Office Use Only

| | | |
|---|---|---|
| | | |

MONASH University

# Semester Two 2018
# Examination Period

## Faculty of Information Technology

**EXAM CODES:** **FIT2099**

**TITLE OF PAPER:** Object Oriented Design and Implementation - PAPER 1

**EXAM DURATION:** 2 hours writing time

**READING TIME:** 10 minutes

***THIS PAPER IS FOR STUDENTS STUDYING AT: (tick where applicable)***

☐ Caulfield ☑ Clayton ☐ Parkville ☐ Peninsula
☐ Monash Extension ☐ Off Campus Learning ☐ Malaysia ☐ Sth Africa
☐ Other (specify)

During an exam, you must not have in your possession any item/material that has not been authorised for your exam. This includes books, notes, paper, electronic device/s, mobile phone, smart watch/device, calculator, pencil case, or writing on any part of your body. Any authorised items are listed below. Items/materials on your desk, chair, in your clothing or otherwise on your person will be deemed to be in your possession.

**No examination materials are to be removed from the room.** This includes retaining, copying, memorising or noting down content of exam material for personal use or to share with any other person by any means following your exam.
Failure to comply with the above instructions, or attempting to cheat or cheating in an exam is a discipline offence under Part 7 of the Monash University (Council) Regulations, or a breach of instructions under Part 3 of the Monash University (Academic Board) Regulations.

**AUTHORISED MATERIALS**

| | | |
|---|---|---|
| **OPEN BOOK** | ☑ **YES** | ☐ **NO** |
| **CALCULATORS** | ☐ **YES** | ☑ **NO** |
| **SPECIFICALLY PERMITTED ITEMS**<br>**if yes, items permitted are:** | ☐ **YES** | ☑ **NO** |

***Candidates must complete this section if required to write answers within this paper***

STUDENT ID: __ __ __ __ __ __ __ __ __ DESK NUMBER: __ __ __ __ __

**Question 1.** **[Total Marks 4 + 2 + 2 + 2 = 10]**

Consider the following code for an Java class `Unit` from Java system that represents information about Units, Students, and Marks from a University with structures like those of Monash (continued on following pages):

```
import java.util.ArrayList;
import java.util.HashMap;

public class Unit {

	private String code;
	private String name;
	private HashMap<Integer, Student> enrolledStudents = new HashMap<Integer, Student>();
	private AssessmentScheme assessmentScheme = null;
	private HashMap<Assessment, HashMap<Student, Mark> > Marks
		= new HashMap<Assessment, HashMap<Student, Mark> >();

	public Unit(String newCode, String newName) {
		code = newCode;
		name = newName;
	}

	public void enrolStudent(Student newStudent) {
		enrolledStudents.put(newStudent.getPersonID(), newStudent);
	}

	public void unenrolStudent(Student student) {
		enrolledStudents.remove(student.getPersonID());
	}

	public boolean isEnrolled(Student student) {
		return enrolledStudents.containsKey(student.getPersonID());
	}

	public ArrayList<Student> getEnrolledStudents() {
		ArrayList<Student> students = new ArrayList<Student>(enrolledStudents.values());
		return students;
	}

	public void setAssessmentScheme(AssessmentScheme assessmentScheme) {

		this.assessmentScheme = new AssessmentScheme(assessmentScheme);
		for (Assessment a : assessmentScheme.getAssessments()) {
			Marks.put(a, new HashMap<Student, Mark>());
		}
	}

	public AssessmentScheme getAssessmentScheme() {
		return new AssessmentScheme(assessmentScheme);
	}

	public boolean hasAssessmentScheme() {
		return assessmentScheme != null;
	}


	public boolean hasCompletedAssessments(Student student) {
		boolean hasCompleted = true;
		for (Assessment a : assessmentScheme.getAssessments()) {
			hasCompleted &= Marks.get(a).containsKey(student);
		}
		return hasCompleted;
	}
```

```
	public void markStudent(Assessment assessment, Student student, int score, String
comment) throws Exception {
		/* Start Preconditions */
		// Precondition: studentEnrolledInUnit
		if (! isEnrolled(student)) {
			throw new Exception("Precondition violated: studentEnrolledInUnit");
		}
		// Precondition: scoreInValidRange
		if (
			(score < 0) // too low
			 || (score > assessment.getWeight())  // too high
		   ) {
			throw new Exception("Precondition violated: scoreInValidRange");
		}
		/* End Preconditions */

		Mark mark = new Mark(assessment, student, score, comment);
		Marks.get(assessment).put(student, mark);
	}

	public HashMap<Student, Mark> getMarks(Assessment assessment) {
		return (HashMap<Student, Mark>) Marks.get(assessment).clone();
	}

	public int getUnitMark(Student student) throws Exception {
		/* Start Preconditions  */
		// Precondition: hasCompleted
		if (!hasCompletedAssessments(student)) {
			throw new Exception("Precondition violated: hasCompleted");
		}
		/* End Preconditions */

		int unitMark = 0;
		for (Assessment a : assessmentScheme.getAssessments()) {
			unitMark += Marks.get(a).get(student).getScore();
		}
		return unitMark;
	}

	public String description() {

		return code + " " + name;
	}

}
```

**Question 1. (continued)**

(a) Why is it considered good practice to keep the implementation details of classes private? Give two examples of problems that could arise during software development and maintenance if implementation details were not kept private. Illustrate your answer with examples from the `Unit` class above. **(4 marks)**

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.

(b) Why does the method

```
getEnrolledStudents()
```

return a newly-created `ArrayList`? What would be the risks if this method returned a reference to the existing `enrolledStudents` attribute, e.g.

```
return this.enrolledStudents;
```
? **(2 marks)**

1 mark for saying "to avoid a privacy leak"
1 mark for explaining the risks – i.e. that simply returning a reference to the attribute would mean that there was now a reference to an attribute intended to be private available outside the class. Since HashMaps are mutable this would mean that the contents of the enrolledStudents HashMap could be modified from outside (thus side-stepping any rules)

(c) The precondition code in methods `markStudent(…)` and `getUnitMark(…)` makes use of other method of the class: `isEnrolled(…)` and `hasCompletedAssessments(…)`.

Should methods used in precondition code, such as `isEnrolled(…)` and `hasCompletedAssessments(…)`, be public or private? Explain the reasons for your answer.

**(2 marks)**

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.

(d) If you find duplicated code in an object-oriented program, there are two basic ways of reorganizing the code so that the required code appears in one place only. Explain these two basic ways of eliminating duplicated code. **(2 marks)**

1 mark for saying: move it into a method and call that whenever required
1 mark for saying: if the duplicated code occurs in classes that are conceptually subclasses of some super-type, a method can be created in a superclass and inherited in the subclasses that need it.
(Another possibility: use a generic type to specify a data structure and its methods, so that method code is not duplicate when equivalent data structures containing different types are needed).

**Question 2.** **[Total Marks 3 + 1 + 2 + 2 + 2 = 10]**

*Design By Contract* is an approach to software design that is supported directly in some languages, and which can be applied in others, such as Java, through the use of extensions or other language features. In Design By Contract, the designer of a class tells the clients of the class what the class does and what it needs via the *specification* of the class.

An important principle in Design By Contract is that of *Command-Query Separation*.

(a) What is Command-Query Separation? Why is it useful in Design By Contract? Give an example of a situation where not using Command-Query Separation could cause a problem. **(3 marks)**

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.

Another design principle that is used in Object-Oriented design is the *Liskov Substitution Principle*.

(b) What is the Liskov Substitution Principle? **(1 marks)**

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)

(c) What is the impact of the Liskov Substitution Principle on the rules relating pre- and post-conditions of superclass methods and subclass methods in Design By Contract? **(2 marks)**

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.

(d) Why should a method not to try to cope with a violation of its preconditions, but simply to report the problem to the client that called it? What problems could arise if it did not do this? **(2 marks)**

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.

(e) Why might it be useful to write the preconditions for a method before it is implemented? Why might this be easier than simply writing the method? Illustrate your answer with an example **(2 marks)**

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.

**Question 3.** **[Total Marks 2 + 3 + 5 = 10]**

In you assignments, you worked on a rogue-like game set in the Harry Potter universe. Imagine now that you're working on another rogue-like game, this time set in a world of superheros like Spider-man or Batman. In this game, some, but not all, characters can have one or more superpower. Every superpower is implemented by a different class in the system. Superpowers include becoming invisible, shooting webs, jumping tall buildings, and so on.

To implement this functionality, you decide that those characters with superpowers need to have some kind of standard way to choose and execute one of their superpowers; that is:

- Choose a superpower
- Set up and execute the superpower
    - The requirements for setting up and executing difference superpowers differ. For example, to set up shooting a web, it is necessary first to choose a target. Becoming invisible, however, does not require this.

You are considering the best design for giving characters these abilities. Two alternatives occur to you:

- Creating an interface that all characters with superpowers must `implement`.
- Creating an abstract class that all characters with superpowers must `extend`.

(a) What is an *interface* in Java? Illustrate your answer with a code example for the scenario above. **(2 marks)**

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.

(b) What is an *abstract class* in Java? How does an abstract class differ from an interface? Illustrate your answer with a code example for the scenario above. **(3 marks)**

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.

(c) Explain the advantages and disadvantages of using interfaces and abstract classes to implement the superpower functionality for characters described above. Which would choose, and why? **(5 marks)**

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.

**Question 4.** **[Total Marks: 2 + 2 + 2 + 4 = 10 marks]**

A key principle in software engineering is that we should reduce dependencies between software elements whenever possible.

(a) Explain why is it important to reduce dependencies between software elements whenever possible. Include an example of a real-world problem that could occur if this is not done. **(2 marks)**

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

(b) Explain how encapsulation and language features such as access modifiers can be used to reduce and control dependencies. Provide a Java example that illustrates your explanation. **(2 marks)**

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.

(c) Explain what it means to find a good abstraction for use in a program. How does this differ from the notion of abstract classes? **(2 marks)**

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.

Modern computers can use many different kinds of storage devices (e.g. flash drives, spinning hard drives, solid state disks).

(d) Explain how the Dependency Inversion Principle makes it possible for us to write code to store data to these devices without needing to know what type of physical device we are actually using. Include a UML class diagram to illustrate your answer. **(4 marks)**

1 mark: We can create an abstraction for the storage device that does not depend on any specific hardware implementation
1 mark: Clients wishing to store data can do so via this abstraction without ever needed to know the concrete device subclass being used.
2 marks for valid UML example (1 for basic concept, 1 for correct syntax).

**Question 5.** **[Total Marks: 5 + 5 + 5 + 5 = 20 marks]**

Consider the code below and on the following pages from an information system for a library that currently lends out books and magazines.

```
public class Library {

	private static int nextID = 1;
	public static final int bookLoanLength = 14; // max loan length for books in days
	public static final int magazineLoanLength = 7; // max loan length for magazines
in days

	public Library() {
		// nothing to do in constructor at this stage
	}

	public static int newID() {
		return(nextID++);
	}

	public static final int bookLateCharge(int daysLate) {
		// charge per day late in cents
		return 100*daysLate;
	}

	public static final int magazineLateCharge(int daysLate) {
		// charge per day late in cents
		return 50*daysLate;
	}
}
```

---

```
public class Book {

	private int id; // unique id of product in library
	private String title; // short name to appear in catalogue
	private String author;
	private String summary;

	public Book(String title, String author, String summary) {
		this.title = title;
		this.author = author;
		this.summary = summary;
		this.id = Library.newID();
	}

	public int getId() {
		return id;
	}

	public String getTitle() {
		return title;
	}

	public String description() {
		String description = id + ": " + title + "\n"
			+ "by " + author + "\n"
			+ summary;
		return description;
	}

}
```

```
public class Magazine {

	private int id; // unique id of product in library
	private String title; // short name to appear in catalogue
	private int volume;
	private int number;

	public Magazine(String title, int volume, int number) {
		this.title = title;
		this.volume = volume;
		this.number = number;
		this.id = Library.newID();
	}

	public int getId() {
		return id;
	}

	public String getTitle() {
		return title;
	}

	public String description() {
		String description = id + ": " + title
			+ " (vol. " + volume
			+ ", num. " + number + ")";
		return description;
	}

}
```

---

```
import java.util.Date;
import java.util.HashMap;
import java.util.concurrent.TimeUnit;

public class LoanRecord {

	private HashMap<Book, Date> booksOnLoan = new HashMap<Book, Date>();
	private HashMap<Magazine, Date> magazinesOnLoan = new HashMap<Magazine, Date>();


	public void addBook(Book bookOnLoan) {
		Date now = new java.util.Date();
		booksOnLoan.put(bookOnLoan, now);
	}

	public void addMagazine(Magazine compactDiscOnLoan) {
		Date now = new java.util.Date();
		magazinesOnLoan.put(compactDiscOnLoan, now);
	}

	/* Useful for testing */
	public void addBookWithStartDate(Book bookOnLoan, Date startDate) {
		booksOnLoan.put(bookOnLoan, startDate);
	}
```

```
	public int overdueCharge() {
		int totalCharge = 0;
		for (Book book : booksOnLoan.keySet()) {
			int daysOverdue =  this.daysOnLoanBook(book) - Library.bookLoanLength;
			if (daysOverdue > 0) {
				totalCharge += Library.bookLateCharge(daysOverdue);
			}
		}
		for (Magazine magazine : magazinesOnLoan.keySet()) {
			int daysOverdue =  this.daysOnLoanMagazine(magazine) -
Library.magazineLoanLength;
			if (daysOverdue > 0) {
				totalCharge += Library.magazineLateCharge(daysOverdue);
			}
		}
		return totalCharge;
	}

	public String contentsDescription() {
		String description = "";
		for (Book book : booksOnLoan.keySet()) {
			description += String.format("%s\t%d days on loan\n",
					book.getTitle(),
					this.daysOnLoanBook(book)
			);
		}
		for (Magazine magazine : magazinesOnLoan.keySet()) {
			description += String.format("%s\t%d days on loan\n",
					magazine.getTitle(),
					this.daysOnLoanMagazine(magazine)
			);
		}
		return description;
	}

	private int daysOnLoanBook(Book book) {
		Date now = new java.util.Date();
		// calculate different between loan time and now in milliseconds
		long timeOnLoan = now.getTime() - booksOnLoan.get(book).getTime();
		// convert to days
		long timeOnLoanInDays = TimeUnit.DAYS.convert(timeOnLoan,
TimeUnit.MILLISECONDS);
		return (int)timeOnLoanInDays;
	}

	private int daysOnLoanMagazine(Magazine magazine) {
		Date now = new java.util.Date();
		// calculate different between loan time and now in milliseconds
		long timeOnLoan = now.getTime() - magazinesOnLoan.get(magazine).getTime();
		// convert to days
		long timeOnLoanInDays = TimeUnit.DAYS.convert(timeOnLoan,
TimeUnit.MILLISECONDS);
		return (int)timeOnLoanInDays;
	}

}
```

```java
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class TestLibrary {

	public static void main(String[] args) throws ParseException {

		// Create some library items
		Book cloudstreet = new Book(
			"Cloudstreet", // title
			"Tim Winton", // author
			"Winner of the Miles Franklin Award..." // summary
		);

		Magazine ieeeSoftware1_2 = new Magazine(
			"IEEE Software", // title
			1, // volume
			2 // number
		);

		// add some items to a loan record
		LoanRecord loanRecord = new LoanRecord();
		// add a book with loan starting on 1st of September 2018
		SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy", Locale.ENGLISH);
		Date startDate = sdf.parse("01/09/2018");
		loanRecord.addBookWithStartDate(cloudstreet, startDate);
		// add a magazine with loan starting now
		loanRecord.addMagazine(ieeeSoftware1_2);

		// show description of loan record, and overdue charges
		System.out.println("Loan Description:\n" + loanRecord.contentsDescription());
		float overdueCharge = loanRecord.overdueCharge()/100;
		System.out.println("Overdue Charges: $"
			+ String.format("%.2f", overdueCharge) + "\n");
	}

}
```

**Question 5. (continued)**

(a) What code and design smells do you detect in classes `Library` and `LoanRecord`? Explain the problems you see, and why they are bad. **(5 marks)**

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.

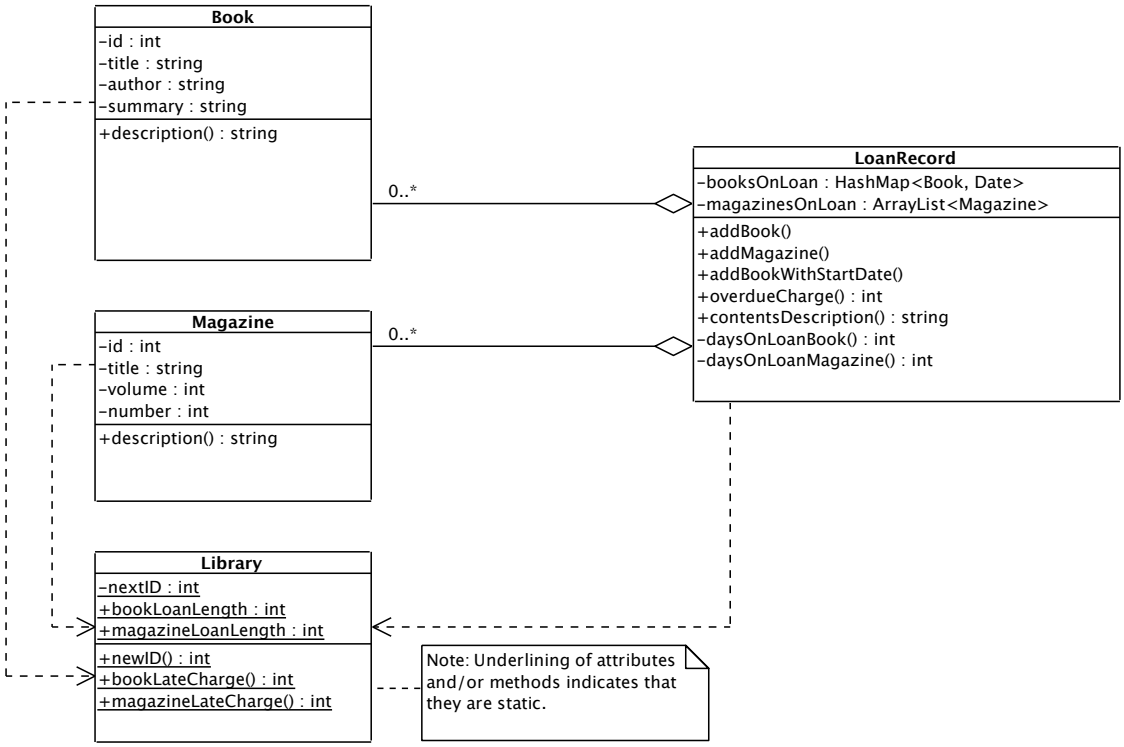
1 mark for each issue in Library
1 mark of each issue in LoanRecord
NB. (0.5 for mentioning the issue, 0.5 for explanation of why it is bad)

(b) Imagine that the library now decides that it is going to start lending DVDs. DVDs have an ID number, a title, a director, and a star. Explain why supporting DVDs would be painful with the current design of the class `LoanRecord`. **(5 marks)**

In order to support a new kind of item for loan, such as DVD, it will be necessary to modify the code in multiple places.
In LoanRecord:
a new data structure, DVDsOnLoan will be needed.
New methods addDVD(…) and daysOnLoanDVD(…) will be needed (though they will be almost identical to the existing methods of books and magazines).
Methods overdueCharge() will need another (almost identical) loop that iterates of DVDsOnLoan
as will contentsDescription().
In Library
a new method DVDLateCharge will be needed – with the appropriate magic number.
3 (1 mark each) for mentioning three of these issues
1 mark for saying that this is painful because it takes time and effort
1 mark for further explaining the pain, e.g. that it is more error-prone, since we must touch a lot of code (including existing code we might break), that it increases the time and effort required for testing, since we need to write new tests, and/or modify existing ones.

**Question 5. (continued)**

The UML class diagram below shows the current design of this part of the system (getter and setter methods are not shown, but can be assumed to exist). Your task is to refactor this design to solve the problems you identified in parts (a) and (b).




(c) Draw a UML class diagram showing your proposed redesign of this part of the system to address the problems you identified in parts (a) and (b) (include a DVD class). Include attributes and methods as above. **(5 marks)**

- Introduces abstract superclass (e.g. Item, Product, LibraryItem or similar) for Book, Magazine and DVD – 1 mark
- LoanRecord modified so that it only knows about Item – 1 mark
- Attributes and methods of Item and its subclasses are correctly distributed: (id and description() in Item, everything specific to the item types in the subclasses) – 1 mark
- Late charge calculation delegated to Item (possibly via Library) – 1 mark
- UML syntax correct – 1 mark

(d) Write a rationale for the design you showed in part (c). Explain how it addresses the code and design smells you identified earlier, and facilitates the extension of the system to support the addition of DVDs to the library's collection. Also explain any other advantages or disadvantages your proposed design. **(5 marks)**

Explanation of how it addresses the smells mentioned in part (a) – 3 marks
Explanation of how it address extension of system – 1 mark
Explanation of other advantages and disadvantages – 1 mark