Designing Classes

• This chapter emphasizes the design of software
• Two versions of the project “zuul” are presented
  – Zuul is an “adventure” game where the player goes from room to room
  – A “bad” version is fully functional as a framework but poorly designed
  – Based on software engineering principles (next slide) we redo this design
  – The “better” version is then presented
  – As a sequence of two labs, you will develop an adventure game of your own design
Software Engineering Principles

• Code duplication should be avoided
• *Cohesion* should be maximized, this describes the unity of the fields and methods within a class
• *Coupling*, the interconnection between classes, should be minimized
• Strong encapsulation uses information hiding to minimize coupling
• When a design is completed and the software functionality is expanded, it may be necessary to *refactor* the design in order to maximize cohesion and minimize coupling
/**
 * This class is the main class of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game. Users
 * can walk around some scenery. That's all. It should really be extended
 * to make it more interesting!
 *
 * This main class creates and initialises all the others: it creates all
 * rooms, creates the parser and starts the game. It also evaluates and
 * executes the commands that the parser returns.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 1.0 (February 2002)
 */

class Game
{
    private Parser parser;
    private Room currentRoom;

    /**
     * Create the game and initialise its internal map.
     */
    public Game()
    {
        createRooms();
        parser = new Parser();
    }

    // methods go here
}
/**
 * Create all the rooms and link their exits together.
 */

private void createRooms()
{
    Room outside, theatre, pub, lab, office;
    // create the rooms
    outside = new Room("outside the main entrance of the university");
    theatre = new Room("in a lecture theatre");
    pub = new Room("in the campus pub");
    lab = new Room("in a computing lab");
    office = new Room("in the computing admin office");
    // initialise room exits
    outside.setExits(null, theatre, lab, pub);
    theatre.setExits(null, null, null, outside);
    pub.setExits(null, outside, null, null);
    lab.setExits(outside, office, null, null);
    office.setExits(null, null, null, lab);
    currentRoom = outside;  // start game outside
}

// javadoc removed to fit on slide

public void play()
{
    printWelcome();
    // Enter the main command loop. Here we repeatedly read commands and
    // execute them until the game is over.
    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing. Good bye.");
}
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to Adventure!");
    System.out.println("Adventure is a new, incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if(currentRoom.northExit != null)
        System.out.print("north ");
    if(currentRoom.eastExit != null)
        System.out.print("east ");
    if(currentRoom.southExit != null)
        System.out.print("south ");
    if(currentRoom.westExit != null)
        System.out.print("west ");
    System.out.println();
}

// Note: reordered from source code to fit on slide
private void printHelp()
{
    System.out.println("You are lost. You are alone. You wander");
    System.out.println("around at the university.");
    System.out.println();
    System.out.println("Your command words are:");
    System.out.println("   go quit help");
}
private boolean processCommand(Command command) {
    boolean wantToQuit = false;
    if (command.isUnknown()) {
        System.out.println("I don't know what you mean...");
        return false;
    }

    String commandWord = command.getCommandWord();
    if (commandWord.equals("help"))
        printHelp();
    else if (commandWord.equals("go"))
        goRoom(command);
    else if (commandWord.equals("quit"))
        wantToQuit = quit(command);
    return wantToQuit;
}

private void goRoom(Command command) {
    if (!command.hasSecondWord()) {
        // if there is no second word, we don't know where to go...
        System.out.println("Go where?");
        return;
    }

    String direction = command.getSecondWord();
    // Try to leave current room.
    Room nextRoom = null;
}
if (direction.equals("north"))
    nextRoom = currentRoom.northExit;
if (direction.equals("east"))
    nextRoom = currentRoom.eastExit;
if (direction.equals("south"))
    nextRoom = currentRoom.southExit;
if (direction.equals("west"))
    nextRoom = currentRoom.westExit;

if (nextRoom == null)
    System.out.println("There is no door!");
else {
    currentRoom = nextRoom;
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if (currentRoom.northExit != null)
        System.out.print("north ");
    if (currentRoom.eastExit != null)
        System.out.print("east ");
    if (currentRoom.southExit != null)
        System.out.print("south ");
    if (currentRoom.westExit != null)
        System.out.print("west ");
    System.out.println();
}
}
/**
 * "Quit" was entered. Check the rest of the command to see
 * whether we really quit the game. Return true, if this command
 * quits the game, false otherwise.
 */

private boolean quit(Command command)
{
    if(command.hasSecondWord()) {
        System.out.println("Quit what?");
        return false;
    }
    else
        return true;  // signal that we want to quit
}
/*
 * Class Room - a room in an adventure game.
 *
 * A "Room" represents one location in the scenery of the game. It is
 * connected to other rooms via exits. The exits are labelled north,
 * east, south, west. For each direction, the room stores a reference
 * to the neighboring room, or null if there is no exit in that direction.
 *
 * @author  Michael Kolling and David J. Barnes
 * @version 1.0 (February 2002)
 */

class Room {
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    /**
     * Create a room described "description". Initially, it has
     * no exits. "description" is something like "a kitchen" or
     * "an open court yard".
     */
    public Room(String description)
    {
        this.description = description;
    }
}
/**
 * Define the exits of this room. Every direction either leads
 * to another room or is null (no exit there).
 */

public void setExits(Room north, Room east, Room south, Room west)
{
    if(north != null)
        northExit = north;
    if(east != null)
        eastExit = east;
    if(south != null)
        southExit = south;
    if(west != null)
        westExit = west;
}

/**
 * Return the description of the room (the one that was defined
 * in the constructor).
 */

public String getDescription()
{
    return description;
}
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

class Parser
{
    private CommandWords commands; // holds all valid command words

    public Parser()
    {
        commands = new CommandWords();
    }

    // methods go here
}
public Command getCommand()
{
    String inputLine = "";   // will hold the full input line
    String word1;
    String word2;

    System.out.print("> ");     // print prompt

    BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    try {
        inputLine = reader.readLine();
    } catch(java.io.IOException exc) {
        System.out.println("There was an error during reading: "+ exc.getMessage());
    }

    StringTokenizer tokenizer = new StringTokenizer(inputLine);

    if(tokenizer.hasMoreTokens())
        word1 = tokenizer.nextToken();      // get first word
    else
        word1 = null;

    if(tokenizer.hasMoreTokens())
        word2 = tokenizer.nextToken();      // get second word
    else
        word2 = null;
// note: we just ignore the rest of the input line.

// Now check whether this word is known. If so, create a command
// with it. If not, create a "null" command (for unknown command).

if(commands.isCommand(word1))
    return new Command(word1, word2);
else
    return new Command(null, word2);
/ * 
* This class holds an enumeration of all command words known to the game. 
* It is used to recognise commands as they are typed in. 
* 
* @author Michael Kolling and David J. Barnes 
* @version 1.0 (February 2002) 
*/

class CommandWords 
{
    // a constant array that holds all valid command words
    private static final String validCommands[] = {
        "go", "quit", "help", "look"
    };

    public CommandWords()
    {
        // nothing to do at the moment...
    }

    public boolean isCommand(String aString)
    {
        for(int i = 0; i < validCommands.length; i++) {
            if(validCommands[i].equals(aString))
                return true;
        }
        // if we get here, the string was not found in the commands
        return false;
    }
}
class Command
{
    private String commandWord;
    private String secondWord;

    /**
     * Create a command object. First and second word must be supplied, but
     * either one (or both) can be null. The command word should be null to
     * indicate that this was a command that is not recognised by this game.
     */
    public Command(String firstWord, String secondWord)
    {
        commandWord = firstWord;
        this.secondWord = secondWord;
    
    */
public String getCommandWord() {
    return commandWord;
}

public String getSecondWord() {
    return secondWord;
}

public boolean isUnknown() {
    return (commandWord == null);
}

public boolean hasSecondWord() {
    return (secondWord != null);
}
Some Indications of Poor Design

• High Coupling
  – Changes in one class may result in many changes to other classes
  – We want to avoid this rippling of changes

• Low Cohesion
  – Classes should be more than just a collection of fields and methods that may be unrelated
  – High cohesiveness will lead to code reuse

• Code Duplication
  – If the same code appears in multiple places it will be difficult to maintain since changes must be made in multiple locations
  – Abstracting duplicate code is a cleaner design and improves maintenance
Example of Fixing Code Duplication

• The following code appears in printWelcome and goRoom

```java
    System.out.print("Exits: ");
    if(currentRoom.northExit != null)
        System.out.print("north ");
    if(currentRoom.eastExit != null)
        System.out.print("east ");
    if(currentRoom.southExit != null)
        System.out.print("south ");
    if(currentRoom.westExit != null)
        System.out.print("west ");
    System.out.println();
```

• This can be abstracted to a new method printLocationInfo
Extensions Should be Easy

• If a “common sense” extension is very difficult to implement then the original code is poorly designed

• Consider the change from four exits (N, S, E, W) to six by adding “up” and “down”

• Making this change in the original code would be very difficult

• We solve this problem by encapsulating the exits into a single collection, a HashMap, that can easily be extended without changing the code that processes the contents of the HashMap
Responsibility-driven Design

• Each class should have a well-defined responsibility
• By making Room responsible for information about exits, the code is simplified and expandable
• Here is the new version of getExitString

```java
private String getExitString()
{
    String returnString = "Exits:";
    Set keys = exits.keySet(); // returns keys in a HashMap
    for(Iterator iter = keys.iterator(); iter.hasNext(); )
        returnString += " " + iter.next();
    return returnString;
}
```
Implicit Coupling

• Expanding the list of available commands is fairly easy, suppose we want to add the command look

```java
private static final String validCommands[] = {
    "go", "quit", "help", "look"  
};
```

• A small problem remains in printing the available commands
  – The original list of commands is hard coded
  – The new version is flexible using a method showAll

```java
public void showAll() {
    for(int i = 0; i < validCommands.length; i++) {
        System.out.print(validCommands[i] + " ");
    }
    System.out.println();
}
```
Cohesion for Readability

• It is fairly obvious when code if repeated in different locations, it should be modularized.
• What if code is only called once, why should it be modularized? One reason is for readability.
• Consider the printWelcome method that is only called once, it is modularized to make the calling method more readable.
• In general, if code for a method becomes too lengthy (e.g., can’t be seen on a single screen in the editor), it should be modularized.
Cohesion for Reuse

• Even if in current version of the software only calls a method once, by modularization we encourage reuse

• After the software expands, some new component may use a prior method that has been encapsulated

• Object-oriented programming design tends to produce a lot of very small methods rather than lengthy code segments, that’s why software reuse is much more common with object-orientation
Refactoring

- As an application expands, it is necessary to rethink the overall design at the class level and refactor the design to make expansion “clean”
- The first step is to redo the existing code with the same functionality but with the new design
- The second step is to add the new functionality
- We will examine a particular example of refactoring relative to our adventure game after we examine our second version of the code
- This will be the basis of a lab activity you will complete
A Better Version of “zuul”

- We implement the changes discussed in the previous slides
- The following classes change significantly
  - Game
  - Room
  - CommandWords
- There is a minor change in the Parser
  - The showCommands method is added
- There is no change in the Command class
/**
 * This class is the main class of the "World of Zuul" application. 
 * "World of Zuul" is a very simple, text based adventure game. Users 
 * can walk around some scenery. That's all. It should really be extended 
 * to make it more interesting!
 *
 * This main class creates and initialises all the others: it creates all 
 * rooms, creates the parser and starts the game. It also evaluates and 
 * executes the commands that the parser returns.
 *
 * @author  Michael Kolling and David J. Barnes 
 * @version 1.0 (February 2002)
 */

class Game
{
    private Parser parser;
    private Room currentRoom;
    /**
     * Create the game and initialise its internal map.
     */
    public Game()
    {
        createRooms();
        parser = new Parser();
    }

    // methods go here
}
/**
 * Create all the rooms and link their exits together.
 */

private void createRooms()
{
    Room outside, theatre, pub, lab, office;

    // create the rooms
    outside = new Room("outside the main entrance of the university");
    theatre = new Room("in a lecture theatre");
    pub = new Room("in the campus pub");
    lab = new Room("in a computing lab");
    office = new Room("in the computing admin office");

    // initialise room exits
    outside.setExit("east", theatre);
    outside.setExit("south", lab);
    outside.setExit("west", pub);
    theatre.setExit("west", outside);
    pub.setExit("east", outside);
    lab.setExit("north", outside);
    lab.setExit("east", office);
    office.setExit("west", lab);
    currentRoom = outside;  // start game outside
}
public void play()
{
    printWelcome();

    // Enter the main command loop. Here we repeatedly read commands and //
    // execute them until the game is over.

    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing. Good bye.");
}

private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to Adventure!");
    System.out.println("Adventure is a new, incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentRoom.getLongDescription());
}
/**
 * Given a command, process (that is: execute) the command.
 * If this command ends the game, true is returned, otherwise false is
 * returned.
 */

private boolean processCommand(Command command) {
    boolean wantToQuit = false;

    if(command.isUnknown()) {
        System.out.println("I don't know what you mean...");
        return false;
    }

    String commandWord = command.getCommandWord();
    if (commandWord.equals("help"))
        printHelp();
    else if (commandWord.equals("go"))
        goRoom(command);
    else if (commandWord.equals("quit")) {
        wantToQuit = quit(command);
    }
    return wantToQuit;
}
private void printHelp()
{
    System.out.println("You are lost. You are alone. You wander");
    System.out.println("around at the university.");
    System.out.println();
    System.out.println("Your command words are:");
    parser.showCommands();
}

private void goRoom(Command command)
{
    if(!command.hasSecondWord()) {
        // if there is no second word, we don't know where to go...
        System.out.println("Go where?");
        return;
    }
    String direction = command.getSecondWord();
    // Try to leave current room.
    Room nextRoom = currentRoom.getExit(direction);
    if (nextRoom == null)
        System.out.println("There is no door!");
    else {
        currentRoom = nextRoom;
        System.out.println(currentRoom.getLongDescription());
    }
}
/**
 * "Quit" was entered. Check the rest of the command to see
 * whether we really quit the game. Return true, if this command
 * quits the game, false otherwise.
 */
private boolean quit(Command command)
{
    if(command.hasSecondWord()) {
        System.out.println("Quit what?");
        return false;
    }
    else
        return true;  // signal that we want to quit
}
import java.util.Set;
import java.util.HashMap;
import java.util.Iterator;

/**
 * Class Room - a room in an adventure game.
 *
 * A "Room" represents one location in the scenery of the game. It is
 * connected to other rooms via exits. For each existing exit, the room
 * stores a reference to the neighboring room.
 *
 * @author  Michael Kolling and David J. Barnes
 * @version 1.0 (February 2002)
 */

class Room
{
    private String description;
    private HashMap exits; // stores exits of this room.
    /**
     * Create a room described "description". Initially, it has no exits.
     * "description" is something like "in a kitchen" or "in an open court
     * yard".
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap();
    }
}
/**
 * Define an exit from this room.
 */
public void setExit(String direction, Room neighbor)
{
    exits.put(direction, neighbor);
}

/**
 * Return the description of the room (the one that was defined in the
 * constructor).
 */
public String getShortDescription()
{
    return description;
}

/**
 * Return a long description of this room, in the form:
 *  You are in the kitchen.
 *  Exits: north west
 */
public String getLongDescription()
{
    return "You are " + description + ".\n" + getExitString();
}
/**
 * Return a string describing the room's exits, for example
 * "Exits: north west".
 */

private String getExitString()
{
    String returnString = "Exits:";
    Set keys = exits.keySet();
    for(Iterator iter = keys.iterator(); iter.hasNext(); )
        returnString += " " + iter.next();
    return returnString;
}

/**
 * Return the room that is reached if we go from this room in direction
 * "direction". If there is no room in that direction, return null.
 */

public Room getExit(String direction)
{
    return (Room)exits.get(direction);
}
class CommandWords
{
    // a constant array that holds all valid command words
    private static final String validCommands[] = {
        "go", "quit", "help"
    };

    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {
        // nothing to do at the moment...
    }
}
/**
 * Check whether a given String is a valid command word.
 * Return true if it is, false if it isn't.
 */
public boolean isCommand(String aString)
{
    for(int i = 0; i < validCommands.length; i++) {
        if(validCommands[i].equals(aString))
            return true;
    }
    // if we get here, the string was not found in the commands
    return false;
}

/*
 * Print all valid commands to System.out.
 */
public void showAll()
{
    for(int i = 0; i < validCommands.length; i++) {
        System.out.print(validCommands[i] + "  ");
    }
    System.out.println();
}
Adding Functionality to Zuul

• Once you have designed your network of rooms, the game only becomes interesting by adding functionality, such as
  – Add items that can be picked up and put down
  – There is a limit to the amount a player can carry; this could be based on a weight limit
  – An item can be made “uncarriable” by setting its weight higher than the limit
  – The new commands would be take or drop followed by a second word naming the item
Adding Players to Zuul

• The changes to add items also imply we need to add a Player class since it is a player who carries the items

• Adding a single player in the current version will be very simple but allows for future functionality

• In expanding the game, one might
  – Have a goal that we allow the game to end with the player being a “winner” by accomplishing a goal
  – Also allow the player to fail and end the game, such as falling into a bottomless pit in a room

• Multiple players, possibly fighting each other, may be possible in future versions
Some Possible Expansion

• Add items and players as discussed previously
• Add doors that require opening with a key in order to traverse to a neighboring room
• Add hazards and warnings, such as “there is a breeze” if you are next to a bottomless pit
• Add a transporter room that allows you to jump directly to another room
• Add characters to the game who can communicate information to the player
Your Adventure Game

• Do not get too carried away with this part of the lab; add features one at a time
• Plan ahead in your design of your game setting a winning goal but also allow for failure
• Keep the program running correctly at each stage of development
• Limited, but functional expansion will receive a higher grade than a more elaborate expansion that does not function properly
• Added code should be carefully documented using JavaDoc