TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2015

TRIMESTER 2

SWEN222

Software Design

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Instructions: Answer all questions
All questions are of equal value

Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use a separate answer booklet.

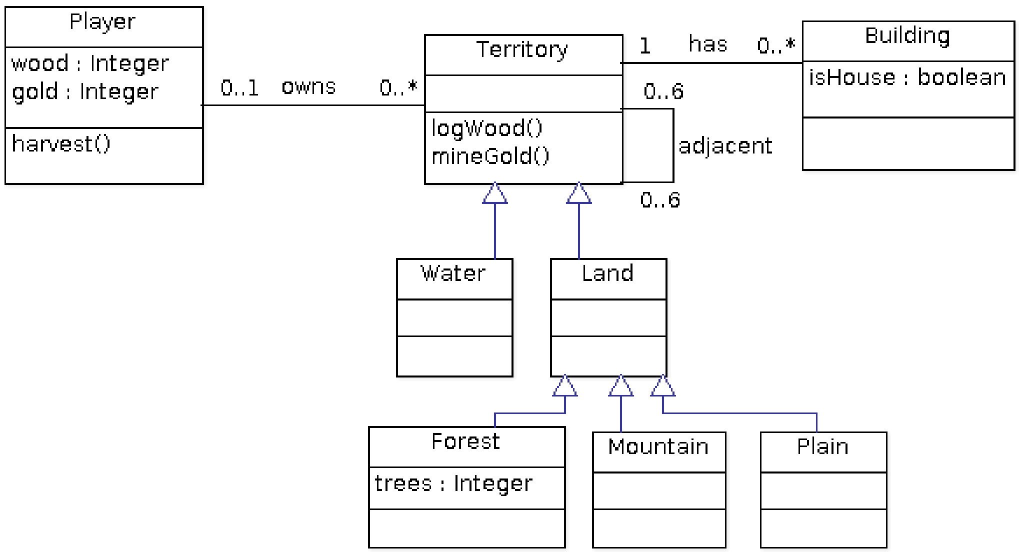
Question	Topic	Marks
1.	Software Design	30
2.	Design by Contract	30
3.	Software Design Qualities	30
4.	Design Patterns	30
	Total	120

Question 1. Software Design [30 marks]

Consider the following (incomplete) description for a simple board game called *Land and Water*.

> "The game board is made up of 37 hexagonal *territories*. Each territory is *adjacent* to at most six other territories. Each territory either represents *land* or *water*. Land territories are further classified as *plains*, *forests* or *mountainous regions*. Forests contain a certain number of trees and, when all trees are chopped down, automatically become plains.
>
> Up to six *players* can play the game. Each player owns one or more territories and can harvest resources of *gold* or *wood* by mining them or logging them. Players can spend their resources by building *houses* or *castles* on territories they own."

(a) [9 marks] Provide a *class diagram* covering those aspects of the game outlined in the description above. Your diagram should contain at most eight classes.



(b) [9 marks] Provide suitable *Class-Responsibility-Collaborator (CRC)* cards describing three classes from the game.

Territory Responsibility	Collaborators
Identifies adjacent territories Can hold houses/castles Allows resources to be harvested	Territory

Player Responsibility	Collaborators
Identifies owned territories Stores accumulated resources Allows harvesting territories	Territory

Forrest Responsibility	Collaborators
Stores remaining number of trees	

(c) [12 marks] Provide a straightforward *Java implementation* for each of the three classes identified in part **(b)**. You need only consider those aspects of the game described on Page 2.

```
class Player {
    private List<Territory> owned = new ArrayList<Territory>();
    private int amountOfGold = 0;
    private int amountOfWood = 0;

    public List<Territory> getTerritories() { return owned; }
    public int getAmountOfGold() { return amountOfGold; }
    public int getAmountOfWood() { return amountOfWood; }

    public void harvest() {
       for(Territory t : owned) {
         amountOfWood += t.logWood();
         amountOfGold += t.mindGold;
       }
    }
}

abstract class Territory {
    private List<Territory> adjacents = new ArrayList<Territory>();
    private List<Building> builds = new ArrayList<Building>();

    public abstract int logWood();
    public abstract int mineGold();

    public List<Territory> getAdjacents() { return adjacents; }
    public List<Building> getBuildings() { return buildings; }
}

class Forest extends Territory {
    private int numberOfTrees;

    public Forest(int trees) { numberOfTress = trees; }

    public int logWood() {
       if(numberOfTrees > 0) {
          trees = trees - 1;
          return 1;
       } else { return 0; }
    }

    public int mineGold() { return 0; }
}
```

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 2. Design by Contract [30 marks]

Consider the following implementation of a Character Buffer, which compiles without error:

```java
public class CharBuffer {
 private char[] buffer;
 private int length = 0;

 public CharBuffer(int max) { buffer = new char[max]; }

 public CharBuffer(char[] buffer) {
  this.buffer = buffer;
  this.length = buffer.length;
 }

 public void append(char c) {
  if(length == buffer.length) {
   // not enough space in buffer!
   char[] nbuffer = new char[buffer.length * 2];
   // copy elements from old buffer to new buffer
   System.arraycopy(buffer,0,nbuffer,0,buffer.length);
   // activate new buffer
   buffer = nbuffer;
  }
  buffer[length] = c;
  length = length + 1;
 }

 public char charAt(int index) {
     return buffer[index];
 }

 // set the character at a given index
 public void set(int index, char c) {
   buffer[index] = c;
 }

 // Return size of buffer's active portion
 public int length() { return length; }
}
```

(a) For each method listed below, provide appropriate *preconditions* and *postconditions*:

(i) [2 marks] `CharBuffer(int max)`

REQUIRES:

ENSURES:

```
requires: max >= 0
ensures: length() == 0
```

(ii) [2 marks] `CharBuffer(char[] buffer)`

REQUIRES:

ENSURES:

```
requires: buffer != null
ensures: length() == buffer.length
ensures: all [ i in 0..buffer.length | buffer[i] == charAt(i) ]
```

(iii) [2 marks] `set(int index, char c)`

REQUIRES:

ENSURES:

```
requires: 0 <= index < length()
ensures: charAt(index) == c
ensures: all other items in this remain unchanged
```

(iv) [2 marks] `length()`

REQUIRES:

ENSURES:

```
ensures: \result >= 0
ensures: all items in this remain unchanged
```

(b) [4 marks] Method contracts can help establish who is to *blame* when an error occurs. Briefly, discuss what this means.

Method contracts provide a contract between the client and implementer. The client must ensure the pre-conditions of the method are met and, if not, is to blame for any resulting fault. Likewise, assuming the preconditions are met, the implementer must ensure the post-condition is met and, if not, is to blame for any resulting fault.

(c) [5 marks] An important aspect of the functional programming paradigm is that methods are *side-effect free*. Briefly, state what this means.

A function is side-effect free if it:

1. Does not modify any state that existed before the function was called.
2. Does not call any methods which are not themselves side-effect free.
3. Does not perform any I/O (e.g. printing to the console, etc).

(d) [5 marks] The precondition/postcondition of a method often itself includes calls to other methods. Briefly, discuss why it is helpful for such methods to be side-effect free.

If a precondition or postcondition has side-effects, then the meaning of these remains unclear. For example, when reasoning about a pre-condition should one take into consideration any side-effects which occur within the pre-condition itself? Likewise, if pre- or post-conditions are evaluated at runtime, should their side-effects propagate into the method itself?

(e) [8 marks] Rewrite the `CharBuffer` class to use a *functional design*.

```java
 public class CharBuffer {
  private char[] buffer;

  public CharBuffer(int max) { buffer = new char[max]; }

  public CharBuffer(char[] buffer) {
      this.buffer = Arrays.copyOf(buffer,buffer.length);
  }

  public CharBuffer append(char c) {
     char[] nbuf = Arrays.copyOf(buffer,buffer.length+1);
     nbuf[buffer.length] = c;
     return new CharBuffer(nbuf);
  }

  public char charAt(int index) {
     return buffer[index];
  }

  // set the character at a given index
  public CharBuffer set(int index, char c) {
     char[] nbuf = Arrays.copyOf(buffer,buffer.length);
     nbuf[index] = c;
     return new CharBuffer(nbuf);
  }

  // Return size of buffer
  public int length() { return buffer.length; }
}
```

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 3. Software Design Qualities [30 marks]

(a) [4 marks] Define what is meant by the term *coupling* in software design.

Coupling is an indication of the strength of the interconnections between the components in a design. Highly coupled systems have strong interconnections, with program units dependent on each other

(b) [5 marks] Describe the *positive* implications of *strongly coupled* object oriented designs.

- Performance
- Efficient integration of strongly related activities
- Simpler to implement initially
- Greater coordination between classes

(c) [5 marks] Describe the *negative* implications of *strongly coupled* object oriented designs.

Maintenance, unpredictable changes may propagate through design Potential loss of reuse opportunity for individual classes Complex designs

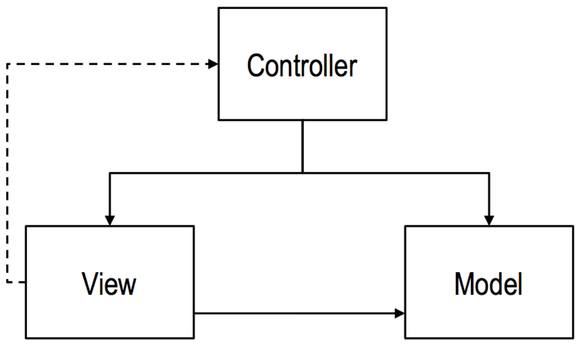
(d) [6 marks] Describe three benefits of *inheritance* in Java software designs.

Code reuse Models domains efficiently Simplifies modification of current classes Compilers can enforce static type checking

(e) [6 marks] Provide a *UML class diagram* of the key features of the COMPOSITE pattern.

(f) [4 marks] Explain the role that inheritance plays in the COMPOSITE design pattern.

Question 4. Design Patterns [30 marks]



(a) [10 marks] Explain what is meant by the phrase "We need **SMART Models**, **THIN Controllers**, and **DUMB Views**."

Controller is more like a coordinator, lightweight functions collecting input without processing for meaning, most information passed through to Model

Views should be consistent with the expected behaviour for the operating system, they should not store information or embody logic

Model should manage all information about the game state and programme behaviour and drive any necessary changes to the controller and view

The code sample provided below provides an implementation of a Controller for a piece of software.

```java
public class GameController implements KeyListener {
  private GameBoard board;
  private GameView view;
  private Point currentLocation;
  private GameObject objectInHand;

  // *** Setup code omitted for clarity ***

  public void keyPressed(KeyEvent e) {
    switch (e.getKeyChar()) {
      // Move up command
      case 'w':
        Point newLocation = new Point(currentLocation);
        newLocation.y += 1;
        GameSquare newSquare = board.getSquare(newLocation);
        if (checkSquareCanBeEntered(newSquare) == true) {
          Point oldLocation = new Point(currentLocation);
          currentLocation = newLocation;
          view.drawPlayerMovingToSquare(oldLocation, newLocation);
        } else {
          view.playInvalidMoveSound();
          view.drawPlayerStandingInSquare(currentLocation);
        }
        break;
      // Pick up object command
      case 'q': {
        GameSquare square = board.getSquare(currentLocation);
        GameObject squareObject = square.getObject();
        if (squareObject == null) {
          view.playInvalidCommandSound();
        } else {
          square.removeObject();
          objectInHand = squareObject;
        }
        view.drawPlayerStandingInSquare(currentLocation);
        break;
      }
      // Drop current object command
      case 'e': {
        GameSquare square = board.getSquare(currentLocation);
        square.addObject(objectInHand);
        objectInHand = null;
        view.drawPlayerStandingInSquare(currentLocation);
        break;
      }
      default:
        view.playInvalidCommandSound();
        view.drawPlayerLookingBored(currentLocation);
    } }

    public boolean checkSquareCanBeEntered(GameSquare theSquare) {
      if (theSquare == null) {
        // Trying to move off the board
        return false;
      } else if (theSquare.type == GameBoard.WALL_SQAURE) {
        // Trying to into a wall
        return false;
      } else if (theSquare.content != null) {
        // Square is already occupied
        return false;
      }
      return true;
    }
}
```

(b) [20 marks] Refactor this code to improve the GameController implementation in line with the statement made in part (a). Please clearly indicate any consequential changes to the GameBoard and GameView classes by providing the relevant code.

A good answer to this question demonstrates clearly that you understand how to design code according to the Model View Controller Design Pattern. In simple terms this means that the Model should contain the State of the system and any logic involved in changing that state. The Controller should act as an interface between the various user interface controls (mouse, keyboard) and the Model using an API that is exposed by the Model using a range of public functions. Finally the view should be able to render the state without acting upon it. Ideally, no logic should be present in the Controller or the View.
Specifically in this case the following problems should be addressed:

- the currentPlayerLocation and objectInHand variables have no place in the Controller
- the logic for each of the actions in the Controller needs to be moved elsewhere, mostly to the Model, but some aspects should be in the View
- the checkSquareCanBeEntered function needs to be moved out into the Model as well

A good answer for the question would look like the code below. Note that the Model is broken up into GamePlayer, GameBoard and GameState classes, this allows a tidier implementation of the View and also opens the possibility that the code might support multiple players. This is not intended to be a complete implementation but would be sufficient to earn full marks marks were not removed for syntax errors, naming issues or similar irrelevancies, the focus was on the MVC Design.

```java
public class GamePlayer {
    // Setup code omitted for clarity
    private Point currentPlayerLocation;
    private GameObject objectInHand;

    // Getters and Setters omitted for clarity

    public void DrawPlayer() {
        // Assumption is that drawing context established by GameView
    }
}
```

```java
public class GameBoard {
    // Setup code omitted for clarity
    private[][] gameSquares;

    public boolean checkSquareCanBeEntered (GameSquare theSquare){
        if (theSquare == null) {
            // Trying to move off the board
            return false;
        } else if (theSquare.type == GameBoard.WALL_SQAURE) {
            // Trying to into a wall
            return false;
        } else if (theSquare.content != null) {
            // Square is already occupied
            return false;
        }
        return true;
    }
}
```

```java
public class GameView {
    // Setup code omitted for clarity
    private GameState gameState;

    public void Update() {
        // Draw the UI
        // Call the draw function of the Board, and thus the Squares
        // Call the draw function of the player(s)
        GamePlayer aPlayer = GameState.getCurrentPlayer();
        aPlayer.DrawPlayer();
    }
}
```

```java
public class GameState {
    private GameBoard gameBoard;
    private GameView gameView;
    private GamePlayer currentPlayer;

    public void MoveUp() {
        Point newLocation = currentPlayer.getCurrentPlayerLocation();
        newLocation.y += 1;
        GameSquare newSquare = gameBoard.getSquare(newLocation);
        if (checkSquareCanBeEntered(newSquare) == true) {
            currentPlayer.setCurrentPlayerLocation(newLocation);
        } else {
            gameView.playInvalidMoveSound();
        }
        gameView.update();
    }

    public void PickUp() {
        GameSquare currentSquare = gameBoard.getSquare(currentPlayer.getCurrentPlayerLocation());
        GameObject squareObject = currentSquare.getObject();
        if (squareObject == null) {
            gameView.playInvalidCommandSound();
        } else {
            currentSquare.removeObject();
            currentPlayer.pickup(squareObject);
        }
        gameView.update();
    }
    public void Drop() {
        GameSquare currentSquare = gameBoard.getSquare(currentPlayer.getCurrentPlayerLocation());
        GameObject squareObject = currentPlayer.drop();
        if (squareObject == null) {
            gameView.playInvalidCommandSound();
        } else {
            currentSquare.addObject(squareObject);
        }
        gameView.update();
    }
    public void InvalidCommand() { gameView.playInvalidCommandSound(); }
}
```

```java
public class GameController implements KeyListener {
    // Setup code omitted for clarity
    private GameState gameState;

    public void keyPressed(KeyEvent e) {
        switch (e.character) {
        // Move up command
        case "w":
            gameState.MoveUp();
            break;
            // Other move commands omitted for clarity
            // Pick up object command
        case "q":
            gameState.PickUp();
            break;
            // Drop current object command
        case "e":
            gameState.Drop();
            break;
        default:
            gameState.InvalidCommand();
        }
    }
}
```

* * * * * * * * * * * * * * *

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID: . . . .

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.