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

Student ID: .....

EXAMINATIONS — 2012

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 Quality	40
2.	Design Patterns	40
3.	Design by Contract	12
4.	Testing	16
5.	Functional Design and Variance	12
	Total	120

Question 1. Design Quality [40 marks]

(a) [4 marks] **CORE:** Describe form of *refactoring* that *Eclipse* supports.

(b) [5 marks] **CORE:** Discuss the validity of the following claim: "A good API should make implementation details highly visible and accessible."

(c) [5 marks] **CORE:** Discuss the validity of the following claim: "Where possible, classes in an API should be immutable."

(d) [5 marks] **CORE:** Compare and contrast the following uses of inheritance, in terms of whether they are sensible uses of inheritance and whether they are sensible ways of using inheritance in or with APIs:

- *Stack extends Vector*
- *Set extends Collection*

(e) [5 marks] **CORE:** Discuss the validity of the following claim: "Class diagrams and sequence diagrams exist solely to define the requirements and the high-level design, and should be completely discarded once coding has begun."

(f) [5 marks] **EXTENSION:** We often claim that — for class and component design — *less coupling is better*. There are different types of coupling, and historically people have argued that some types of coupling are worse than others.

Some types of coupling are:

- **Control coupling** "Occurs when method A() invokes method B() and passes a control flag to B(). The control flag then "directs" logical flow within B()".
- **Data coupling** "Occurs when methods pass data arguments to other methods."
- **Common Coupling** "Occurs when a number of methods all make use of a global variable."
- **Content Coupling** "Occurs when one component directly modifies data that is internal to another component."

Rank these types of coupling by "badness", and provide a sentence for each justifying your answer.

(g) [5 marks] **EXTENSION:** We often claim that — for class and component design — *more cohesion is better*. There are different types of cohesion, and historically people have argued that some types of cohesion are better than others.

Some types of cohesion are:

- **Communicational** "All methods that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it."
- **Temporal** "Methods are combined into classes based on when they execute, such as combining all the methods required at start-up into a single class, or all the methods that are required to handle a particular error."
- **Sequential** "Components or methods are grouped in a way that allows the first to provide input the next, and so on. The intent is to implement a sequence of operations."
- **Functional** "The component only performs a single computation and returns a result without side-effects."

Rank these types of cohesion by "goodness", and provide a sentence for each justifying your answer.

(h) [6 marks] **CHALLENGE:** CRC cards are an example of a design prototyping tool that is far more primitive (in terms of the technologies used) than the computers and software we have access to. Similarly, throwaway code is an example of a software subsystem that is far less rigourously designed, tested, proven or documented than the software systems we ultimately intend on engineering. Compare and contrast CRC cards and throwaway code, in terms of what the offer the discipline of software engineering, and in how they may lead to advanced, rigourously engineered, software systems.

Question 2. Design Patterns [40 marks]

(a) [8 marks] Draw a *class diagram* of the *Abstract Factory* pattern.

(b) [8 marks] Describe how the *Abstract Factory* pattern can assist with software testing.

(c) [5 marks] Consider the following description of a physics engine software component. The physics engine is a complex piece of software, with over a hundred different cooperating classes. As the designer of the physics engine, we want to ensure that the users of our library/component (i.e. other code in other parts of a larger software system) can access the functionality of the physics engine through a small subset of classes — in this case: World and SolidObject. Identify and describe the design pattern that can be applied so that the component's users can simply refer to the two classes World and SolidObject.

```
class World{
  public void add(SolidObject o){ }
  public void remove(SolidObject o){ }
  public void notifyTimePasses(){ }
  public void applyAcceleration(SolidObject o, float x,float y, float z){
  }
abstract class SolidObject{
  public void setMass(){ }
  public abstract boolean overlapsWith(SolidObject other);
  /*other omitted methods*/
  }
```

(d) [5 marks] A video game software system uses the physics engine, and the video game's overall architecture uses a model/view/controller architecture. Identify which of the model, view or controller should directly access the physics engine, and justify your answer.

(e) [2 marks] A SolidObject can be composed of many other SolidObjects. What design pattern can be used to describe this relationship between different SolidObject instances?

(f) [2 marks] Explain how you would use such pattern in this context.

(g) [9 marks] Provide complete Java code for the implementation of this kind of SolidObject that can be composed of many other SolidObjects.

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. Design By Contract [12 marks]

Consider the following implementation of a bank account, which compiles without error:

```
class Account {

   private final long accountNumber;

   /** balance can go below zero, but not below this value */
   private double overdraftLimit = -1000;

   private double balance = 0;

   public Account(long accountNumber) {
       this.accountNumber = accountNumber;
   }

   public long getAccountNumber() {
       return accountNumber;
   }

   public double getOverdraftLimit() {
       return overdraftLimit;
   }

   public void setOverdraftLimit(double overdraftLimit) {
       this.overdraftLimit = overdraftLimit;
   }

   public double getBalance() {
       return balance;
   }

   public void withdraw(double amount) {
       balance -= amount;
   }

   public void deposit(double amount) {
       balance += amount;
   }
}
```

(a) [2 marks] What class invariant(s) should this account class maintain?

(b) [6 marks] Which methods need preconditions to maintain class invariants? Write the precondition for each of these methods. You can use either formal notation or english.

(c) [4 marks] Write suitable postconditions for the deposit() and withdraw() methods. You can use formal notation or english.

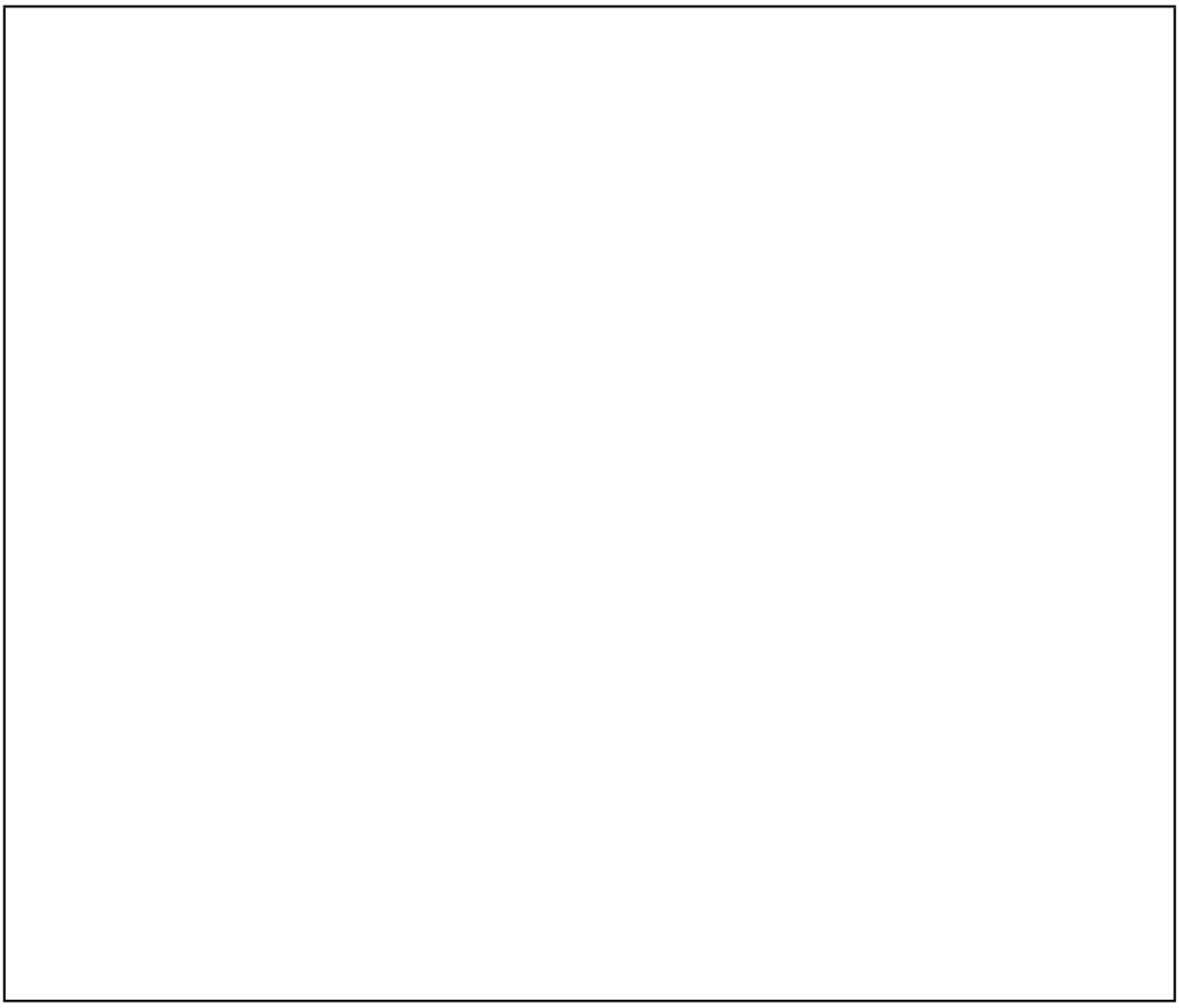
Question 4. Testing [16 marks]

The Bank class describes the methods and fields for a bank in an online banking system. Your team has also written some unit tests for the Bank class.

```
public interface Bank {
   /** return a user's account balance */
   int getAccountBalance(long accountNumber)
        throws BankException;

   /** deposit funds into the user's nominated bank account */
   void deposit(long accountNumber, double amount)
        throws BankException;

   /** withdraw funds from the user's nominated bank account */
   void withdraw(long accountNumber, double amount)
        throws BankException;

   /** transfer funds from the user's account into another account */
   void transfer(long fromAccount, long toAccount, double amount)
        throws BankException;
}

public class BankTransferTest {
   Account account1, account2;
   Bank bank;

   @Before
   public void setUp() {
       account1 = new Account(0); account2 = new Account(1);
       bank = new BankImpl(account1, account2); }

   @Test(throws = InvalidAccountException.class)
   public void testTransferFromInvalidAccount() throws Exception
   {    bank.transfer(10, 1, 10);       }

   @Test(throws = InvalidAccountException.class)
   public void testTransferToInvalidAccount() throws Exception
   {    bank.transfer(0, 10, 10);       }

   @Test(throws = InsufficientFundsException.class)
   public void testTransferMoreThanCreditLimit() throws Exception
   {    bank.transfer(0, 1, 10000); }

   @Test
   public void testTransferSuccessful() throws Exception
   {    bank.transfer(0, 1, 100);       }
}
```

(i) [12 marks] Implement the transfer(...) method so that it passes the tests that they have written. Note that there are additional exception classes called: InvalidAccountException, InsufficientFundsException and the more general BankException that exist, but that aren't explicitly declared here for reasons of space. You can assume that they all have default constructors.

```
public class BankImpl implements Bank {
   private Map<Long, Account> accounts =
       new HashMap<Long, Account>();

   public BankImpl(Account... account) {
      for (Account a : account)
         accounts.put(a.getAccountNumber(), a);
   }
   ...
   public void transfer(long fromAccount, long toAccount,
                        double amount) throws BankException {
```



```
   }
   ...
}
```

(ii) [2 marks] The team member writing the tests missed a cause that could lead to an exploit. What is the exploit, and describe any changes to the API required to prevent the exploit.

(iii) [4 marks] Implement a regression test that will trigger this exploit if it is possible.

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 5. Functional Design and Variance [12 marks]

Integer and Long are examples of classes that use functional design from the Java standard libraries. The following class implements a point class for a game but does not use functional design.

```
/** Class invariant: x and y are positive */
class Point {
   public float x;
   public float y;

   /** Precondition: x and y must be positive */
   public Point(float x, float y) {
       this.x = x;
       this.y = y;
   }

   /** Precondition: v is not null */
   public void add(Point v) {
       this.x += v.x;
       this.y += v.y;
   }
}
```

(a) [4 marks] Demonstrate an alternative implementation of the Point class that demonstrate functional design:

(b) [4 marks] Suppose we wanted to implement a "Velocity" vector class to simulate basic physics. Velocity vectors should be have x/y coordinates but do not need to be positive. Discuss the pros and cons of implementing Vector by extending the point Point, particularly with regards to pre and post conditions on the existing methods, and the Fragile Base Class problem.

(c) [4 marks]

The following two lists use properties of wildcard co-variance and contra-variance to simulate read-only and write-only collections.

```
List<? extends Point> list1;
List<? super Point> list2;
```

Identify which is which and justify your conclusion.

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.

****