FIT2099: Exam Excerpts

Index

Contents

Keeping implementation details of classes private

1 mark: things are private can be changed with confidence that nothing depends on them and will thus break
1 mark: if attributes are not private changes can be made to the state of an object without using its methods, thus making it impossible to enforce rules or policies.
1 mark each for two examples. Possible examples include:

- If the HashMaps were public it would be possible to add or delete students from the enrolment HashMap from outside the class, thus making it impossible to enforce rules such as prerequisites, no duplicate students, etc.
- There are no setters for the unit name and code – they are supposed to be fixed from the moment the object is created. If they were public, they could be changed from outside the class.

Other sensible examples should be accepted.

Risks of repeated code

Repeated code makes software harder to maintain – if it turns out to contain a bug, or is in need of refactoring, the required changes have to be repeated everywhere the code has been copied.
Repeated code makes software harder to test – each repeat could contain a typo or error, and so it all needs to be tested separately.
Repeated code is a source of technical debt.
Repeated code makes software harder to read, because there's more of it.
If the code has been cut-and-pasted because the developer doesn't understand it, then it might not do what is needed (might even have security issues).
The developer might have intended to modify the code after cutting-and-pasting, but forgot to do so. The maintainer must work out whether it really is supposed to be identical to other code in the system.
(Other answers are possible; use your judgement as to whether they are valid.)

1 mark for mentioning a valid point; a further 2 marks for a *clear* explanation of the problem, to a maximum of 6 marks. If the explanation is vague but correct, it is worth 1 mark. Ignore anything about the **advantages** of cut-and-paste reuse; the question is asking specifically for **risks**.

Fixing repeated code

Strategies:

- Refactor
  - make a series of small changes to the code structure
  - test at each step to ensure that the code still works properly
- General
  - identify areas of repeated code
  - if in same class, put repeated code into new method
    - call method wherever the repeated code is
  - if in different classes, decide which class should hold new functionality then proceed as above

Specific refactors to reduce repetition in code:

- methods: extract a method and call it wherever the repeated code appears
- inheritance: if repeated code exists in subclasses, it may be possible to pull it up into the superclass
- generics: if code is repeated to handle different input types, may be possible to make the type a parameter
- others – use judgment

Any of these basic approaches is acceptable.

6 marks: complete and clear explanation, mentioning the need to test/verify that the functionality hasn't been broken

JavaDocs for private attributes

It doesn't matter much, because these attributes are private. They can only be accessed from within the current class, so any programmer who wishes to use them must already be looking at the source code and so will be able to see the non-Javadoc comments.

2 marks if they just say "the attributes are private". Full marks if they explain why this means that Javadoc isn't needed.

Declared Exceptions vs Undeclared Exceptions

- Advantage of declared exceptions: developers of client code can easily see which methods might throw exceptions, and therefore will know where they need to put try/catch blocks. The IDE and compiler can also check that exception handling code exists, which makes it less likely that a thrown exception will end up crashing the program.
3 marks for either of these explanations. Just saying "fewer crashes" or "easier for programmers" without clear reasoning is only worth 1.
- Advantage of undeclared exceptions: less tedious to write. Declarations are shorter. Compiler and IDE won't force you to write exception handling for "can't-happen" cases.

"Less tedious to write", on its own, is only worth 1.

Other valid/reasonable answers are fine – look for evidence that the student understands the principles of development that lead to high-quality software and isn't just thinking about the immediate experience of programming.

Writing Executable Preconditions with Assertions and Exceptions

Assertions: allow the programmer to specify a logical expression which must be true at a certain point in the program. Assertions can be placed at the start of a method to ensure that its preconditions have been met. A possible disadvantage is that assertions need to be enabled via a parameter at the start-up of the Java runtime.

Exceptions: another way to write executable preconditions is to write some logical condition checking for the parameters (e.g. using an if...then structure) at the start of the method, and to throw an exception if a precondition is violated.

Why not to cope with violated preconditions

1 mark for saying that the method can't know the right to do. It can't know which clients will use it – they may not be written until years later. Alternative answer: trying to cope with the problem violates the Fail Fast principle – they client needs to know that it passed an incorrect parameter.
1 mark for saying one of: program could behave incorrectly, problem would be made harder to find because failure would occur further from and later than it would otherwise, or similar.

Access control for methods in preconditions

1 mark for saying "public"
1 mark for explaining why: clients need to be able to check if they are going to violate a precondition. They thus need access to any methods used in the precondition.

Benefits of Writing Preconditions before Implementation

1 mark for saying one of:

- The author of the method will know exactly what it can assume about the parameters of the method
- Provides documentation of anyone writing a client of the method, even before the method exists
- Provides information useful for writing a stub/mock for the method that could be used in testing before the method exists
- Give a mark for other similarly sensible suggestions

1 mark for saying something like: it provides constraints on the parameters that can be known even before the method details have been worked out. For example a method to compute a square root must have only non-negative parameters – we know this before even starting to think about how to write the method.

Benefits of Writing Postconditions Before Implementation

Potential points: before implementation as capturing requirements precisely. It becomes a specification that the implementer can use and test against. In testing and debugging, because it catches problems earlier and gives a clear indication of what the problem is.

3 marks for explanation of when and how.
Must mention at least two points for full marks. Explanation of each much be clear.
Max of 2 marks for a clear explanation of one instance where it is helpful.

Command-Query Separation, Relation to Design by Contract

1 mark for saying "Command-Query separate says that every method should be either a command or a query but not both".
1 mark for saying that pure queries can be used in preconditions and/or post conditions (or "assertions") with the knowledge that turning off these checks will not change the behaviour of the program.
1 mark for saying that if methods used in preconditions/postconditions/assertions change the state of one or more objects, then turning of checks may change behaviour of the program. Alternative answer: commands that return a value risk that value not being checked even it is signalling an error.

Liskov Substitution Principle and Pre/Post Conditions

1 mark for saying "The LSP says that an object of a subclass should be able to be used anywhere where an object of its superclass(es) was expected, and correctness is preserved" (or something with equivalent meaning, but different wording)

1 mark for saying it means that preconditions in subclass methods can only be the same or weaker than those of the corresponding superclass methods – i.e. the subclass method can accept more, but cannot accept less.
1 mark for saying it means that postconditions in subclass methods can only be the same or stronger than those of the corresponding superclass methods – i.e. the subclass method can promise more, but cannot promise less.

Interface in Java

1 mark for saying something like: an interface in Java is a collection of method signatures (declarations) that any class that implements the interface must provide definitions for. An interface contains no non-static method definitions (though in Java 8 default implementations were introduced)
1 mark for a valid example using the scenario above.

Abstract Class in Java

1 mark for saying that an abstract class in Java is a class that (can) contain abstract methods – i.e. method declarations without any implementation.
1 mark for saying it differs from an interface in that it **can contain attributes** and some method implementations. (0.5 mark alternative answer: a subclass can implement multiple interfaces, but only extend one (possibly abstract) class).
1 mark for a valid example using the scenario above.

Interfaces vs Abstract Classes

1.5 marks for explaining advantages and disadvantages of using interface
    e.g. Advantage: can have characters that implement multiple interfaces for multiple behaviours
    e.g. Disadvantage: can't provide method definitions used by all the implementing character classes
1.5 marks for explaining advantages and disadvantages of using abstract class
    e.g. Disadvantage: can't have characters that inherit from multiple abstract classes for multiple behaviours
    e.g. Disadvantage: possible problem if Character already has a superclass – though could be done as a subclass of Character with further concrete subclasses
    e.g. Advantage: can provide method definitions used by all the inheriting character classes
2 marks for justifying their choice

Other sensible suggestions should be rewarded.

Why Reduce Dependencies

1 mark every dependency is a potential point of failure whenever the software needs to be modified (meaning more work and care is needed whenever the code is touched)
1 mark for providing as real-world example

Encapsulation and Access Modifiers to control dependencies

0.5 mark: Encapsulation boundaries provide a "firewall" which dependencies can be prevented from crossing. They can thus be used to prevent certain dependencies from being created.
0.5 mark: Java provides mechanisms to prevent dependencies crossing encapsulation boundaries via access modifiers such as private, protected (and the default, package-private).
1 mark for a valid example in Java code.

What is good abstraction? Abstraction vs Abstract Classes

1 mark: A good abstraction captures the aspects of a real-world concept that must be represented in a program, e.g. the name, address, phone number, etc. of a student in Uni system, but not their eye colour, number dogs owned etc.
1 mark: Differs from abstract classes in that they are a programming construct, about the existence of abstract methods, not about the concept of capturing a good abstraction.

More Specific Stuff

Class Diagram and Sequence Diagram Mark Schemes

UML syntax
3 marks: correct
2 marks: minor errors (e.g. incorrect "object: Class" labels for lifelines) that do not significantly impeded comprehension.
1 mark: significant errors that impede comprehension, or lots of minor errors such that the diagram is significantly harder to understand than it should be.
0 marks: bad enough that it is impossible to understand the functioning of the system.

Semantics
2 marks: plausibly correct semantics (that is, computeDistance repeatedly requests x and y from points to find nearest neighbour, then requests details from closest neighbour, and then a send request is passed to SMTP sender.
1 - in the ballpark.
0 - major errors.

Diagram syntax
8 marks: CD and sequence diagram syntax is correct
6 marks: minor errors, e.g. incorrect "object : Class" labels for SD lifelines or wrong (but still comprehensible) arrow types in CD
4 marks: one diagram correct, one missing; or major problems that impede comprehension such as missing message labels in SD
2 marks: student has drawn some boxes and maybe some lines

Diagram semantics
6 marks: clearly supports required scenario; sequence of messages in SD makes sense; messages are being sent to classes that might reasonably be able to act on them; message passing in SD is reflected in associations in CD;

attention paid to design principles.
4 marks: reasonable attempt but CD and SD inconsistent (or SD missing, etc.). Or: some component of required functionality isn't supported in the model but could easily be added.
2 marks: seems to be modelling something relevant to the question

Explanatory paragraph
6 marks: Paragraph is consistent with the design expressed in the diagrams, makes sense, and addresses design principles seen in class (e.g. DRY, coupling/connascence, code/design smells, abstraction, etc.) A likely example here is use of inheritance to avoid repeated code and switch logic.
4 marks: An explanation that makes sense but does not use design-related terminology correctly. OR: a clear description of functionality that does not adequately address design concepts. OR: something that is mostly good but misses a major point/overlooks an obvious flaw in the design.
2 marks: some useful comment on the design.

Ease of Extending a System

Some plausible issues:

Sending is in a separate class – makes it somewhat easier

Controller directly invokes SMTPSender - dependency inversion principle not applied.

Interface of SMTPSender specifies address as a string rather than an abstract address class, makes it hard to modify if a non-string address is necessary.

Accept any other reasonable issues identified.

3 marks: expresses at least a couple of issues using appropriate terminology and correctly identifies their effect on modifiability.
2 marks: only identifies one issue but talks about it well, or identifies more than one but problems with language or analysis.
1 mark: makes some point relevant to modifiability in this instance.

Code Smells

Smells:

Class Library

- Embedded literals in methods bookLateCharge(…) and magazineLateCharge(…) – "magic numbers" 50 and 100. Hard to find and change if necessary (can't know if this is the only spot they are used; global search-and-replace won't work if the same value coincidentally occurs in multiple places, but with different meanings.
- Use of separate methods of calculating the late charge for different kinds of item is problematic. What happens if a new item (e.g. DVD) is to be offered in future? A new method would be needed.
- Arguably Feature Envy, since constants bookLoanLength and magazineLoanLength are defined in Library, but only used in LoanRecord.

Class LoanRecord

- Use of separate data structures for storing books and/or magazines on loan is bad design. A new HashMap would need to be added every time a new kind of item was offered by the library (e.g. DVD). This leads to…
  - Repeated code needed for adding different kinds of items to the loan record.
  - Repeated code in methods overdueCharge() and contentsDescription(), as it is necessary to iterate over each data structure, doing the same thing for both books and magazines
  - Repeated code in daysOnLoanBook(…) and daysOnLoanMagazine(…) to do the same calculation for the almost identical kinds of item

  All of this repeated code makes it harder to read and understand the code, and much harder to debug – if an error is found, it is necessary to track down all the duplicate code and check and fix that too. It also makes it hard to extend the system – if a new kind of item is to be offered (e.g. DVD), with this design it will require yet more loops containing near-identical code, and more near identical methods.

1 mark for each issue in Library
1 mark of each issue in LoanRecord
NB. (0.5 for mentioning the issue, 0.5 for explanation of why it is bad)

eXtreme Programming (XP) is an agile development method invented by Kent Beck.

Beck argues that while good design is important, *documenting* your design, or modelling your design, is not important. He argues that you should focus on "designing always" while coding, rather than producing any design artefacts before implementation.

In about half a page and in your own words, describe some key advantages and disadvantages to this approach, compared to explicitly documenting your design using UML notations such as class and sequence diagrams.

- Advantages
  - Savings in time (for the short-term) at least
  - Reduces the chance for any redundant design giving more choice to the developers, if done right, the documents should be
- Disadvantages
  - Harder to collaborate with others without proper documentation, people working on the same project would have a harder time getting familiar
  - Harder for others to spot any design smells in your design, may lead to future design flaws
  - Design documents force conscious thinking over what makes a good design, as opposed to this hacking
  - Increases cognitive load

The Java language uses **static typing**. That is, the type of every variable and method parameter, and the return types of methods, must be explicitly specified by the programmer.

Other languages, such as Python and JavaScript, use **dynamic typing**. Variables can hold a value of any type, and methods can return values of any type. At runtime, just before an operation such as an attribute access is performed, the interpreter for the language checks that this will be possible.

Explain in your own words what you think the advantages and disadvantages of static typing are compared to dynamic typing, in the context of developing reliable, maintainable systems. Write about half a page.

- Advantages
  - Type errors are checked at compile time, drastically reducing errors
  - Developer's have a better track of variable types (reducing cognitive load) and having a lower error rate
- Disadvantages
  - Arguably less tedious for the programmer
  - Less verbose
  - Debug cycle is shorter and less cumbersome due to no compilation step

It important to manage dependencies in software design. Encapsulation can be used to help limit dependencies.

(a) Explain what is meant by encapsulation in object-oriented design and how it helps to limit dependencies. **(2 marks)**

Elements of code can be encapsulated at various levels. One example is encapsulation within a method. Variables declared inside a method are local to that method – they cannot be accessed from outside the method.

(b) Give examples of two other levels of encapsulation that are possible in Java. Explain the features of the Java language that allow these levels of encapsulation to be enforced.

(3 marks)

- Encapsulation is the idea of grouping a set of data and operations on the data within a module
- It helps to limit dependencies as it allows things that interact with each other to be grouped together inside an encapsulation boundary and calls that cross the boundary can be properly minimised and controlled

- Within the class and within the package
  Access modifiers allows methods/classes to be set as public/private/protected allowing developers to control what can be accessed or not
  Defensive copying by cloning attributes allow for preventing attributes inside the boundary to be modified illegally

Watch Code

Driver.java

```java
import java.util.ArrayList;

import edu.monash.fit2099.watches.*;

public class Driver {

   public static void main(String[] args) {

      ArrayList<Watch> watches = new ArrayList<Watch>();

      watches.add(new Watch1());
      watches.add(new Watch2());
      try {
         watches.add(new Watch(new int[] {24, 60, -60, 1000} ));
      }
      catch (Exception e) {
         System.out.println("Watch construction failed with message: \n\t"
               + e.getMessage()
               + "\nLet's not bother with this watch for now."
         );
      }

      System.out.println("##############################");
      for (Watch watch : watches) {
         System.out.println("Testing Watch: " +
watch.getClass().getSimpleName());
         watch.testWatch(200);
         System.out.println("##############################");
      }
   }

}
```

Counter.java

```java
package edu.monash.fit2099.counters;

public class Counter {

   private int value = 0;

   public void reset() {
      value = 0;
   }

   public void decrement() {
      value--;
   }

   public void increment() {
      value++;
   }

   public int getValue() {
      return value;
   }

   @Override
   public String toString() {
      return Integer.valueOf(this.getValue()).toString();
   }

}
```

LinkedCounter.java

```java
package edu.monash.fit2099.counters;

public class LinkedCounter extends MaxCounter {

   Counter neighbour;

   public LinkedCounter(int max, Counter neighbour) {
      super(max);
      this.neighbour = neighbour;
   }

   @Override
   public void increment() {
      super.increment();
      if (this.getValue() == 0) {
         neighbour.increment();
      }
   }
}
```

MaxCounter.java

```java
package edu.monash.fit2099.counters;

public class MaxCounter extends Counter {

   private final int max;
   private final String fieldFormat;

   public MaxCounter(int max) {

      if (max <= 0) {
         throw new IllegalArgumentException("Maximum value of a MaxCounter
must be greater than zero.");
      }

      this.max = max;
      // create a format string with the correct field width for this counter
      double fieldWidth = Math.ceil(Math.log10(max));
      fieldFormat = "%0" + String.format("%.0f", fieldWidth) + "d";
   }

   public int getMax() {
      return max;
   }

   @Override
   public void increment() {
      super.increment();
      if (this.getValue() == max) {
         this.reset();
      }
   }

   @Override
   public String toString() {
      return String.format(fieldFormat, this.getValue());
   }
}
```

Watch.java

```java
package edu.monash.fit2099.watches;

import java.util.ArrayList;

import edu.monash.fit2099.counters.*;

/**
 * Implements a watch made up of an arbitrary number of counters linked
together.
 * Each counter has a maximum value. When it reaches its maximum, it resets
its
 * value to zero, and increments its neighbour.
 *
 * @author David Squire
 *
 */
public class Watch {

   static public final int MAX_HOURS = 24;
   static public final int MAX_MINUTES = 60;
   static public final int MAX_SECONDS = 60;
   static public final int MAX_MILLISECONDS = 1000;

   private ArrayList<MaxCounter> counters = new ArrayList<MaxCounter>();

   public Watch() {
       /* needed so no-argument constructors in subclasses can call it.
        * A Watch created using this will do nothing, as it has no
        * counters.
        */
   }

   /**
    * Create a Watch using an array integers that specify the maximum values
of the counters
    * that make up the desired Watch. Elements of the array must be in order
from the most significant
    * counter (e.g. hours) at position {@code 0}, to the least significant
(e.g. seconds) at position
    * {@code maxValues.length - 1}
    *
    * @param maxValues an array of integers that specify the maximum values of
the counters.
    * There must be at least one element in the array, and all the maximum
values must be
    * greater than 0.
    *
    */
   public Watch(int[] maxValues) {

      assert maxValues != null : "Null reference passed to Watch
constructor.";
      assert maxValues.length >= 1 : "Must pass at least one counter maximum
value in array parameter";
```

```java
      MaxCounter lastCounter = new MaxCounter(maxValues[0]);
      this.addCounter(lastCounter);
      for (int i = 1; i < maxValues.length; i++) { // notice we start from 1,
not 0
         // Commented out to demonstrate the Exception thrown by the
MaxCounter constructor
         // assert maxValues[i] > 0 : "Counter maximum values must be greater
than zero. maxValues[" + i + "] = " + maxValues[i];
         MaxCounter thisCounter = new LinkedCounter(maxValues[i],
lastCounter);
         this.addCounter(thisCounter);
         lastCounter = thisCounter; // notice we can assign a LinkedCounter to
a MaxCounter
      }
   }

   protected void addCounter(MaxCounter newCounter) {
      counters.add(newCounter);
   }

   protected MaxCounter getLeastSignificantCounter() {
      return counters.get(counters.size() - 1);
   }

   public void display() {
      String prefix = "";
      for (MaxCounter thisCounter : counters) {
         System.out.print(prefix + thisCounter);
         prefix = ":";
      }
      System.out.println();
   }

   /**
    * Increment the least significant counter of the Watch.
    */
   public void tick() {
      getLeastSignificantCounter().increment();
   }

   public void testWatch(int numTicks) {
      for (int i = 0; i < numTicks; i++) {
         display();
         tick();
      }
   }

}
```

Watch1.java

```
package edu.monash.fit2099.watches;

public class Watch1 extends Watch {

   private Watch2 myWatch2;

   public Watch1() {
      myWatch2 = new Watch2();
   }

   public void tick() {
      myWatch2.tick(); // delegation
   }

   @Override
   public void display() {
      myWatch2.display(); // delegation
   }

}
```

Watch2.java

```
package edu.monash.fit2099.watches;

public class Watch2 extends Watch {

   public Watch2() {
      super(new int[] {MAX_HOURS, MAX_MINUTES});
   }
}
```

Watch3.java

```
package edu.monash.fit2099.watches;

public class Watch3 extends Watch {

   public Watch3() {
      super(new int[] {MAX_HOURS, MAX_MINUTES, MAX_SECONDS,
MAX_MILLISECONDS});
   }

}
```