

Lecture 25: Design

Last time:

1. Interfaces (cont.)
2. Wrappers

Today:

1. Project #5 is due 10/31 at 11 pm
2. Program designs: algorithms, interfaces, use cases





Project #5 Assigned!

- Project due Tuesday, 10/31 at 11 pm
- Project is **closed**
 - You must complete the project by yourself
 - Assistance can only be provided by teaching assistants (TAs) and instructors
 - You must not look at other students' code
- Start now!
 - Read entire assignment from beginning to end before starting to code
 - Check out assignment now from CVS
 - Follow the instructions *exactly*, as much of grading is automated



Conditional Expressions

- `if-else` can be used to create conditional *statements*
- Java also has a conditional *expression* operation
 - Form: `<bexp> ? <exp1> : <exp2>`
 - Meaning
 - If `<bexp>` evaluates to true ...
 - ... then `<exp1>` is evaluated
 - ... otherwise `<exp2>` is evaluated
- Example

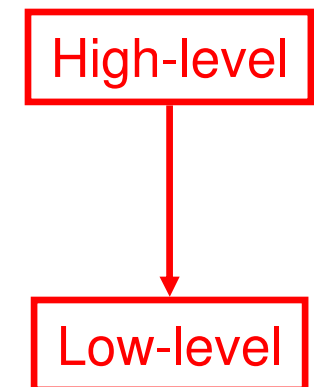
```
int x = 0;
int y = (x < 0) ? x+1 : x+2;
System.out.println(y);
```

2 is printed
- Difference between `if-else`, `<bexp> ? <exp1> : <exp2>`
 - `if-else` is for **statements**
 - `<bexp> ? <exp1> : <exp2>` is for **expressions**
 - Can't e.g. say
 - `int y = if (x<0) x+1 else x+2;`
 - `(x < 0) ? y = x+1; : y = x+2;`



Coding vs. Software Design

- **Coding**: writing of (Java) code to implement classes, methods, etc.
 - Projects so far have been primarily coding
 - We have told you what to code
- **Design**: determination of what to code
 - What classes are needed?
 - How should classes interact?
 - What methods belong in each class?
 - How should method functionality be implemented?



Low-Level Design: Pseudo-Code and Algorithms



- We have already talked about **pseudo-code** as a design technique
 - NOT English
 - NOT a program
 - Something in-between
 - Captures the logic, flow of desired code
 - Note that pseudo-code could be translated into any programming language (not just Java)
- Pseudo-code is used to represent **algorithms** = step-by-step solutions to problems
- Algorithms are often coded as single methods



Example: Linear Search

- Recall `findMin` from last time
 - Given a non-empty array ...
 - ... find the smallest element
- Algorithm used is called **linear search**

In pseudo-code:

set variable `min` to initial element of array

for each subsequent element in array

compare element to `min`

if element is less than `min`, assign its value to `min`

- The (polymorphic) `findMin` method is the Java code implementing linear search



findMin

```
static Comparable findMin (Comparable[] a) {  
    Comparable min = a[0];  
    for (int i=1; i < a.length; i++)  
        if (a[i].compareTo(min) < 0)  
            min = a[i];  
    return min;  
}
```

Concerns at the Algorithmic Level of Design



- **Correctness**

Does my algorithm correctly solve the problem?

- **Efficiency**

Is my algorithm fast enough for the job?

- **Clarity**

Is my algorithm understandable?

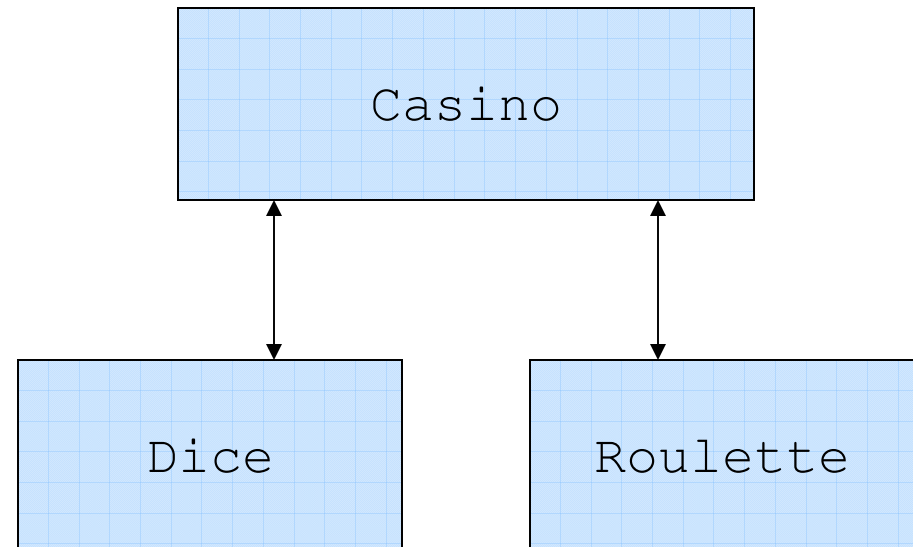


Interfaces and Design

- Next level up the design hierarchy: what methods should go in classes?
- This information can be captured using **interfaces**
- These interfaces can also be used to identify opportunities for **polymorphism** (reusable code)

Example: Casino

- Suppose we are writing a simple casino program
 - Two games: Roulette and Dice
 - Roulette bets: 1-36, Even, Odd
 - Dice bets: 1-6
- Rough structure of classes given to right



Designing a Casino Game Interface



- What methods should be in interface between Casino and Dice / Roulette?
 - Methods should support a generic “game driver” in Casino class
 - Methods should be common to both Dice, Roulette
- What notions are common to Dice, Roulette?
 - Players make types of bets
 - Players bet a certain amount
 - Players play against the house (not each other)
 - There is either a payoff or a loss



Interface Game.java

```
public interface Game {  
    String getName ();           // Name of game  
    String[] getValidBets ();   // Types of bets  
    boolean isValidBet (String b); // Is bet valid?  
    void setBetAmount (int amt); // Set bet amount  
    void setBet (String bet);   // Set bet type  
    int play ();                // Play the game  
}
```



Rules of Thumb

- Keep interfaces small
- Think carefully about operations needed “between classes”
- Use interfaces to support polymorphism (and keep code size down)

A Generic Game Method in Casino



See `play` method in `Casino.java`

Upper Levels of Software Design



- Where do ideas for classes, interactions between classes come from?
 - Software development part of larger system design process
 - System design requires identifying what system users expect system to do
 - These user requirements often suggest system components and how they fit together
- First part of software design: understand system design



System Design: What Is It?

- **System design** is concerned with:
 - coordinating a collection of entities...
 - ... to achieve a complex process
- Each entity has its own responsibilities to the others to achieve an overall objective
- E.g. Restaurant
 - Entities Chef, owners, waiters, etc.
 - System Restaurant



System Examples

- **Classroom environment:** Lecturers, TAs, students, ...
 - **Library:** Circulation (checkout and return), indexing services (online catalogue), library users, book buyers, shelvers, ...
 - **Pharmacy:** Patients (and medical records), pharmacists, doctors, drug retailers, the pharmacy (products in stock), ...
 - **Video game:** Race cars, motorcycles, warriors, space ships, death squads, monsters, aliens, mutants, guns, swords, weapons of mass destruction, cute Japanese cartoon animals with huge eyes, ...



Essential Questions

- What is the **desired behavior** of the program (as a whole)?
- What are the **entities** that produce this behavior?
- How does each one **work**?
- How do these entities **interact**?

Specifying Desired Behavior with Use Cases



- **Use case:** a description of an interaction of a user and the system. Use cases include:
 - **Prerequisites (pre-conditions)**
What must hold for this use case to arise?
 - **Possible actions and interactions**
What happens?
 - **Effects (post-conditions)**
What conditions hold, what changes have taken place, as a result of these actions.
- **Example: Customer in a restaurant**
 - **Pre-conditions:**
Customer: hungry and has money
Restaurant: is open
 - **Actions:** get menu, order food, be served, eat, pay, leave
 - **Post-conditions:**
Customer: full, less money
Restaurant: more money, less food



Principal Design Elements

- **Components**
 - What are the **entities** that make up our system?
 - What are the **roles** they play?
 - How do we separate the system into **distinct units**?
- **Contract**
 - What are the **responsibilities** and services associated with each component?
 - What **guarantees** does it make?
- **State**

What is the current status/state of the units that define our system?
- **Interaction**

Use cases

Example: Pharmacy Store System



- Components
Pharmacist, customers, doctors, prescription, store stock, ...
- Contract
 - Pharmacists can fill prescriptions
 - Doctors write prescriptions
 - Pharmacists only fill the prescriptions written by doctors
 - Pharmacists and doctors maintain patient records
 - etc.
- State (patient)
Current prescriptions, number of times refilled, date of last refill, health insurance information
- Fill prescription use case:
 - A valid prescription presented by the customer
 - Pharmacists checks patient records and informs of possible side-effects
 - Pharmacist dispenses prescription
 - Patient records updated
 - Medication given to patient



Relationship to Java

- System: A Java program
- Components: Java classes and objects
- State: Object instance variables, class static variables
- Contract: **API / interface**
 - The external (class user) view of an object
 - Provides an abstraction of what the object does, without indicating how it is implemented
 - Includes prototypes of actions (methods)
- The contract is implemented using methods