TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2011

END-OF-YEAR

SWEN222

Software Design

Time Allowed: 2 Hours

Instructions: There are 120 possible marks on the exam.
Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use a separate answer booklet.
Non-electronic Foreign language dictionaries are allowed.
Calculators ARE NOT ALLOWED.
No reference material is allowed.

Question	Topic	Marks
1.	Design Patterns	30
2.	Design Quality	30
3.	Design By Contract	30
4.	Class Design	30
	Total	120

Question 1. Design Patterns [30 marks]

(a) [4 marks] Provide an appropriate *class diagram* which describes the COMPOSITE pattern.

(b) [6 marks] An *XML Object* consists of a name `String`, zero or more attribute `String`s and zero or more *children*. Each child of an XML Object is an XML Object and, hence, may also have children.

Sketch an implementation of XML Objects which uses the COMPOSITE pattern.

(Question 1 continued on next page)

(Question 1 continued)

(c) You have been asked to develop a *multi-player chess program* using the *Model-View-Controller (MVC)* pattern. The program has the following requirements:

- Players play the chess game through a *graphical user interface*. This shows the current state of the chess board using images to represent the pieces in the game.
- The program must check every move made by a player is valid under the rules of chess.
- A *client-server* architecture should be used. Players are *clients* who connect to the chess *server*. When a move is made, the client notifies the server which then notifies the other client.

(i) [12 marks] Briefly, discuss each of the three components in the MVC pattern. For each, you should identify which aspects of the chess program it is responsible for.

Model:
The model is responsible for storing the state of the chess game, including the position of pieces on the board, time remaining, etc. The model also implements the rules of chess, and provides methods for checking that a move is valid and updating the board in such case.

View:
The view provides a graphical user interface which shows the current state of the model. The model must be queried to determine the location of pieces on the board, etc. The view also intercepts input from the user using observers and passes this information onto the controller.

Controller:
The controller is responsible for interpreting evens from the view and converting them into commands understandable by the model. In this case, the controller is also responsible for handling the client-server architecture and will co-ordinate the movement of information between clients and the server.

(Question 1 continued on next page)

(Question 1 continued)

Someone suggested implementing a *command-line* version of the chess program. This would draw the chess board using a simple text-based user interface, rather than a graphical user interface.

(ii) [4 marks] Briefly, discuss how implementing a command-line version of the program might lead to a better separation between *Model* and *View*.

(iii) [4 marks] The *Subject-Observer* pattern is often used when implementing a Graphical User Interface. Briefly, outline how this pattern might be used in the chess program.

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 2. Design Quality [30 marks]

(a) Consider the following code fragment:

```
class Block {
  private ArrayList<Stmt> body = ...;
  ...
  public Set<String> usedVariables() {
    HashSet<String> uses = new HashSet<String>();
    for(Stmt s : body) { s.addUsedVariables(uses); }
    return uses;
} }

class LoopBlock {
  private ArrayList<Stmt> body = ...;
  private Expression condition = ...;
  ...
  public Set<String> usedVariables() {
    HashSet<String> uses = new HashSet<String>();
    for(Stmt s : body) { s.addUsedVariables(uses); }
    condition.addUseddVariables(uses);
    return uses;
} }
```

(i) [5 marks] Briefly, discuss how you might improve the above code by *refactoring* it.

(Question 2 continued on next page)

(Question 2 continued)

(ii) [5 marks] *Code Smells* are often an indication that a program should be refactored. Briefly, discuss what this means. You should use examples to aid your discussion where appropriate.

(b) [5 marks] In the context of API Design, Josh Bloch made the following comment:

"*You can always add, but you can never remove.*"

Briefly, discuss the pros and cons of following this rule.

(Question 2 continued on next page)

(Question 2 continued)

(c) Briefly discuss each of the following statements. In each case, explain what the key concept is and justify why you believe the statement is (or is not) true.

(i) [5 marks] *"Well-designed systems have low coupling"*.

(ii) [5 marks] *"Well-designed systems have low cohesion."*

(Question 2 continued on next page)

(Question 2 continued)

(iii) [5 marks] *"Well-designed systems follow the principle of least astonishment."*

Question 3. Design By Contract [30 marks]

Consider the following implementation which compiles without error.

```
public class Train {
 private List<Carriage> carriages = null;
 private int capacity;
 public void createTrain(){ carriages=new LinkedList<Carriage>(); }

 public void add(Carriage newCarriage) {
  carriages.add(newCarriage);
  capacity += newCarriage.capacity;  }

 public void remove(Carriage oldCarriage) {
  carriages.remove(oldCarriage);
  capacity -= oldCarriage.capacity; }

 public int numberOfFullCarriages() {
  int count = 0;
  for (Carriage c: carriages) { if (c.isFull()) { count++; } }
  return count; }
}

public class Carriage {
 public int capacity;
 public int passengers;

 public boolean isFull() {
  return capacity == passengers; }

 public int addPassengers(int num) {
  passengers += num;
  if (passengers > capacity) {
   int added = num - (passengers - capacity);
   passengers = capacity;
   return added;
  }
  return num; }
}

public class GuardedCarriage extends Carriage {
 private int guards;

 public GuardedCarriage(int numGuards) {
  guards = numGuards; }

 public boolean isFull() {
  return capacity == (passengers + guards); }
}
```

(a) [2 marks] What happens if the following piece of code is run?

```
Train train1 = new Train();
Carriage carriage1 = new Carriage();
carriage1.capacity = 50;
train1.addCarriage(carriage1);
```

(b) [6 marks] The code clearly has design problems, and seems to be an example of *Throwaway Code*. Describe two other related software evolution patterns. Your description should include a brief outline of the problem the pattern addresses, and the solution the pattern proposes.

(c) [6 marks] Write suitable *pre-* and *post-conditions* for each of the following methods.

```
public void Train.remove(Carriage oldCarriage);
```

```
public int Carriage.addPassengers(int num);
```

(d) [4 marks] Give suitable *class invariants* for the `Carriage` class. You may use written English if you prefer.

(e) [6 marks] Ideas from the *functional programming paradigm* are commonly used to reduce the complexity of object-oriented software. Functional programming emphasises the use of methods which are *side-effect free*. State which methods in `Train` and `Carriage` are side-effect free, and which are not. For those which are not, briefly state why.

(f) [6 marks] In the context of the Fragile Base Class problem, how can the Carriage and Guarded-Carriage classes be improved?

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: . . . .

Question 4. Software Design [30 marks]

This question deals with the design of a simplified software system for managing University degrees.

- A university *degree* consists of a *schedule* of *courses* that can be used in that degree, and a list of *majors*. A degree requires 360 points to complete, and all courses are worth 15 points.
- A major may appear in multiple degrees and consists of one or more courses that a student must do to satisfy that major. A student who wishes to graduate with a particular degree must satisfy at least one major in that degree.
- A course may appear in the schedules of multiple degrees or majors, and includes a set of prerequisites (which are the names of courses a student must complete first).
- A student may only enroll in a course listed in the schedule of a degree they are attempting to complete. A student may not do more than one degree, although they may do more than one major in the same degree. Once they have completed a degree, they cannot do any more courses. The system maintains the set of courses they have already passed to enforce this.

(a) [12 marks] The four classes listed below form the core of the design for the University degree system. For each class, identify the *collaborators* and *responsibilities* it has.

Degree:

Major:

(Question 4 continued on next page)

(Question 4 continued)

Course:

Student:

(Question 4 continued on next page)

Student ID: . . . .

(Question 4 continued)

(b) [10 marks] Pick two classes from the four identified above and sketch their API. In particular, you should list the methods they contain, and provide appropriate documentation for them. **You do not need to give other implementation details.**

(Question 4 continued on next page)

(Question 4 continued)

The Flyweight pattern is a solution to the problem of having too many very similar objects that consume memory resources. The pattern involves separating intrinsic (context-independent) state and extrinsic (context-dependent) state.

(c) [6 marks] There may be thousands of objects created to represent students and courses in the university degree system. Discuss how you would incorporate the *Flyweight Pattern* into your design to reduce memory consumption.

(d) [2 marks] Imagine the case where courses may instead be worth either 12, 15, 18, 20, 24 or 30 points. Does your answer to question (b) change? If so, why? If not, why not?

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.

****