Interning
  
  – Flyweight

• Factories
  
  – Factory method
  
  – Factory object
  
  – Prototype
Sharing

• Java constructors always return a new object

• Three solutions:
  – Singleton: only one object exists at runtime
  – Interning: only one object with a particular (abstract) value exists at runtime
  – Flyweight: separate intrinsic and extrinsic state, and intern the intrinsic state

Singleton

Only one object of the given type exists

```java
class Bank {
    private static Bank theBank;
    // private constructor
    private Bank() { ... }
    // factory method
    public static Bank getBank() {
        if (theBank == null) {
            theBank = new Bank();
        }
        return theBank;
    }
    ...
}
```

Interning pattern

• Reuse existing objects instead of creating new ones
  – Less space
  – May compare with == instead of equals()
• Permitted only for immutable objects

Interning mechanism

• Maintain why HashMap and not HashSet?
  – If an object already appears, return that instead
  ```java
  String canonicalName(String n) {
      if (segnames.containsKey(n)) {
          return segnames.get(n);
      } else {
          segnames.put(n, n);
          return n;
      }
  }
  ```
  – Java builds this in for strings:
    ```java
    String.intern()
    ```
  – Set supports contains but not get

Failure to use the Interning pattern: java.lang.Boolean

```java
public class Boolean {
    private final boolean value;
    // construct a new Boolean value
    public Boolean(boolean value) {
        this.value = value;
    }
    public static Boolean FALSE = new Boolean(false);
    public static Boolean TRUE = new Boolean(true);
    // factory method that uses interning
    public static Boolean valueOf(boolean value) {
        return value ? TRUE : FALSE;
    }
}
```

Recognition of the problem

• Javadoc for Boolean constructor:
  – Allocates a Boolean object representing the value argument.
  – Note: It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory valueOf(boolean) is generally a better choice. It is likely to yield significantly better space and time performance.
    [http://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html)
• Josh Bloch (JavaWorld, January 4, 2004):
  – The Boolean type should not have had public constructors. There’s really no great advantage to allow multiple trues or multiple falses, and I’ve seen programs that produce millions of trues and millions of falses, creating needless work for the garbage collector.
  – So, in the case of immutables, I think factory methods are great.
Why not intern mutable objects?

// Interned class Team represents a set of Students
Team team1 = team.getTeam("Bobby", "Mary");
Team fightingMongooses = Team.getTeam("Bobby", "Mary");

Team1.addStudent("Clair");
fightingMongooses.teamSize() == 2 ?

What if objects are mostly the same?

A car has:
- engine
- cylinders
- crankcase
- pistons
- wheels
- spokes
- diameter
- required pressure
- ...

Model T

Separate out the constant stuff from the stuff that changes. Intern the constant stuff.

Flyweight pattern

- Good when many objects are mostly the same
  - Interning works only if objects are entirely the same (and immutable!)
- Intrinsic state: same across all objects
  - intern it
- Extrinsic state: different for different objects
  - if possible, make it implicit: don’t represent it!
    - making it implicit also requires immutability
    - represent immutable parts explicitly

Example without flyweight: bicycle spoke

```java
class Wheel {
    FullSpoke[] spokes;
    ...
}
class FullSpoke {
    int length;
    int diameter;
    material;
    float weight;
    float threading;
    boolean crimped;
    int location; // rim and hub holes this is installed in
}
```

- Typically 32 or 36 spokes per wheel, but only 3 varieties per bicycle.
- In a bike race, hundreds of spoke varieties, millions of instances

Alternatives to FullSpoke

```java
class IntrinsicSpoke {
    int length;
    int diameter;
    material;
    float weight;
    float threading;
    boolean crimped;
}
```

- This doesn’t work: it’s the same as FullSpoke
```java
class InstalledSpoke extends IntrinsicSpoke {
    int location;
}
```
- This works, but flyweight version uses even less space
```java
class InstalledSpokeWrapper {
    IntrinsicSpoke t; // refer to interned object
    int location;
}
```
Original code to true (align) a wheel

class FullSpoke {
    // Tension the spoke by turning the nipple the
    // specified number of turns.
    void tighten(int turns) {
        ... location ... // location is a field
    }
}
class Wheel {  
    FullSpoke[] spokes;
    void align() {
        while (wheel is misaligned) {
            ... spokes[i].tighten(numturns) ...
        }
    }
}

Flyweight code to true (align) a wheel

class IntrinsicSpoke {
    void tighten(int turns, int location) {
        ... location ... // location is a parameter
    }
}
class Wheel {  
    IntrinsicSpoke[] spokes;
    void align() {
        while (wheel is misaligned) {
            ... spokes[i].tighten(numturns, i) ...
        }
    }
}

Flyweight discussion

• What if FullSpoke contains a wheel field pointing at the Wheel containing it?
• What if FullSpoke contains a boolean broken field?
• Flyweight is manageable only if there are very few mutable (extrinsic) fields.
• Flyweight complicates the code.
• Use flyweight only when profiling has determined that space is a serious problem.

Factories

• Problem: client desires control over object creation
• Factory method
  – Hides decisions about object creation
  – Implementation: put code in methods in client
• Factory object
  – Bundles factory methods for a family of types
  – Implementation: put code in a separate object
• Prototype
  – Every object is a factory, can create more objects like itself
  – Implementation: put code in clone methods

Motivation for factories: Changing implementations

• Supertypes support multiple implementations
  – Interface Matrix { ... }
  – class SparseMatrix implements Matrix { ... }
  – class DenseMatrix implements Matrix { ... }
• Clients use the supertype (Matrix)
  – Still need to use a SparseMatrix or DenseMatrix constructor
  – Switching implementations requires code changes

Use of factories

• Factory
class MatrixFactory {
    public static Matrix createMatrix() {
        return new SparseMatrix();
    }
}
• Clients call createMatrix, not a particular constructor
• Advantages
  – To switch the implementation, only change one place
  – Can decide what type of matrix to create
Example: bicycle race

class Race {
    // factory method
    Race createRace() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
        ...
    }
}

Example: Tour de France

class TourDeFrance extends Race {
    // factory method
    Race createRace() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle();
        ...
    }
}

Example: Cyclocross

class Cyclocross extends Race {
    // factory method
    Race createRace() {
        Bicycle bike1 = new MountainBicycle();
        Bicycle bike2 = new MountainBicycle();
        ...
    }
}

Factory method for Bicycle

class Race {
    Bicycle createBicycle() { ... }
    Race createRace() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle();
        ...
    }
}

class TourDeFrance extends Race {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}

class Cyclocross extends Race {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}

Code using factory methods

class Race {
    Bicycle createBicycle() { ... }
    Race createRace() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle();
        ...
    }
}

class TourDeFrance extends Race {
    RoadBicycle createBicycle() {
        return new RoadBicycle();
    }
}

class Cyclocross extends Race {
    MountainBicycle createBicycle() {
        return new MountainBicycle();
    }
}

Factory objects/classes encapsulate factory methods

class BicycleFactory {
    Bicycle createBicycle() { ... }
    Wheel createWheel() { ... }
    ...
}

class RoadBicycleFactory extends BicycleFactory {
    RoadBicycle createBicycle() {
        return new RoadBicycle();
    }
}

class MountainBicycleFactory extends BicycleFactory {
    MountainBicycle createBicycle() {
        return new MountainBicycle();
    }
}
Using a factory object

```java
class Race {
    BicycleFactory bfactory;
    // constructor
    Race() {
        bfactory = new BicycleFactory();
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
    }
    Race createRace() {
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
        ...
    }
}
class TourDeFrance extends Race {
    // constructor
    TourDeFrance() {
        bfactory = new RoadBicycleFactory();
    }
}
class Cyclocross extends Race {
    // constructor
    Cyclocross() {
        bfactory = new MountainBicycleFactory();
    }
}
```

Separate control over bicycles and races

```java
class Race {
    BicycleFactory bfactory;
    // constructor
    Race(BicycleFactory bfactory) {
        this.bfactory = bfactory;
    }
    Race createRace() {
        Bicycle bike1 = (Bicycle) bfactory.clone();
        Bicycle bike2 = (Bicycle) bfactory.clone();
        ...
    }
}
class TourDeFrance extends Race {
    // constructor
    TourDeFrance() {
        bfactory = new RoadBicycleFactory();
    }
}
class Cyclocross extends Race {
    // constructor
    Cyclocross() {
        bfactory = new MountainBicycleFactory();
    }
}
```

DateFormat factory methods

- DateFormat class encapsulates knowledge about how to format dates and times in text
  - Options: just date? just time? datetme? where in the world?
  - Instead of passing all options to constructor, use factories.
  - Tidy, and the subtype created doesn’t need to be specified.
- DateFormat df1 = DateFormat.getDateInstance();
- DateFormat df2 = DateFormat.getTimeInstance();
- DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL, Locale.FRANCE);
- Date today = new Date();
- System.out.println(df1.format(today)); // "Jul 4, 1776"
- System.out.println(df2.format(today)); // "10:15:00 AM"
- System.out.println(df3.format(today)); // "lunedie 4 juillet 1776"

Prototype pattern

- Every object is itself a factory
- Each class contains a clone method that creates a copy of the receiver object
  ```java
  class Bicyle {
      Bicyle clone() {
          ...
      }
  }
  ```
  - You will often see Object as the return type of clone
    - This is due to a design flaw in Java 1.4 and earlier
    - clone is declared in Object
    - Java 1.4 did not permit the return type to change in an overridden method

Using prototypes

```java
class Race {
    Bicycle bproto;
    // constructor
    Race(Bicycle bproto) {
        this.bproto = bproto;
    }
    Race createRace() {
        Bicycle bike1 = (Bicycle) bproto.clone();
        Bicycle bike2 = (Bicycle) bproto.clone();
        ...
    }
}
```