Principles of Software Construction: Objects, Design, and Concurrency

Design Goals and Process

Jonathan Aldrich    Charlie Garrod
Review: In the Previous Lecture…

• We designed a forestry simulation. What made it challenging?

• What is it that generally drives design decisions?

• What is the feature of object-oriented programming that facilitates extensibility?
We designed a forestry simulation. What made it challenging?
- Extensibility and evolvability requirements

What is it that generally drives design decisions?
- Quality Attributes (extensibility, evolvability, and others)

What is the feature of object-oriented programming that facilitates extensibility?
- Dispatch
Review/Elaboration: Quality Attributes

- Design driven by extensibility, modifiability requirements
  - Framework should work unmodified with other simulations
  - Can extend with new agents
    - without modifying existing agents or the framework
  - Can modify the simulation setup or modify the set of agents
    - Without modifying the agents or the framework

- Extensibility and modifiability are quality attributes
  - Properties of software that describe its fitness for further development and use
  - Not what the system does but how well it does it

- Other quality attributes
  - Performance
  - Availability
  - Security
  - Testability
  - Usability

The major focus of design is achieving quality attributes
Review: Dispatch in the Previous Lecture

1. assign a0 to grid[0]
2. assign a1 to grid[1]
3. invoke grid[0].timeStep()
4. invoke grid[1].timeStep()

*a simplification: we consider a 1-dimensional grid in this diagram*

Object a0 is a LodgepolePine Dispatch to code in the LodgepolePine class

Object a1 is a LodgepolePine Dispatch to code in the LodgepolePine class
Puzzle: What Does This Code Print?

```java
class Table {
    BallState state;
    void set(BallState s) {
        state = s;
    }
    void play(int n) {
        for (int i = 0; i < n; ++i)
            state.bounce(this);
        state.print();
    }
}

void main(...) {
    Table t = new Table();
    t.set(new Ping());
    t.play(3);
}
```

```java
interface BallState {
    void bounce(Table t);
    void print();
}

class Ping implements BallState {
    void bounce(Table t) {
        t.set(new Pong());
    }
    void print() {
        System.out.println("Ping");
    }
}

class Pong implements BallState {
    void bounce(Table t) {
        t.set(new Ping());
    }
    void print() {
        System.out.println("Pong");
    }
}
```

System.out is an object representing standard output.

Refers to the state field of the current object **this**.
Could have written: **this.state = s;**
Dispatch in Ping-Pong

1. Table t = new Table();
2. t.add(new Ping());
3. t.play(3);
   a. state.bounce(this);
   b. state.bounce(this);
   c. state.bounce(this);
   d. state.print();

*p.simplification: we consider a 1-dimensional grid in this diagram*
Learning Goals

- Review quality attributes, extensibility, and dispatch
- Know the steps of the design process
- Understand quality attributes in more depth
- Learn how several design guidelines promote quality attributes
- Illustrate the design process through an example
- Learn Java’s encapsulation constructs
Why a Design Process?

- **Without a process, how do you know what to do?**
  - A process tells you what is the next thing you should be doing

- **A process structures learning**
  - We can discuss individual steps in isolation
  - You can practice individual steps, too

- **If you follow a process, we can help you better**
  - You can show us what steps you have done
  - We can target our advice to where you are stuck
Steps in the Design Process *(example: forest simulation)*

- **Precondition: understand functional requirements**
  - Step-by-step simulation of the forest, according to a spec

- **Precondition: understand quality attribute requirements**
  - Extension with new agents; easily change simulation setup

- **Design a logical architecture**
  - Driven by quality attributes: what code must change independently

- **Design a behavioral model**
  - The interactions between components, and their order

- **Responsibility assignment**
  - Which components store data and implement behavior

- **Interface design**
  - The operations of each component, and their signatures

- **Algorithm and data structure design – pseudo-code**

- **Postcondition: ready to code**
Steps in the Design Process (example: forest simulation)

- Precondition: understand functional requirements
  - Step-by-step simulation of the forest, according to a spec

- Precondition: understand quality attribute requirements
  - Extension with new agents; easily change simulation setup

- Design a logical architecture
  - Driven by quality attributes: what code must change independently

- Design a behavioral model
  - The interactions between components, and their order

- Responsibility assignment
  - Which components store data and implement behavior

- Interface design
  - The operations of each component

- Algorithm and data structure design

- Postcondition: ready to code

Caveats:
- You may skip steps
- You may backtrack
- Some steps break down further
Data Structure Design: Mathematical Sets of Integers

• Design a library representing mathematical sets of integers

• The library should support:
  ▪ Creating empty and singleton sets
  ▪ Computing the union, intersection, and difference of two sets
  ▪ Testing membership of an integer in a set
  ▪ *(likely more goes here...)*

• Plan to extend the library with efficient representations:
  ▪ A representation that represents singleton sets with little space
  ▪ A representation for which union and intersection are fast
  ▪ A representation for which testing membership is fast
  ▪ A representation that is well-balanced across all operations

• Should be able to modify each representation independently

• Different representations should interoperate

Which of these requirements are quality attributes?
Steps in the Design Process *(example: mathematical sets)*

- **Precondition: understand functional requirements**
  - Operations the sets must support

- **Precondition: understand quality attribute requirements**
  - Extensible representations; Interoperability; Performance

- **Design a logical architecture**
  - Trivial: an interface and multiple representations

- **Design a behavioral model**
  - No interesting constraints on order of operations

- **Responsibility assignment**
  - Trivial: the set object does it all

- **Interface design**
  - Mostly specified by the functional requirements

- **Algorithm and data structure design – pseudo-code**
  - This will be our main focus
IntSet Interface Design *(in-class version)*

**interface** IntSet { 

- The library should support:
  - Computing the union of two sets
  - Testing membership of an integer in a set
  - ... *(and other operations)*
IntSet Interface Design (prepared version)

```java
interface IntSet {
    /** @return the union of this and s */
    IntSet union(IntSet s);

    /** @return true if this contains i */
    boolean contains(int i);

    // and other operations
}
```

- The library should support:
  - Computing the union of two sets
  - Testing membership of an integer in a set
  - ... (and other operations)
Algorithm and Data Structure Design

• What choices will support:
  ▪ A representation that represents singleton sets with little space?
  ▪ A representation for which union and intersection are fast?
  ▪ A representation for which testing membership is fast?
  ▪ A representation that is well-balanced across all operations?
Algorithm and Data Structure Design

- **What choices will support:**
  - A representation that represents singleton sets with little space?
    - A single field holding the singleton member
  - A representation for which union and intersection are fast?
    - A UnionSet object with fields for its constituent sets (and similar for IntersectionSet)
  - A representation for which testing membership is fast?
    - A hashtable allows expected constant-time membership testing
  - A representation that is well-balanced across all operations?
    - A sorted array would provide logarithmic membership testing and union operations
Implementing Set

• Trivial example: an empty set

class EmptySet implements IntSet {
    /** @return the union of this and s */
    IntSet union(IntSet s) {
    }
}

• Some OO rules and concepts:
  ▪ Must provide method bodies for all the messages in the interface
    ▪ It is an error if we forget one, or change its signature
  ▪ May define additional methods and/or data fields
  ▪ The class is a subtype of the interfaces it implements
Implementing Set

• Trivial example: an empty set

```java
class EmptySet implements IntSet {
    /** @return the union of this and s */
    IntSet union(IntSet s) { return s; }
}
```

• Some OO rules and concepts:
  ▪ Must provide method bodies for all the messages in the interface
    ▪ It is an error if we forget one, or change its signature
  ▪ May define additional methods and/or data fields
  ▪ The class is a `subtype` of the interfaces it implements

```java
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int i);
}
```

error: method contains from interface IntSet is not implemented
Implementing Set

- Trivial example: an empty set

```java
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int i);
}
```

```java
class EmptySet implements IntSet {
    /** @return the union of this and s */
    IntSet union(IntSet s) { return s; }
    /** @return true if this contains i */
    boolean contains(int i) { return false; }
}
```

- Some OO rules and concepts:
  - Must provide method bodies for all the messages in the interface
    - It is an error if we forget one, or change its signature
  - May define additional methods and/or data fields
  - The class is a subtype of the interfaces it implements
Typechecking client code

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
}

class EmptySet implements IntSet { ... }
```

1. The `new` expression has type `EmptySet`.
2. OK to assign an `EmptySet` to an `IntSet`, because `EmptySet` implements `IntSet`.
3. `s` has type `IntSet`. We check that `IntSet` defines a `contains` method.
4. The `contains()` method in `IntSet` accepts an `int` argument so the actual argument is OK.
5. `contains()` returns a `boolean`, which we can assign safely to `f`.

```
IntSet s = new EmptySet();

boolean f = s.contains(0); // false
```
Typechecking: What Could Go Wrong?

interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
}
class EmptySet implements IntSet { ...

2. Can’t assign an IntSet to an EmptySet because IntSet is not a subtype of (i.e. does not implement) EmptySet

EmptySet s = new IntSet();

int f = s.contains("hello"); // false

5. contains() returns a boolean, which is not a subtype of int (unlike in C)

3. s has type EmptySet. But EmptySet does not define a contains method

1. Can’t instantiate an interface; its methods are undefined.

4. Even if we spell contains correctly, the method takes an int argument, and String is not a subtype of int
class SingletonSet implements IntSet {
    int member;
    boolean contains(int e) { return member == e; }
    IntSet union(IntSet otherSet) {
        UnionSet u = new UnionSet();
        u.set1 = this;
        u.set2 = otherSet;
        return u;
    }
}

class UnionSet implements IntSet {
    IntSet set1;
    IntSet set2;
    boolean contains(int e) {
        return set1.contains(e) || set2.contains(e);
    }
    // other methods
}

Issue: what if we want to represent unions with an array of ints?

Quality Attribute: Should be able to modify each representation independently

Design Guideline [Representation Hiding]: Hiding the representation of an object from other code helps make it easier to modify the representation
Implementing Singleton and Union Sets (version 2)

class SingletonSet implements IntSet {
    private int member;

    public SingletonSet(int element) { member = element; }

    public boolean contains(int e) { return member == e; }

    public IntSet union(IntSet otherSet) {
        return new UnionSet(this, otherSet);
    }
}

class UnionSet implements IntSet {
    private IntSet set1;
    private IntSet set2;

    public UnionSet(IntSet s1, IntSet s2) {
        set1 = s1; set2 = s2;
    }

    // other methods not shown
}

A private field can’t be used from outside the class

A constructor method initializes the fields

public members—i.e. methods and fields—can be accessed from anywhere

Allocates memory and calls the constructor of UnionSet

Now we can change the representation of unions without affecting other code
Implementing Singleton and Union Sets (version 2)

```java
class SingletonSet implements IntSet {
    private int member;

    public SingletonSet(int element) { member = element; }

    public boolean contains(int e) { return member == e; }

    public IntSet union(IntSet otherSet) {
        return new UnionSet(this, otherSet);
    }
}

SingletonSet s = new SingletonSet(5);
if (s.member <= 5) s.member++;
```

Note: all methods in an interface are implicitly public

Discussion: when is it useful to have a private method?

error: cannot access private field member from outside class SingletonSet
Implicit Constructors

• If you don’t define a constructor, Java generates one for you
  • It has no return type and is named after the class
    • Just like all constructors
  • It has no arguments
  • Fields (if any) are initialized to default values
    • 0 for numeric values
    • \texttt{false} for \texttt{boolean} variables
    • \texttt{null} for reference (pointer) variables

\begin{verbatim}
class EmptySet implements IntSet {
    /** This is equivalent to the auto-generated constructor */
    public EmptySet() {}  
    public IntSet union(IntSet s) { return s; }
    public boolean contains(int i) { return false; }
}
\end{verbatim}
Using Sets Together

**Quality Attribute:** Different representations should be able to interoperate

```java
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

What does this program print?
Using Sets Together

Quality Attribute: Different representations should be able to interoperate

Method Stack

```
main()
  s1
  s2
  temp
```

```java
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

What does this program print?
Using Sets Together

Quality Attribute: Different representations should be able to interoperate

Method Stack

main()
s1
s2
temp

IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));

What does this program print?
Using Sets Together

**Dynamic Dispatch:** determine which method to call based on the runtime class of the object.

**Polymorphism ("many forms"):** Sets can take two forms, and the behavior of a set depends on which form it takes.

**Method Stack**

```
main()
s1
s2
temp
```

```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

- **s1** points to **s**. 
  - **s** is of class **SingletonSet**. 
  - SingletonSet.contains() is called, printing true

- **s2** points to **e**. 
  - **e** is of class **EmptySet**. 
  - EmptySet.contains() is called, printing false
class ArraySet implements IntSet {
    private int members[]; // the array is sorted
    public ArraySet(int ms[], int size) {
        members = new int[size];
        for (int i = 0; i < size; ++i)
            members[i] = ms[i];
    }
    public boolean contains(int element) {
        /* binary search */
    }
    public IntSet union (IntSet s) {
        int ms[] = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
}
class ArraySet implements IntSet {
    private int members[]; // the array is sorted
    public ArraySet(int ms[], int size) {
        members = new int[size];
        for (int i = 0; i < size; ++i)
            members[i] = ms[i];
    }
    public boolean contains(int element) {
        /* binary search */
    }
    public ArraySet union (ArraySet s) {
        int ms[] = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
}
ArraySet – is this Good Code?

class ArraySet {
    private int members[]; // the array is sorted
    private int temp[]; // for performing unions
    public ArraySet union (ArraySet s) {
        if (temp == null || size + s.size > temp.length)
            temp = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
    // other methods not shown
}

the special length field can be used to get the array size
ArraySet – is this Good Code?

class ArraySet {
    private int members[]; // the array is sorted
    private int temp[]; // for performing unions
    public ArraySet union (ArraySet s) {
        if (temp == null || size + s.size > temp.length)
            temp = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
    // other methods not shown
}

cohesion means the code for one issue is localized. High cohesion makes code easier to understand and modify.

This code has low cohesion because data structures used in the union algorithm are spread outside the union method.
Using Sets Together

**Quality Attribute:** Different representations should be able to interoperate

**Quality Attribute:** Support a representation that performs well on all operations

Two quality attributes, interoperability and performance, are in conflict

```
IntSet s1 = new EmptySet();
ArraySet s2 = new ArraySet(new int[] { 5 });
IntSet temp = s2;
ArraySet s3 = s2.union(s1);
IntSet s4 = s1.union(s2);
```

**error:** ArraySet is not a subtype of IntSet

**error:** argument s1 of type IntSet is not a subtype of ArraySet

**error:** argument s2 of type ArraySet is not a subtype of IntSet
An ArraySet Abstract Data Type (ADT)

class ArraySet {
    private int members[]; // the array is sorted
    public ArraySet(int ms[], int size) {
        members = new int[size];
        for (int i = 0; i < size; ++i)
            members[i] = ms[i];
    }
    public boolean contains(int element) {
        /* binary search */
    }
    public ArraySet union (ArraySet s) {
        int ms[] = new int[size + s.size];
        // copy non-duplicate members
    }
}

ArraySet is an Abstract Data Type. Its binary operations access objects of the same fixed class type (and therefore the same fixed representation).

This has performance advantages but interoperability and extensibility disadvantages.

Typical application programs prioritize interoperability/extensibility and therefore prefer interface types over class types.

union is a binary operation because it accepts another set.
ArraySet Design Alternatives, Part 1

```java
class ArraySet implements IntSet {
    public IntSet union (IntSet s) {
        if (s instanceof ArraySet) {
            ArraySet arrSet = (ArraySet) s;
            // optimized code here
        } else {
            // default code here
        }
    }
    // other methods not shown
}
```

- **Benefits**

- **Drawbacks**

**instanceof** checks at run time whether s is really an ArraySet

A cast lets us treat s as an ArraySet. Java checks (again) that the object really is an ArraySet
ArraySet Design Alternatives, Part 1

class ArraySet implements IntSet {
    public IntSet union (IntSet s) {
        if (s instanceof ArraySet) {
            ArraySet arrSet = (ArraySet) s;
            // optimized code here
        } else {
            // default code here
        }
    }
    // other methods not shown
}

• Benefits
  ▪ Provides both interoperability and performance

• Drawbacks
  ▪ Two versions of code are harder to maintain
  ▪ Hard to extend – need new instanceof cases for each new rep.

instanceof checks at run time whether s is really an ArraySet

A cast lets us treat s as an ArraySet. Java checks (again) that the object really is an ArraySet

How do we avoid case-based reasoning? Come up with a higher-level method that unifies the cases, and dispatch to it.

Because extensibility is typically a high priority, object-oriented designers avoid using instanceof
ArraySet Design Alternatives, Part 2

```java
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
    int[] getMembers();
}
```

- Benefits

- Drawbacks
ArraySet Design Alternatives, Part 2

```java
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
    int[] getMembers();
}
```

• Benefits
  ▪ Can implement union efficiently for ArraySet
  ▪ ArraySet is an instance of IntSet and interoperates

• Drawbacks
  ▪ Maybe getMembers() is not a method clients should call
  ▪ Maybe getMembers() is hard for other implementations to implement efficiently
Toad’s Takeaways: Design Goals and Process

• Quality attributes such as extensibility and performance drive design

• Following a process can help with being a more effective designer

• Design guidelines that enhance quality attributes
  ▪ Hiding an object’s representation makes it easier to change representations
  ▪ Making fields private and copying internal arrays (or mutable objects) is one way to hide representation
  ▪ Designing for high cohesion makes code easier to understand and modify
  ▪ Programming to interfaces rather than class types facilitates extensibility and interoperability
  ▪ ADTs, instanceof, and casts can be useful, e.g. for performance, but compromise extensibility and are discouraged in OO. Use dispatch instead!