CS2SE – Week 1: Systems Thinking

Theme

- Analysing of the initial system specification
- Defining the system's boundaries
- Identifying emergent and non-emergent properties

  Key concepts: system boundary, components model, emergent properties

Initial System Analysis

Carefully consider the following initial description of a proposed software system:

Many Aston University students study a "sandwich" course, which means they have to spend their third year on a placement, usually working in industry.

A Placements Tutor at Aston oversees the process. While on placement, the students are supported by their Personal Tutor, who may be different from the Placements Tutor. The Personal Tutor usually makes at least one visit to the student's workplace during the year.

The software system should support the following two processes:

1. A student provides placement details and an administrator creates the student's placement record using these details.
2. Each potential Personal Tutor views available placements on a map and indicates which placements they prefer to visit. The Placements Tutor allocates Personal Tutors to placements, making the allocation fair while respecting the tutors' preferences as much as possible. The software system helps the Placements Tutor by suggesting an efficient and fair allocation based on placement locations and tutors' preferences.

a) Identify FOUR **actors** of the proposed system.

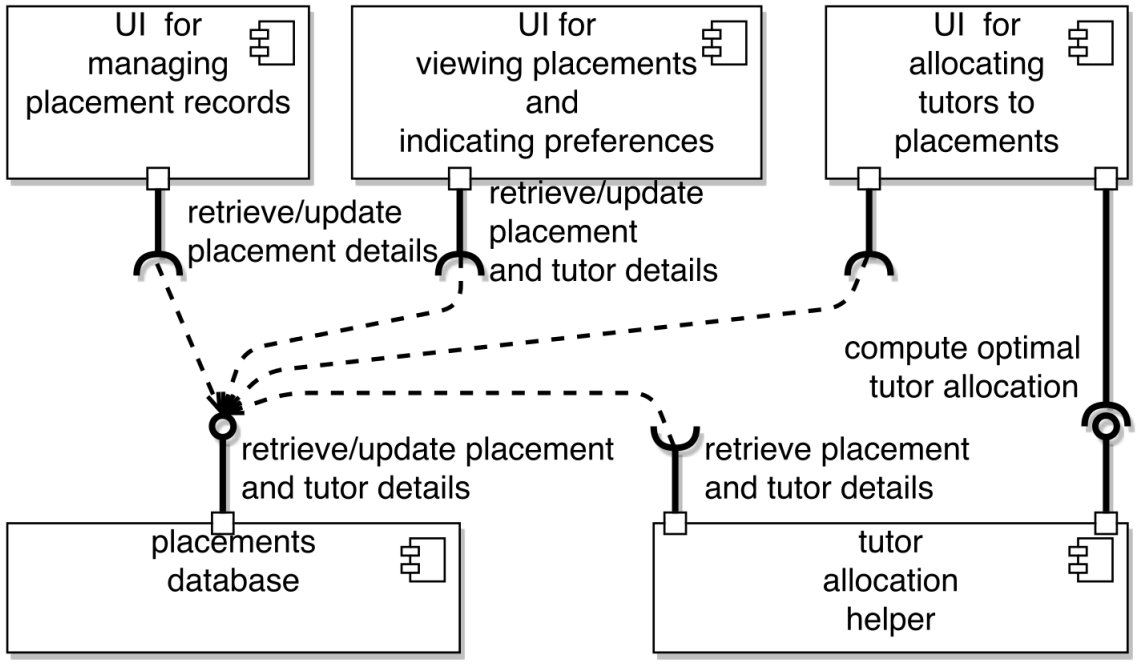
Model answer:

(An actor is someone or something that interacts with the system but is not a component of the system)

- Student
- Administrator
- Personal Tutor
- Placements Tutor
- Web map feature service (e.g., Google Maps)
- University student database
  This is very likely an actor, but we are not sure from the specs; if all student data is entered manually, there is no student database actor!)

b) Identify components suitable for implementing the proposed system and depict them using a **UML component diagram**.

Model answer:



c) Identify FOUR distinct potential important **emergent properties**/behaviours specific to this system.

Model answer:

- The system can suggest a tutor allocation that is fair and meets tutor's preferences.
- The system facilitates communication of up-to-date placement details to all tutors.
- The details of each placement held by the system can be accessed only by the student, the administrator and the tutors.
- The system completes each valid request within 1 second even when it keeps 1000 placements and is used concurrently by 100 Personal Tutors.

d) Identify TWO distinct potential **important non-emergent properties**/behaviours specific to this system.

Model answer:

- The system is coded in Java.
- The human actors can use the system via any recent major browser.
- The system does not keep any unencrypted data in a persistent storage. (Assuming this is true for each of the components, independently of whether the data exchanged by the components is encrypted.)
- The system can hold placement details. (Holding as a behaviour separate from entering the data into the system's storage. Entering the data is an emergent behaviour.)

CS2SE – Week 2: Activity Diagrams

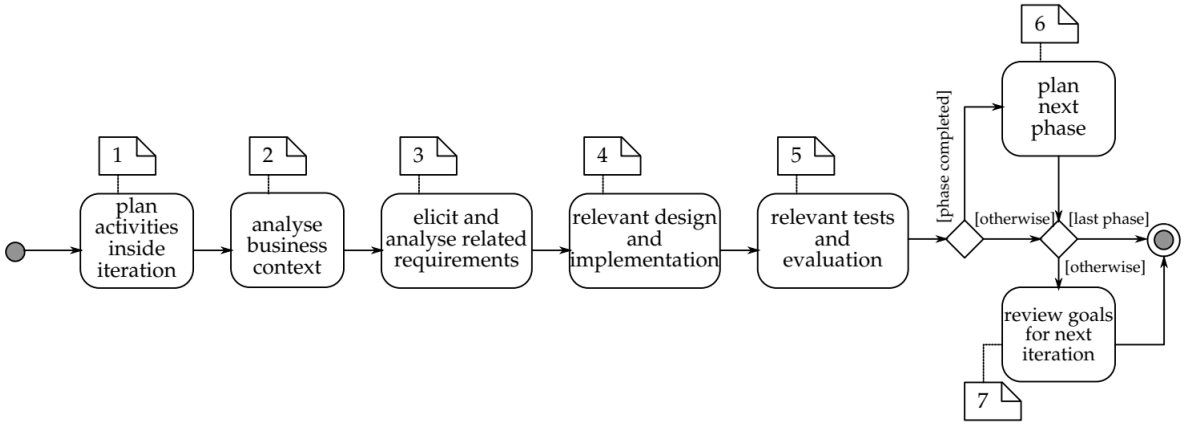
Theme

- Reading activity diagrams with control and data flow
- Making small changes to activity diagrams
- Designing iterations for the four phases of the Unified Process

  Key concepts: activity diagram, control flow, data flow, Unified Process phases, iterations, prototypes, risks

Reading and Modifying an Activity Diagram

1. Consider the following **activity diagram** that attempts to show some aspects of a single **Unified Process** iteration:



Note that the diagram contains logical mistakes, i.e., it shows a model that is not consistent with reality.

a) What sequences of activities does the diagram allow? Consider all possible sequences of activities that this model permits (i.e., legal paths through the diagram, from start to end). Write down each sequence using the activity numbers provided. For example, 1,2,3,4,5,6,7 is one of the paths to consider.

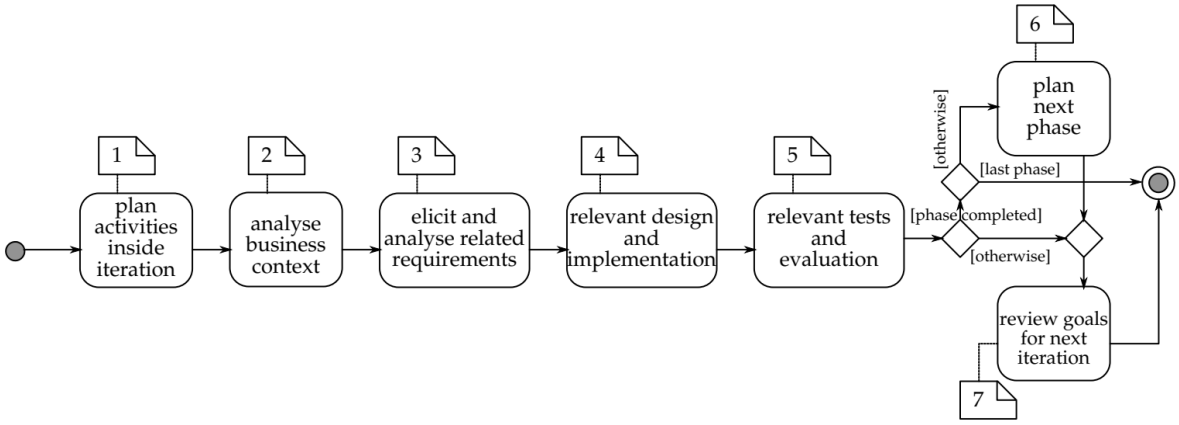
Model answer:

Sequence 1: 1, 2, 3, 4, 5
Sequence 2: 1, 2, 3, 4, 5, 6
Sequence 3: 1, 2, 3, 4, 5, 7
Sequence 4: 1, 2, 3, 4, 5, 6, 7

b) One of the permitted sequences of activities does not make sense and should be forbidden. Fix the diagram and then check that the meaningless sequence is no longer legal and all the other sequences remain legal.

Model answer:

The meaningless sequence is sequence 2 where the next phase is planned even if it does not exist. Moreover, the branching conditions are wrongly wired so that the process could finish in the middle of a phase without any planning. The following modification fixes both problems:



Unified Process phases and iterations

2. Consider the following preliminary system specification:

   Develop an intruder alarm system controlled by one computer with a control panel, connected to a number of sensors, an alarm bell and a telephone line. The system is activated and deactivated by entering a four digit code. The activation code can be changed using the panel when the system is inactive. After activation, the system monitors the sensors. At first it does not react to any sensor signals but when the signals stop occurring and have been absent for 1 minute, the system will react to any further signal by setting off the bell and making a phone call to a designated number. The designated number and the phone message can be changed using the panel when the system is inactive.

   a) Identify FIVE-SIX major **communication and technical risks** of developing the software component of the system and prioritise these risks. *Focus on what could be difficult or go wrong during development, not after development when the system is in operation.*

Model answer:

A communication risk could be a potentially difficult or poorly understood requirement. A technical risk could be a difficult algorithm, design choice or choice of tools or libraries. Perhaps some tools/libraries could be difficult to learn, or it may be unclear which is best suited for a task. Examples include but not limited to:

- technology for receiving signals from sensors
- technology to make a phone call
- recording a voice message
- intuitive user interface
- maintainable architecture supporting all use cases
- operation through power cuts

b) Outline a plausible potential list of iterations for developing the specified system using **Unified Process**. Cover all four phases indicating which iterations belong to which of the four phases. *The plan is expected to have 10–15 iterations.* For each iteration, very briefly specify its objectives and deliverables.

Model answer:

Main objectives (deliverables in italics) for Inception, Elaboration, Construction and Transition phases:

I.1 get sample and users and engineers (*collaboration contracts*)

I.2 identify and prioritise risks (*elaboration plan*)

E.1 identify typical scenarios (*detailed scenario descriptions*)

E.2 sensor monitoring and bell control (*prototype that rings bell when movement detected*)

E.3 making phone calls (*prototype making a call to a fixed number with a fixed message*)

E.4 recording voice message (*prototype with this functionality as before*)

E.5 intuitive user interface (*prototype of control panel + user manual*)

E.6 maintenable system design supporting all use cases (*basic prototype supporting all use cases with some simulation, component/class diagrams demonstrating good decomposition*)

E.7 coping with power cuts (*prototype with backup battery able to operate all use cases through a power cut*)

C.1 alarm start and stop (*polished user interface to this feature, reliable operation*)

C.2 new code entry (*like before*)

C.3 new phone number and message entry (*like before*)

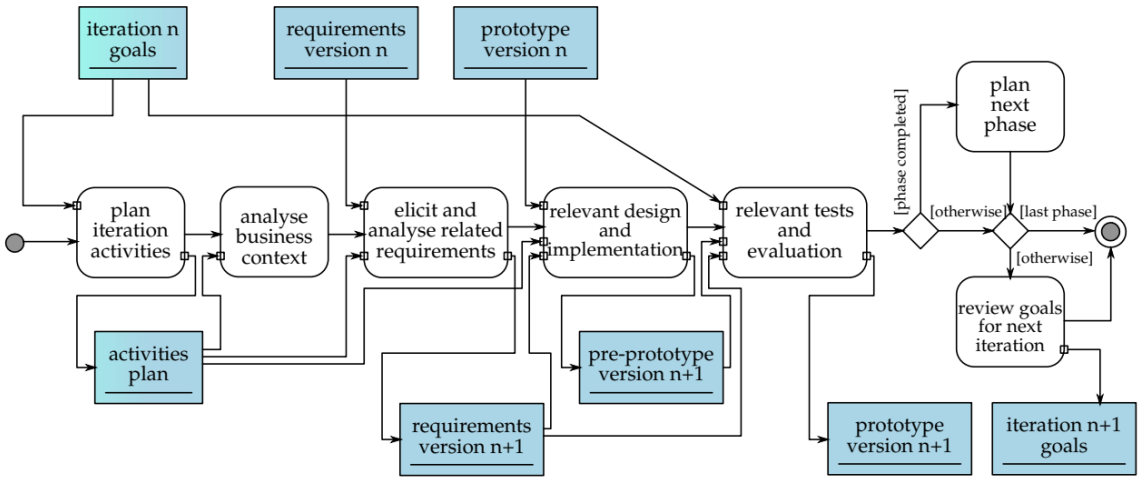
T.1 safety standards (*prototype and documentation required to pass industry safety standards*)

T.2 product documentation (*further installation and maintenance manuals*)

T.3 end user testing (*system installed in a number of houses, end user reports*)

Further modifications of the activity diagram (optional)

3. The following is a version of the activity diagram from Question 1, enriched with information about important objects required and/or produced by activities:



a) Specify how to modify this diagram so that it also shows an evolving plan of future iterations. *Note that the diagram contains the same mistakes as the diagram Question 1. Assume this diagram will be fixed analogously. You are not required to fix this diagram. Also note that instead of a direct transition from one activity to another, sometimes an activity creates an object that triggers the next activity.*

Model answer:

An “iterations plan n” object can exist, similar to the “prototype version n” and “requirements version n”, to be used as input for either “plan next phase” or “review goals for next iteration” actions.

Subsequently, the receiving action will produce an output object called “iteration plan n+1” which will be used in the next iteration/plan.

CS2SE – Week 3: Use Cases

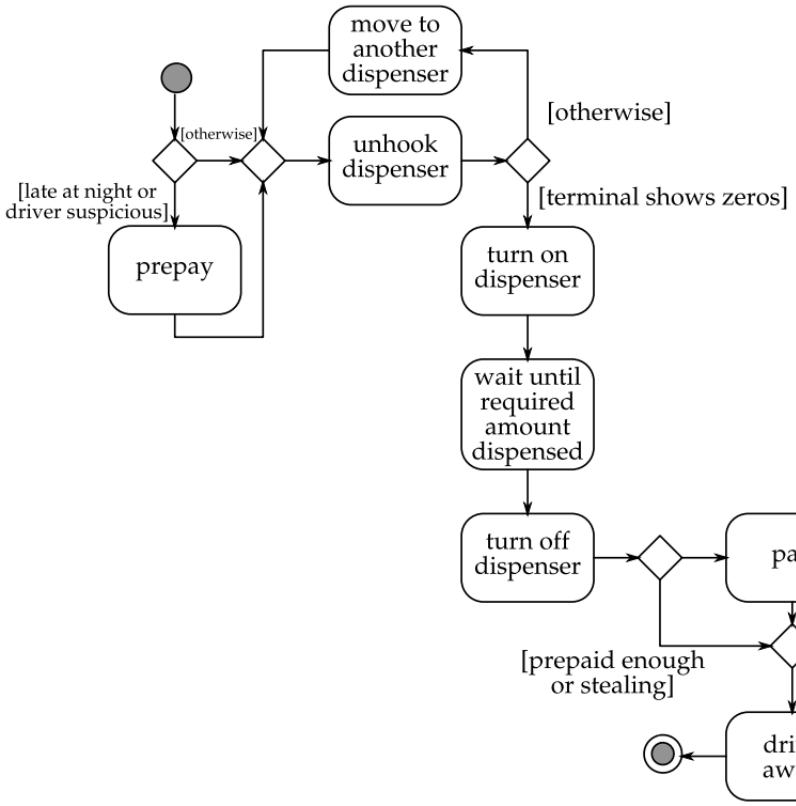
Theme

- Identifying and describing use cases
- Drawing use case diagrams with includes/extends relations

  Key concepts: use case diagrams, includes/extends relations, use case descriptions

Identifying and describing use cases

1. Consider the process of refuelling a vehicle at a fuel station, described by the following **activity diagram**:



Assume your company have been commissioned to develop a central controller for the devices present in a fuel station. You choose to perform use case analysis on the fuel station system to help you structure the development.

a) Identify **actors** and **use cases** for a fuel station system that supports the process shown in the above **activity diagram**.

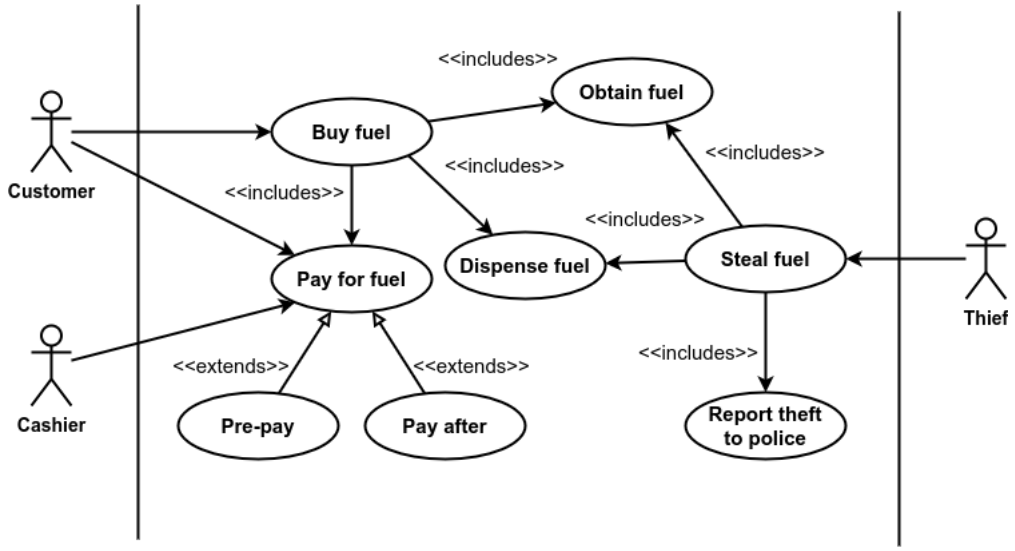
Model answer:

Actors: driver, fuel station cashier, fuel station operator, thief, police
Use cases:

- legally refuel a vehicle (the whole process)
- pay before refuelling (one step)
- dispense fuel (some steps)
- pay for fuel (one step)
- steal fuel (another view of the whole process)
- arrest a thief

b) Draw a **use case diagram** showing the identified **actors** and **use cases** and all appropriate relationships among them. Aim for approximately 4-6 use cases and 3-6 relationships among them, including at least one **inclusion** and one **extension**.

Model answer:



Notice the generalisation arrow for “pre-pay” and “pay after” use cases.

c) Select ONE **use case** that includes or optionally includes (i.e., extended by) another **use case** and describe it appropriately on the supplied **use case description** template (found on Blackboard).

CS2SE – Week 4: Object relationships and state machines

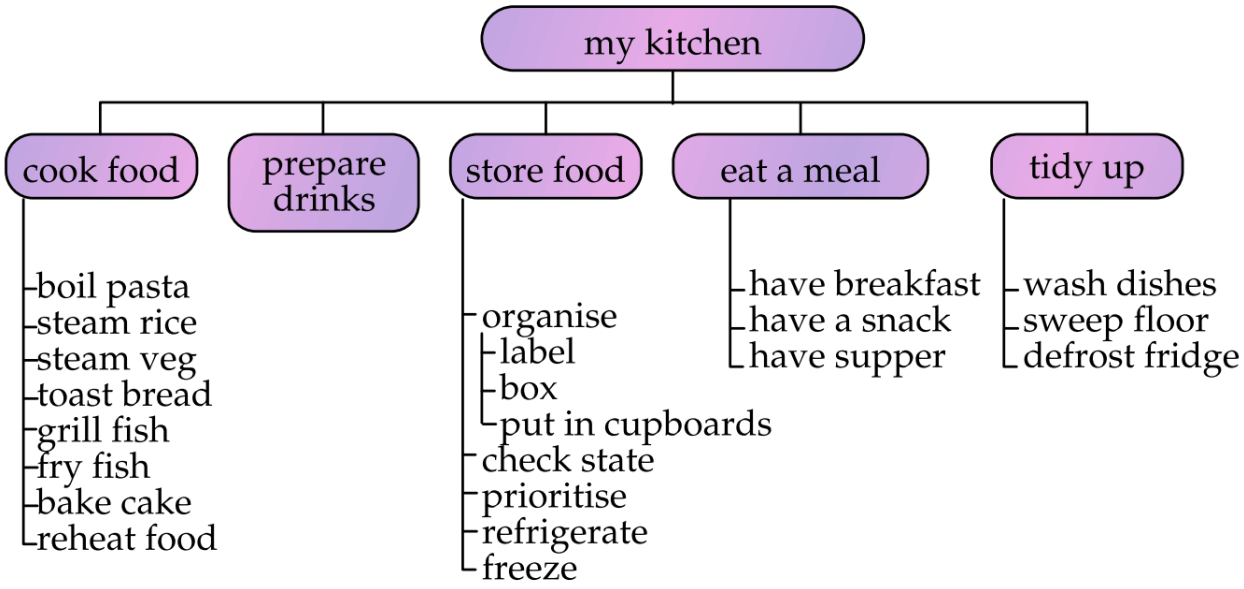
Theme

- Discovering objects and object-object relationships
- Reading state machine diagrams
- Making small changes to state machine diagrams

Key concepts: object relationships, composition, aggregation, object state, state machine diagram

Identifying object relationships

1. Consider the following process map of a certain kitchen:



Please bear in mind task (b) when working on task (a).

a) Pick one of the processes shown in the **process map** of a certain kitchen and identify 4–5 classes of objects that play an important part in this process and have sufficiently many associations among themselves for task (b).

Model answer:

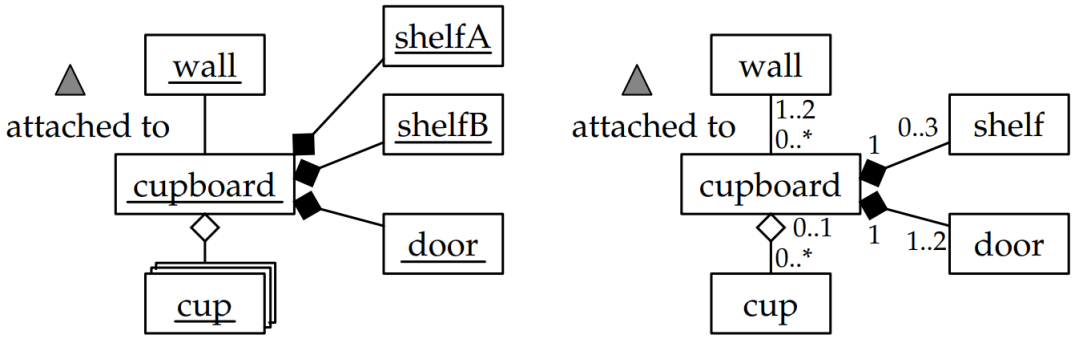
e.g. For “organise” under “store food”: cupboard, shelf, food item, label, box

b) Identify among these objects at least:
   - ONE **composition**,
   - ONE aggregation that is not **composition**,

– TWO associations that are not **aggregations**.

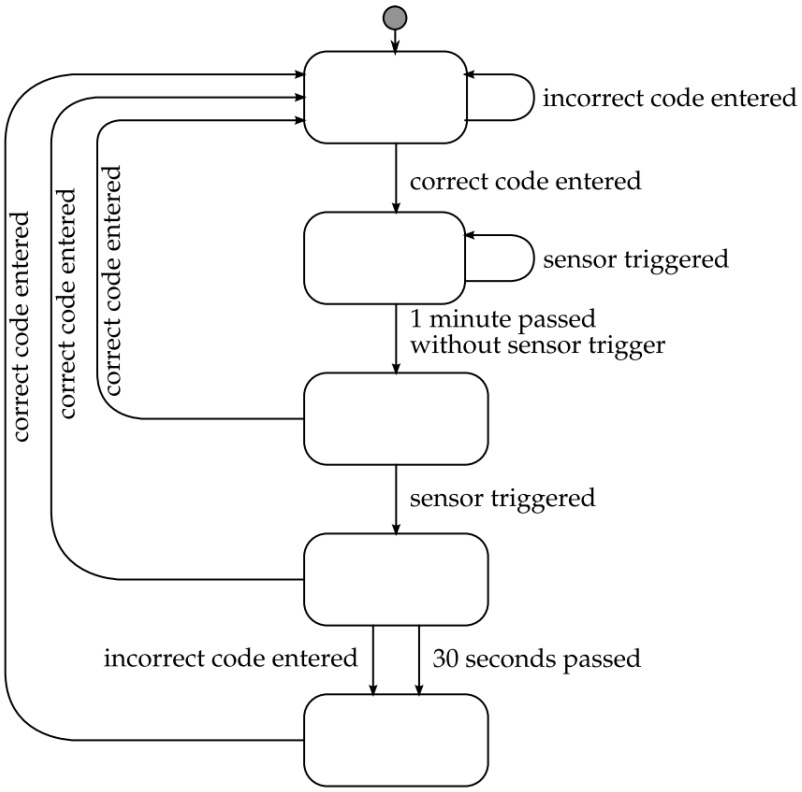
Draw ONE **object diagram** showing all these relationships.
Draw ONE **class diagram** showing these relationships with correct multiplicities.

Model answer:



Reading and modifying a state machine diagram

2. Consider the following state machine diagram that shows some aspects of an intruder alarm system:



a) Suggest helpful names for the **states**.

Model answer:

From top to bottom: idle, pre-activated, activated, triggered, ringing

b) Add a few actions to the diagram, e.g., indicating when the alarm starts ringing and when it stops ringing, and when the controller beeps.

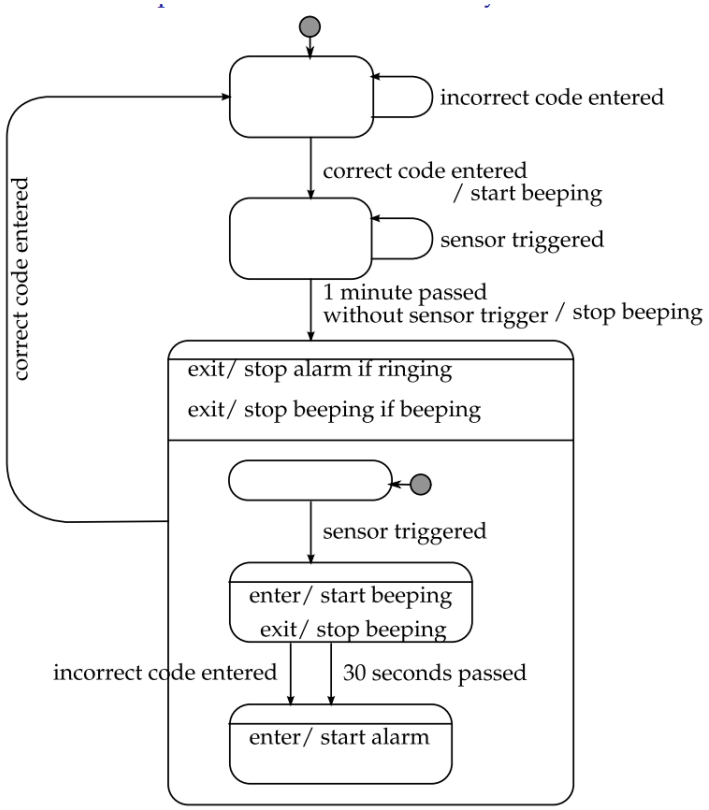
Model answer:

Labels included in the model answer for d). E.g., beep, start beeping, stop beeping.

c) Suggest a way to simplify this diagram using a **composite state**.

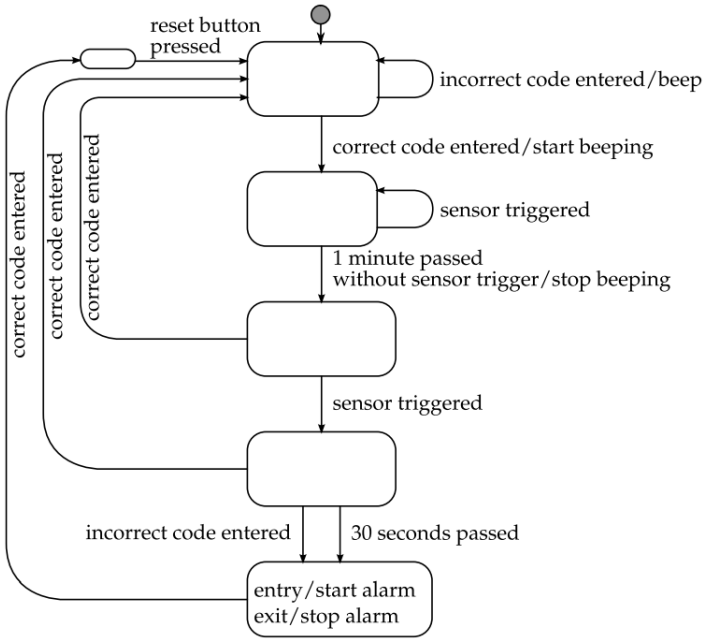
Model answer:

The last three states can be grouped into one composite state “active”. The only way to exit this state is by entering the correct code. Instead of having three such transitions, with the composite state there will be only one of them.



d) Modify the diagram so that is shows the fact that one has to press a reset button after an alarm.

Model answer:



CS2SE – Week 5: Deriving sequence and class diagrams

Theme

– Deriving sequence diagrams from a use case description

– Deriving an outline class diagram from sequence diagrams via communication diagrams

Key concepts: systems analysis, sequence, communication, class diagrams

Introduction

This tutorial is about deriving a class diagram from a use case related to the ICANDO Chemicals business we looked at during the lectures.

ICANDO Chemicals has a number of sites to produce and store chemicals. They want to ensure safety, and thus their sites need to comfort to rigorous safety standards. To decide the best layout for their sites, they ask you to develop a drawing tool that allows engineers to quickly draw a site, including all the storage facilities, offices, boundaries and roads.

The software system must:

- Store site maps, support their creation and editing.
- Store safety rules, support their loading.
- Check whether site maps conform to rules.
- Visually report any rule breaches found.
- Produce audit reports as proofs that sites comply with rules.

ICANDO Chemicals has an IT department, which has already made some progress in conducting domain analysis and feasibility study. For instance, they drafted a business process map as shown in Figure 1.

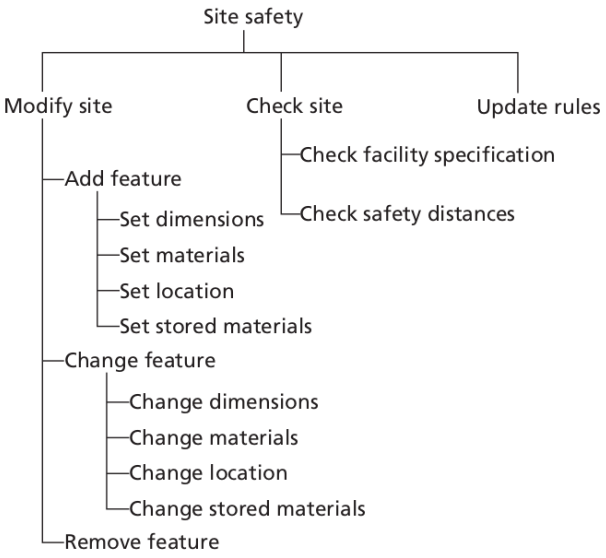


Figure 1: An initial business process map.

They also drafted a UML use case diagram, as shown in Figure 2, and a prototype interface depicted in Figure 3.

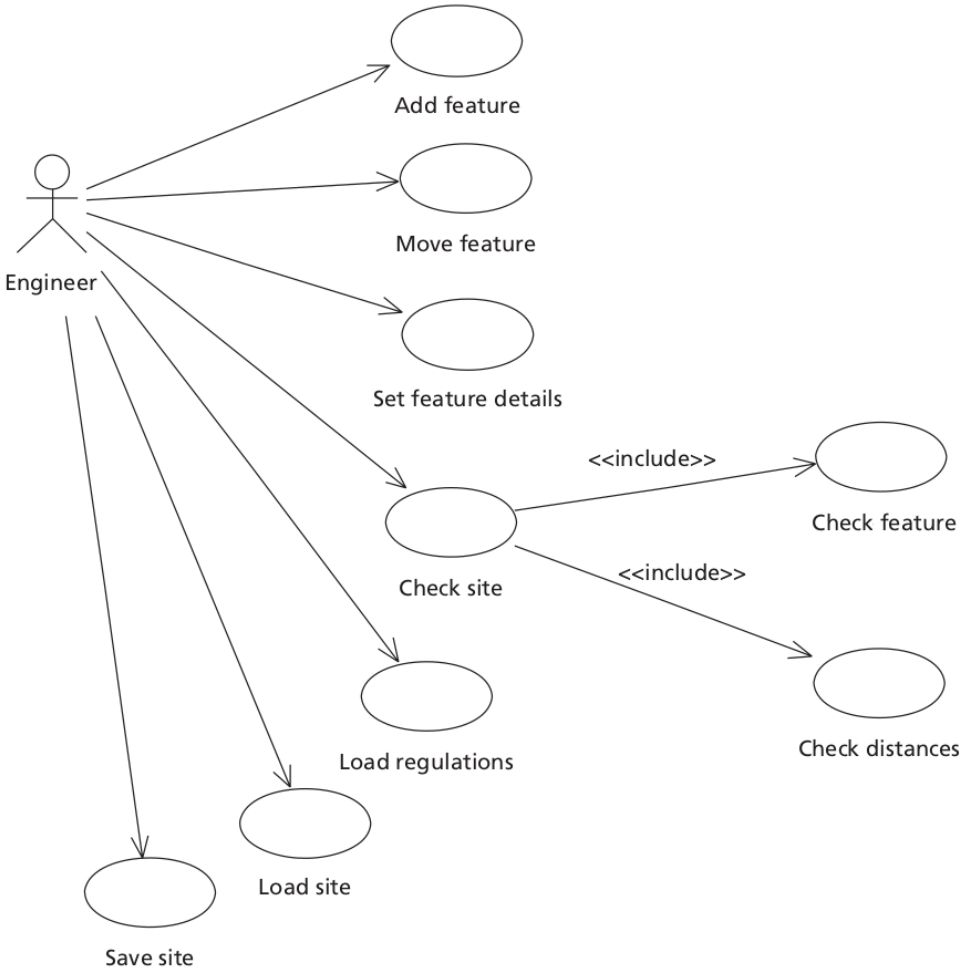


Figure 2: A draft use case diagram.

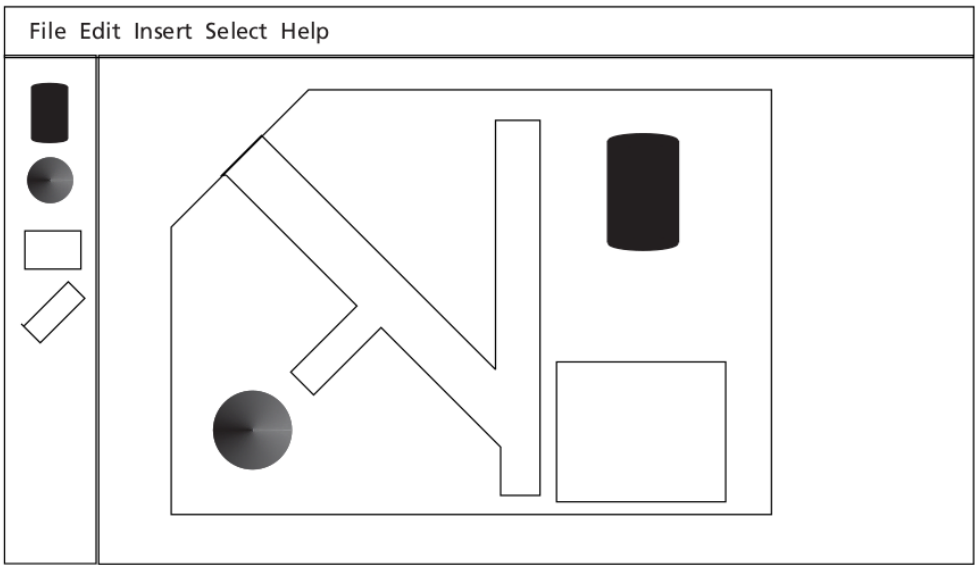


Figure 3: A prototype of the system's interface.

You quickly noticed that a use case “delete feature” was missing, and, after consulting the stakeholders (including the ICANDO Chemicals IT department), you prepared a use case description for it, summarised below:

Use case number :5	**Name**: delete feature
Goal : To delete a facility from the site.	
Brief description: The engineer can delete a facility by clicking on the facility, and hitting the delete key or choosing “delete” from the drop-down edit menu	
Actors : site engineer	
Frequency of execution : up to 5 times per minute (when making frequent mistakes)	
Scalability: once at a time	
Criticality: Not too critical as the same can be achieved by pen on a printout and by saving and reloading intermediate versions.	
Other non-functional requirements: none identified	
Preconditions : A site map is loaded into the system and has at least one feature in it.	
Postconditions : The selected feature is no longer associated with the site in the repository.	
Primary path 1. User clicks on the feature on the backdrop. It is highlighted. 2. User hits the delete key or chooses “delete” from the drop-down edit menu. 3. The system disassociates the item from the site and removes it from the screen.	
Use cases related to primary path : none	
Alternatives 1.1 User repeats 1 for more features. Continue with 2 and repeat 3 for all selected features.	
Use cases related to alternatives : none	
Exceptions 2.1 User hits escape key. The use case is aborted.	
Use cases related to exceptions: none	
Notes :none	

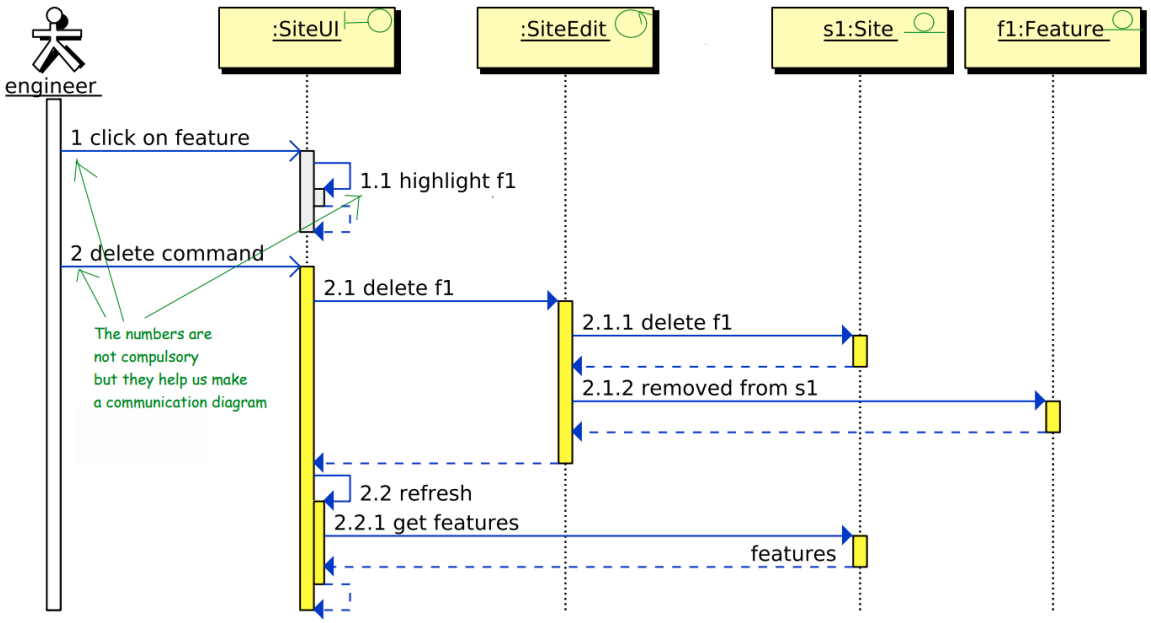
Figure 4: The description for the “delete feature” use case.

Deriving sequence and class diagrams

1. Consider all the materials presented in the Introduction section.

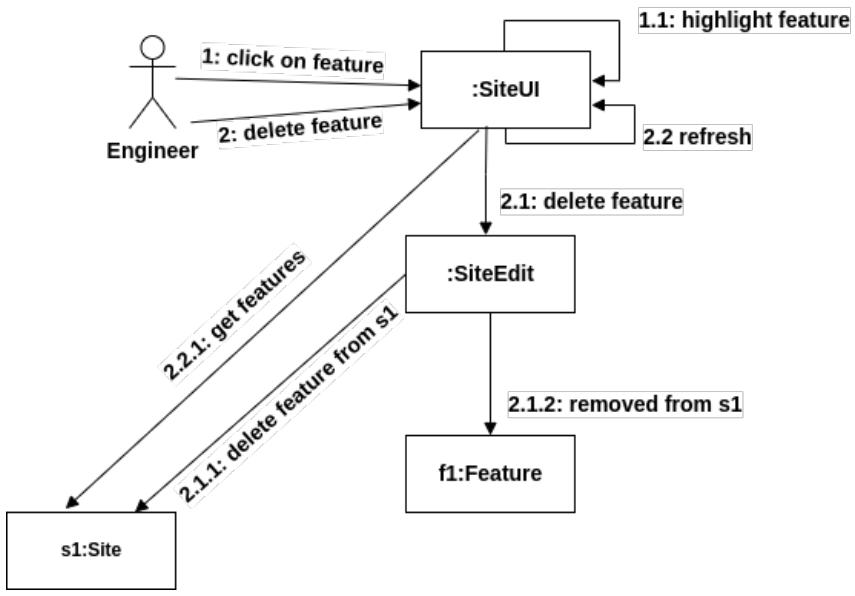
   a) Derive a systems analysis sequence diagram showing the primary path from the use case description.

Model answer:



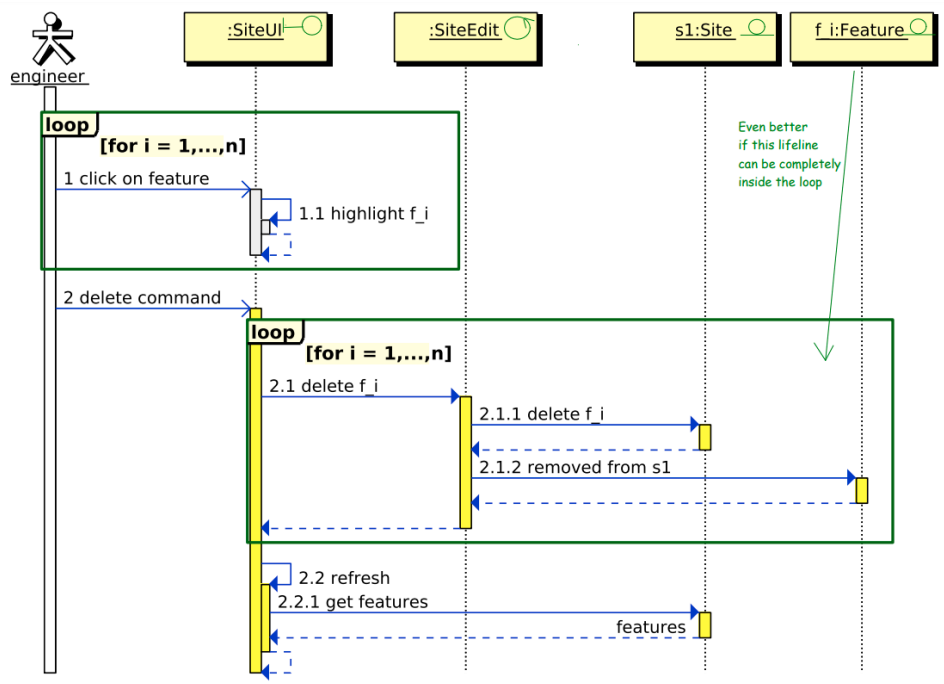
   b) Derive a communication diagram from the sequence diagram from task (a).

Model answer:



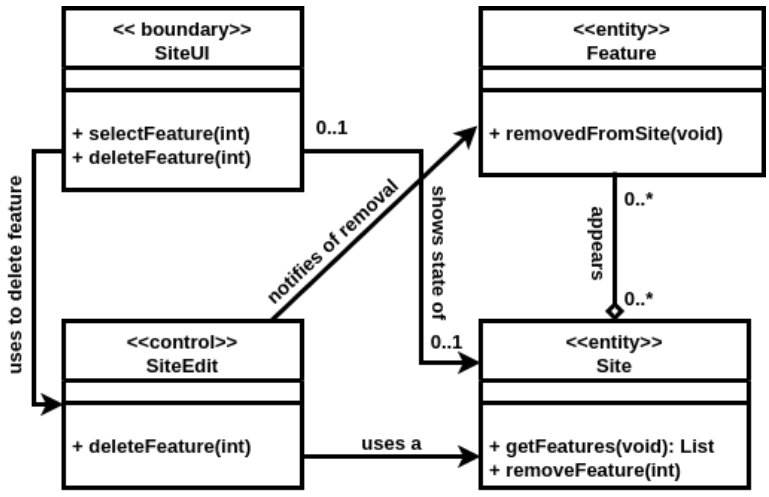
c) Derive a systems analysis sequence diagram showing the alternative path from the use case description. You can extend the diagram from task (a).

Model answer:



d) (Optional) Derive an overview class diagram from the communication diagram. You can include other information evident from the use case description.

Model answer:



CS2SE – Week 6: Software Testing

Theme

- writing test cases
- writing test classes using JUnit

  Key concepts: test cases, unit testing, JUnit

Writing Test Cases

Consider the following specification for the Java application Matcha!

Matcha! is a simple card-matching game. At each game of Matcha!, the player is presented with N piles of M cards. Each pile of cards is kept in a card holder. Only the card at the top of each pile is displayed for the player to see. If $N-1$ displayed cards are the same, a Matcha! state is reached and the matching cards are removed from the piles *automatically*. The removed cards are given to the player for keeping. The goal of this game is to find as many matching sets of cards as quickly as possible.

A user can start the Matcha! game application at command prompt (i.e. without using any IDE) by specifying two game parameters:

a) the number of card piles in the game as an integer (i.e. N), and

b) the number of cards in each pile as an integer (i.e. M).

If the specified game parameters will not lead to a *plausible game*, the application will simply halt. A game that is *plausible* means that:

- The game begins with N piles of cards. Each pile is made up of M cards.
- A game of N piles will always begin with $\frac{N \times M}{N-1}$ different cards.

A user can also start the Matcha! game application at command prompt without specifying any game parameters. In that case, the game will e initialised with the default configuration (i.e. 3 piles, each with 6 cards).

1. In general, what kinds of test cases should test data cover?

   Model answer:

   - Extreme values: very large numbers or strings.
   - Borderline: 0, -1, 0.999.
   - Invalid combinations of values: wrong type, typing out a number instead of the number itself.
   - Nonsensical values: e.g. negative number.
   - Heavy loads: can it handle 1000 concurrent accesses?

2. Which kinds of test cases mentioned in part (a) are relevant for testing the Matcha! game application described above?

Model answer:

- Extreme values
- Borderline values
- Invalid combinations
- Nonsensical values

No obvious performance requirements as the game is single player, thus no need to consider heavy loads.

3. Write three test cases for testing the requirements on starting a Matcha! game. Each test case should include the following information:

   - a description of the test
   - test environment and configuration
   - test data
   - expected outcomes

   Use the following template:

Description	Test Enviroment & Configuration	Test Data	Expected Outcome

Model answer:

Description	Test environment & configuration	Test data	Expected outcome
Starting a default game by running the `Matcha!` application	Perform the test at command prompt.	`java matcha`	A `Matcha!` game with 3 piles of 6 cards would be started. A user should be able to complete the game.
Starting a plausible user-defined game	Perform the test at command prompt.	`java matcha 5 4`	A `Matcha!` game with 5 piles of 4 cards would be started. A user should be able to complete the game.
Attempt to start an implausible user-defined game	Perform the test at command prompt.	`java matcha 4 5`	No `Matcha!` game would be started. An error message would be displayed on the console.
Attempt to start a user-defined game with 1 game parameter	Perform the test at command prompt.	`java matcha 5`	No `Matcha!` game would be started. An error message would be displayed on the console.
Attempt to start a user-defined game with > 2 game parameters	Perform the test at command prompt.	`java matcha 5 4 hello`	No `Matcha!` game would be started. An error message would be displayed on the console.
Attempt to start a user-defined game with non-numeric game parameters	Perform the test at command prompt.	`java matcha five four`	No `Matcha!` game would be started. An error message would be displayed on the console.

Unit Testing with JUnit

Consider the Java classes `Budget`, `BudgetItem` on Blackboard. These Java classes model a partial Java application for managing a student's weekly spending budget.

Class `Budget` keeps track of a list of budget items. A client can:

- add a budget item (a budget item is expected to be in a known category: utility, rent, food, transport, entertainment)
- record the spending of an existing budget item

`BudgetItem` models the detail of each budget item including category of the budget, budget amount and actual expenditure. The budget amount is set when a `BudgetItem` object is created. A client can view the detail and record an expenditure.

4. Write a JUnit class for testing method `recordSpending` in class `BudgetItem`.

Model answer:
Found on Blackboard: `SampleBudgetItemTest`

5. Making use of test fixture, write a JUnit class for testing method `addBudgetItem` in class `Budget`. The tests should check that:

   - a new known budget item can be added
   - a budget item that is already in the budget item list cannot be added
   - a new, but unknown, budget item cannot be added

Model answer:
Found on Blackboard: `SampleBudgetItem`

6. Further Challenges (Optional)

   a) In Eclipse, create JUnit classes for testing `Budget` and `BudgetItem`.
   b) Run the tests.
   c) Fix the issues in `Budget` and `BudgetItem` identified by the test cases.

Model answer:

Adding lines such as:
assert amount $>$ 0 : ”amount cannot be negative”;
Fixing `recordSpending` such that
spending += (int) (amount * 100);

Tutorial T07 and T08

Theme

- Evaluate the trustworthiness of software
- Consider how to design a system in order to meet a measurable objective
- Understand how to implement MVC in Java

Key concepts: trustworthy software, measurable objectives, software architecture, client-server, Model-View-Controller (MVC)

1. Case Study 1: An information system for a property management company

A residential property management company manages various types of residential properties (e.g. houses, flats) on behalf of leaseholders and landlords. Each property is managed by a property manager. Each property manager looks after a portfolio of properties and his/her tasks include performing regular site visits, arranging maintenance and repair, handing keys to new tenants, etc. The normal business hours of the company are Monday to Friday from 08:30 to 17:30.

When there is a fault within a property (e.g. a faulty light switch, a faulty lift, an overgrown garden, pest infestation), the resident reports the fault to a property manager by phone, email or post. Upon receipt of a fault report, the property manager arranges repairs to be carried out as soon as possible. If a fault occurs within a communal area of a property, a property manager may receive multiple reports about the same fault.

a) Suppose you have been tasked to design an online computer system for residents to report faults to property managers.

Name *three* design qualities that are very important and cannot be compromised. Briefly explain your answer.

Model Answer:

Quite a few of the named qualities could be important: what matters is their relevance to the context of this case study, and your explanation of that relevance. Examples could include:

- *Functional*: the system needs to perform its required functions, if it does not work, the clients will become frustrated and lose confidence in the company. This will affect the company's reputation and make the company lose business.
- *Reliable*: the system must not be prone to hardware or software failure and it will deliver the functionality when the users want it; otherwise, in addition to customer frustration, there may be delays in repairs or wasted resources (time and money).

- *Secure*: the system must be protected against errors, attacks and loss of valuable data because residential property management is a highly competitive business. The clients' data must not be lost to competitors, and the company will have to be sure to comply with regulations such as GDPR.
- *Usable*: the system must provide users with a satisfying experience where they can easily find the information they need and be supported in carrying out their tasks. As it is a client-facing system, if the clients are not satisfied, the company will not do well.

b) Suppose you have been tasked to design an online computer system for residential property management.

Briefly describe *three* potential ways to achieve the following *measurable objective*: **Each property manager will be able to process 20% more fault reports.**

Model Answer:

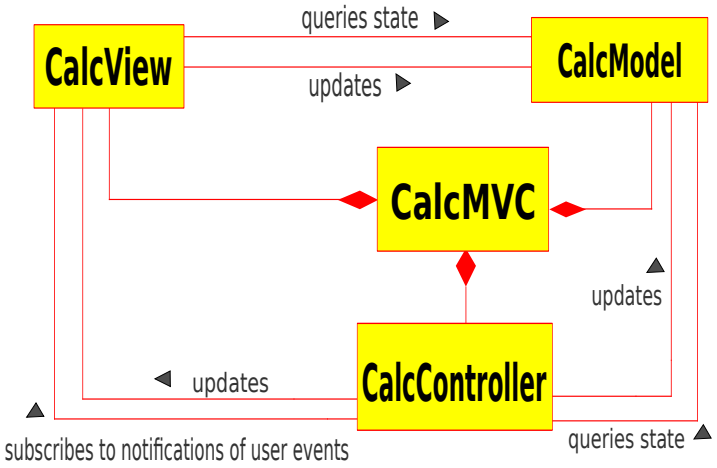
The best way to achieve this objective (without unfairly overloading staff!) would be to ensure that some or all of the fault reports become faster to deal with. Looking at the case study, there are a number of steps in receiving, routing, managing and completing a fault report, and a range of ways in which these might be streamlined or automated to make the process more efficient, and to avoid delays. Possible answers could include:

- Design for accurate *automatic* routing of fault reports to the responsible property manager.
- Design for as many fields in the fault report to be filled with default values as possible.
- Design for rapid response from database.
- Design for *(semi-)automatic* categorisation or processing of fault reports.
- Set up a pre-defined list of potential contractors for each type of job, and allow this list (and contact details) to be easily updated. in the database, and present the relevant list in a context-sensitive way to the property manager.
- Design for automatic identification and removal of duplicate fault reports (i.e., the same fault, in the same location, reported by different people).
- Use natural language processing technique(s) to automatically extract job specification from fault reports.

2. Evaluating Model-View-Controller (MVC) Implementation

The Java classes in the appendix claim to implement the same Java "multiplier" application that was discussed in Unit 8 and demonstrated in the video on Blackboard. This code claims to represent a Java realisation of the *Model-View-Controller* (MVC) architecture. The UML class diagram below describes the class relations between the given Java classes. *NB: The diagram does not include the fields (attributes) nor the operations (methods) compartments.*

Look carefully at the diagram and make sure that you understand how it relates to the original Java code. Your task is to evaluate whether the given diagram and code conforms to the *MVC architecture* discussed in Unit 8.

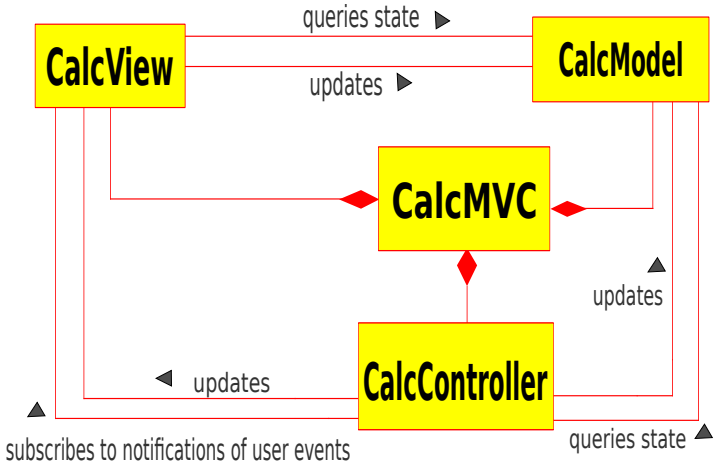


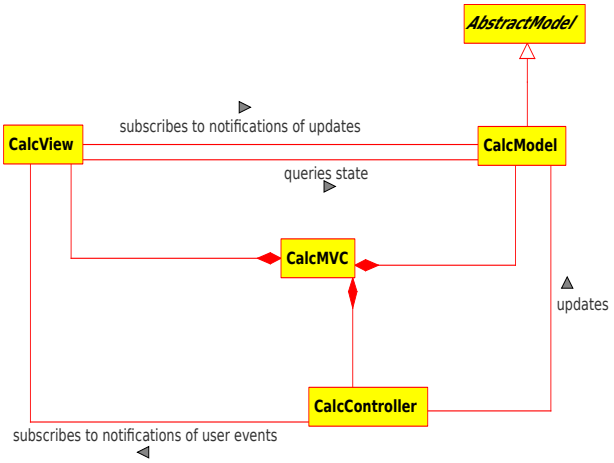
a) Compare the above class diagram with the MVC architecture as described in Unit 8 and by Bennett et al. (2010) and Eckstein (2007). You will see that it does not conform to the architecture.

In what ways does the given Java application *violate* the MVC architecture?

Model Answer:

Code as shown:





Correct MVC model:

- In MVC, to enable simultaneous use of multiple **`views`** (regardless of the views being the same or different), the **`model`** needs to have each of its dependent **`view`** components as subscribers, so that when the **`model`** changes, all dependent **`view`** components will be updated automatically. The **`model`** doesn't need to know anything about the interface or operation signatures of those **`views`** - it just needs to send them information about changes which will be used in whatever way is relevant for that particular view.
  CalcModel (i.e. the **`model`**) in the given Java application does not generate any event to which CalcView could subscribe. Hence, CalcView needs to rely on a *third party* (in this case, CalcController) to update its state.
- In MVC, while the **`view`** may query the **`model`** so as to update the data presented to the user, the **`view`** should not instruct the **`model`** to change directly. It should be the **`controller`**'s responsibility to map user action to model updates.
  CalcView not only instructs CalcModel to initialise its field (line 24), it also defines the named constant `INITIAL_VALUE` (line 10) and uses it to update the state of the CalcModel object. This probably was owing to an attempt to rectify the programming bug in CalcModel at line 11.
- In true MVC, update requests for the **`model`** should come from the **`controller`**, rather than from the **`view`**. CalcView, however, invokes a method in CalcModel (line 24) to change the state of the **`model`**.
- While a **`controller`** in MVC may select a different **`view`** to be presented to the user, it is not responsible for instructing a **`view`** to update itself (i.e. change its state).
  As CalcModel does not notify the dependent **`view`** components of any updates, CalcController needs to invoke methods in CalcView to update the **`view`** (cf. lines 36 & 51). This approach means that changes initiated by user inputs from one **`view`** cannot be propagated to other related views unless the application has only one **`controller`** and it registers with *all* views which access the *same* model.

3. **Further Challenges** `(Optional)`

Consider the definition of classes CalcMVC, CalcModel, CalcView and CalcController in Unit 8.

a) Produce your own class diagram, using a tool of your choice, based on the code.

b) How would the given class definitions need to change in order to enable the application to support *division* as well as multiplication?

c) Assume that *each* arithmetic operation provided by class CalcModel is to be presented in a *different* view.

Suggest modifications to the given Java application in order to address this change.

Note that the suggested modifications must conform to the MVC architecture.

Appendix

A Java *Calculator* (or "multiplier") application:

– Class CalcMVC

```
1  // CalcMVC.java - Calculator in MVC pattern.
2  // Fred Swartz - December 2004
3
4  import javax.swing.*;
5
6  public class CalcMVC {
7      //...  Create model, view, and controller.  They are
8      // created once here and passed to the parts that
9      // need them so there is only one copy of each.
10     public static void main(String[] args) {
11
12         CalcModel      model      = new CalcModel();
13         CalcView       view       = new CalcView(model);
14         CalcController controller = new CalcController(model, view);
15
16         view.setVisible(true);
17     }
18 }
```

– Class CalcModel

```java
// CalcModel.java
// Fred Swartz - December 2004
// Model
// This model is completely independent of the user interface.
// It could as easily be used by a command line or web interface.

import java.math.BigInteger;

public class CalcModel {
    //...  Constants
    private static final String INITIAL_VALUE = "0";

    //...  Member variable defining state of calculator.
    private BigInteger m_total;  // The total current value state.

    /** Constructor */
    CalcModel() {
        reset();
    }

    /** Reset to initial value. */
    public void reset() {
        m_total = new BigInteger(INITIAL_VALUE);
    }

    /** Multiply current total by a number.
     *@param operand Number (as string) to multiply total by.
     */
    public void multiplyBy(String operand) {
        m_total = m_total.multiply(new BigInteger(operand));
    }

    /** Set the total value.
     *@param value
     *  New value that should be used for the calculator total.
     */
    public void setValue(String value) {
        m_total = new BigInteger(value);
    }

    /** Return current calculator total. */
    public String getValue() {
        return m_total.toString();
    }
}
```

– Class CalcView

```
// CalcView.java - View component
// Presentation only.  No user actions.
// Fred Swartz - December 2004

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class CalcView extends JFrame {
    //...  Constants
    private static final String INITIAL_VALUE = "1";

    //...  Components
    private JTextField m_userInputTf = new JTextField(5);
    private JTextField m_totalTf     = new JTextField(20);
    private JButton    m_multiplyBtn = new JButton("Multiply");
    private JButton    m_clearBtn    = new JButton("Clear");

    private CalcModel m_model;

    /** Constructor */
    CalcView(CalcModel model) {
        //...  Set up the logic
        m_model = model;
        m_model.setValue(INITIAL_VALUE);

        //...  Initialize components
        m_totalTf.setText(m_model.getValue());
        m_totalTf.setEditable(false);

        //...  Layout the components.
        JPanel content = new JPanel();
        content.setLayout(new FlowLayout());
        content.add(new JLabel("Input"));
        content.add(m_userInputTf);
        content.add(m_multiplyBtn);
        content.add(new JLabel("Total"));
        content.add(m_totalTf);
        content.add(m_clearBtn);

        //...  finalize layout
        this.setContentPane(content);
        this.pack();

        this.setTitle("Simple Calc - MVC");
        // The window closing event should probably be passed to the
        // Controller in a real program, but this is a short example.
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    void reset() {
        m_totalTf.setText(INITIAL_VALUE);
    }

    String getUserInput() {
        return m_userInputTf.getText();
    }
```

```
57
58      void setTotal(String newTotal) {
59          m_totalTf.setText(newTotal);
60      }
61
62      void showError(String errMessage) {
63          JOptionPane.showMessageDialog(this, errMessage);
64      }
65
66      void addMultiplyListener(ActionListener mal) {
67          m_multiplyBtn.addActionListener(mal);
68      }
69
70      void addClearListener(ActionListener cal) {
71          m_clearBtn.addActionListener(cal);
72      }
73  }
```

– Class CalcController

```java
// CalcController.java - Controller
// Handles user interaction with listeners.
// Calls View and Model as needed.
// Fred Swartz - December 2004

import java.awt.event.*;
public class CalcController {
    //The Controller needs to interact with both the Model and View.
    private CalcModel m_model;
    private CalcView  m_view;

    /** Constructor */
    CalcController(CalcModel model, CalcView view) {
        m_model = model;
        m_view  = view;

        //...  Add listeners to the view.
        view.addMultiplyListener(new MultiplyListener());
        view.addClearListener(new ClearListener());
    }

    /////////////////////////// inner class MultiplyListener
    /** When a multiplication is requested.
     *  1. Get the user input number from the View.
     *  2. Call the model to multiple by this number.
     *  3. Get the result from the Model.
     *  4. Tell the View to display the result.
     * If there was an error, tell the View to display it.
     */
    class MultiplyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String userInput = "";
            try {
                userInput = m_view.getUserInput();
                m_model.multiplyBy(userInput);
                m_view.setTotal(m_model.getValue());

            } catch (NumberFormatException nfex) {
                m_view.showError("Bad input: '" + userInput + "'");
            }
        }
    }//end inner class MultiplyListener

    /////////////////////////// inner class ClearListener
    /**  1. Reset model.
     *   2. Reset View.
     */
    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            m_model.reset();
            m_view.reset();
        }
    }// end inner class ClearListener
}
```

References

Revell, A., *Trustworthy software*, [Online]. Available at: https://www.bcs.org/content-hub/trustworthy-software/ [14/10/2019].

Savvas, A., *Government launches PAS 754 'software trustworthiness' standard*, Computerworld, 16 June 2014. [Online]. Available at: https://www.computerworld.com/article/3420586/government-launches-pas-754--software-trustworth html [15/10/2022].

UK-TSI, *Trustworthy Software Guidance Document: Trustworthy Software Essentials (TSE)*, TS502-1, Traffic Light Protocol (TLP) White, Issue 1.2, February 2016. [Online]. Available at: http://tsfdn.org/wp-content/uploads/2016/03/TS502-1-TS-Essentials-Guidance-Issue-1.2-WHITE.pdf [15/10/2022].

Tutorial T09

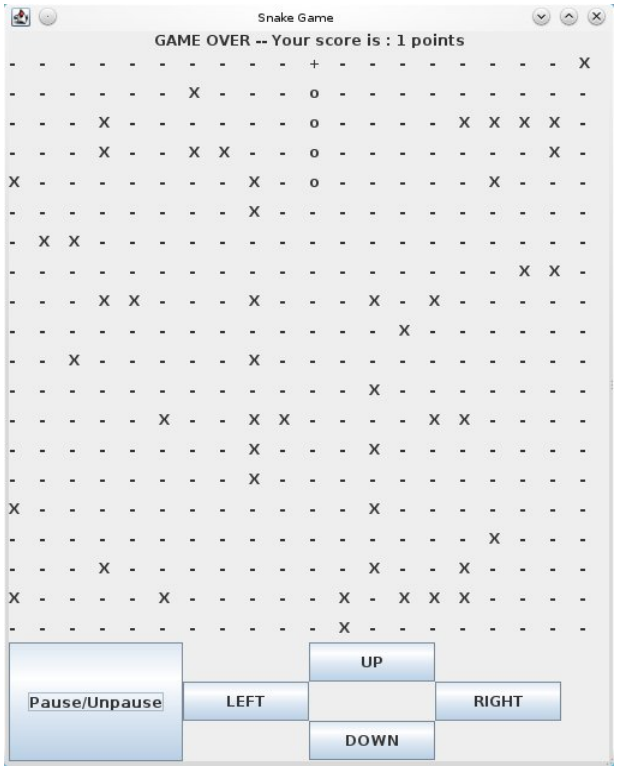
Theme

- describing a detailed class design using UML class notations
- evaluating a detailed class design in terms of cohesion and coupling
- improving a detailed class design to achieve low coupling and high cohesion

Key concepts: detailed class design, UML class notations, cohesion and coupling

Scenario

Consider the snake game used to be played on old computer terminals, which can only display a fixed rectangular grid of varying text characters. A simple re-creation of this game in Java is illustrated below.



In this implementation of the game, the snake body is "drawn" using circles (letters 'o') and its head is formed by a plus (letter '+'). Initially, the snake's head is at the centre of the grid and its body is three 'o's long, straight, extending towards the bottom of the grid. The crosses are food, which is randomly distributed in the grid. At any time there has to be the same amount of food (i.e. 70 pieces) across the grid.

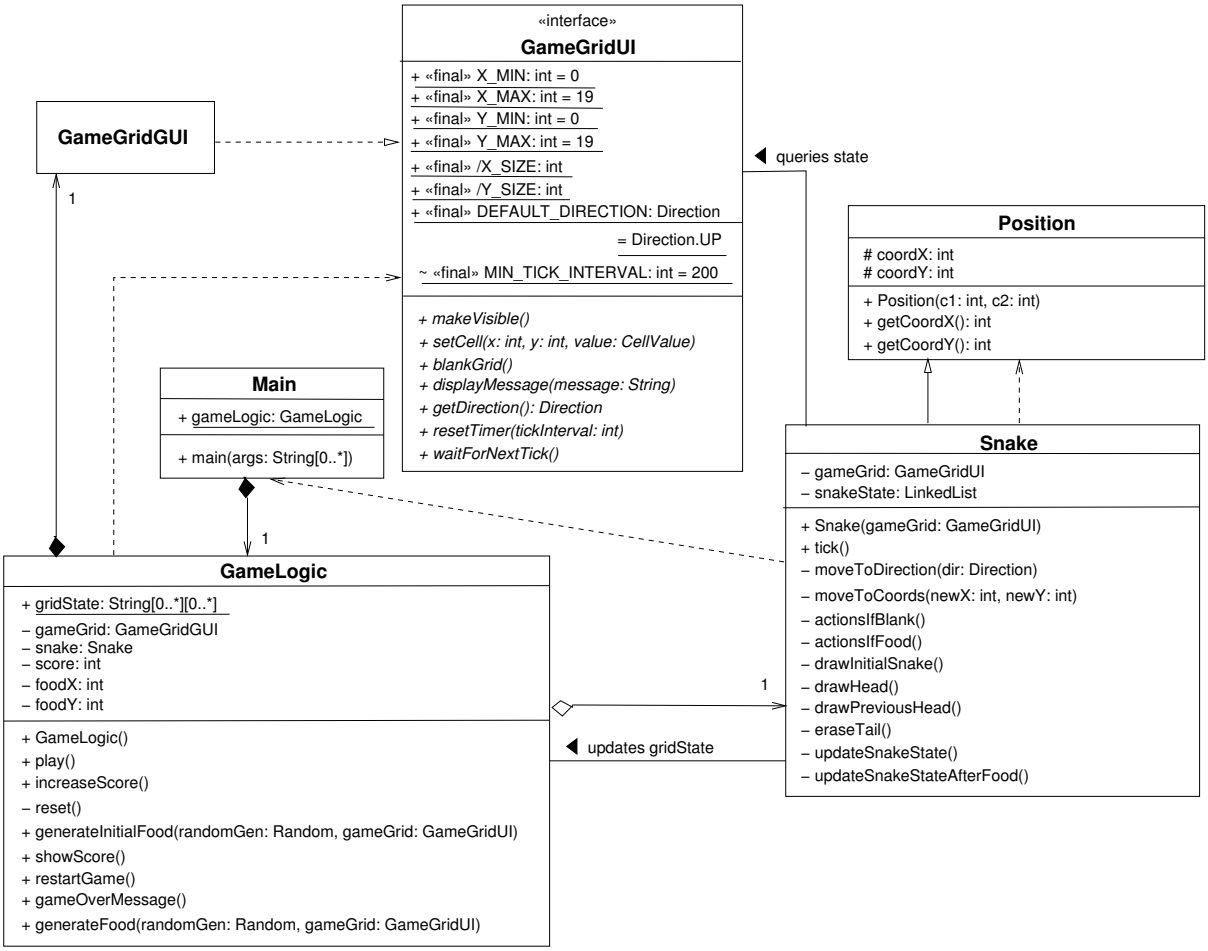
The snake keeps moving its head and pulling its body in a single track. When its head eats a piece of food, it does not pull its tail in the next step, thus growing its length by one unit. As soon as the snake eats a piece of food, another piece appears randomly at one of the positions that were blank at that moment so that the total amount of food remains the same. The direction in which the snake's head moves is determined by the player using the four buttons at the centre bottom of the window. The player earns one point for every piece of food eaten by the snake. The score display is continuously updated with every such event. When the snake attempts to leave the grid or it crashes into its own body, it dies immediately and the game is over. After some time (equivalent to the time needed for 10 moves of the snake), a new game starts automatically.

1. Object-Oriented Detailed Design

Consider the Java classes in the Appendix. These Java classes implement a Java application for the Snake Game as was described in the given scenario.

a) *Evaluating Object-Oriented Detailed Design*

The following UML class diagram is intended to show the detailed design of the given Snake Game application.



(i) Cross-check the given UML class diagram against the Java code to make sure that you understand the notation for the attributes and operations in each component and the relations between the components. There is an additional aggregation relationship which could be added to this diagram. Identify this relatonship and describe it.

Enumerated types have been omitted from this class diagram. NB: Note that the arrow on an aggregation or a composition relation (as shown here) is not mandatory. The arrow indicates the direction of navigation between the involved classes. Such information may be omitted from the aggregation notation.

Model Answer:
Class Snake has a LinkedList of Position objects representing the locations of the different parts of the snake's body. This could be represented with a composition link.

(ii) Evaluate the given detailed class design (as seen in the class diagram) in terms of the following types of coupling:

A. Interaction coupling between the three classes GameLogic, GameGridUI and Snake

Interaction Coupling is a measure of the number of message types an object sends to other objects and the number of parameters passed with these message types.

Model Answer:
Class GameLogic is tightly coupled with interface GameGridUI. There are 6 different types of messages passed from class GameLogic to interface GameGridUI, i.e. waitForNextTick, makeVisible, blankGrid, setCell, displayMessage and resetTimer.
Class Snake is tightly coupled with class GameLogic, with 4 different types of messages being passed from class Snake to class GameLogic, i.e. restartGame, setCell, generateFood and increaseScore. If class Snake were to be well-designed, there ought to be no need for it to pass any message to class GameLogic because class Snake, as a model, should be instructed to be updated by class GameLogic (which models the game rules), not the other way round.
Class GameLogic is, however, loosely coupled with class Snake as it only invokes method tick in class Snake.

B. Inheritance coupling.

Inheritance Coupling describes the degree to which a subclass actually needs the features it inherits from its base class.

Model Answer:
Poor.
Class Snake inherits features `coordX` and `coordY` from class Position. The inherited fields `coordX` and `coordY` are used to store the x and y coordinates of the snake's head.
The inherited features are unnecessary because class Snake keeps the location of each body part of the snake in the LinkedList `snakeState`, with its head stored as the first element of the linked list. Keeping the x and y coordinates of the snake's head in separate fields is redundant. This is also prone to data corruption, especially when the programmer forgets to update either copy of the data.

(iii) Evaluate the given detailed class design (as captured in your class diagram in part (i)) in terms of the following types of cohesion:

A. Operation cohesion of the methods actionsIfFood() and updateSnakeState() in class Snake

Operation cohesion measures the degree to which an operation focuses on a single functional requirement.

Model Answer:
Poor and good, respectively.
Method actionsIfFood() in class Snake performs a range of operations from updating the appearance of the snake on the game board, generating new food for the game board to update the game score. While these operations are what needed to take place when the snake had ingested some food, these operations in fact address different

functional requirements.
By contrast, method updateSnakeState() performs a small set of operations which are all related to redrawing the snake in its new position. The tasks naturally go together because they combine to define the new configuration of cells which will be rendered

B. Class cohesion in the class Snake.
Class cohesion measures the degree to which a class is focused on a single requirement.

Model Answer:
Poor.
Class Snake doesn't just include operations to maintain its state, it is also responsible for generating food for the game board, drawing the snake on the game board, and restarting the game.
Class Snake also records the object reference for the GameGridGUI object as one of its attributes, which is inappropriate.

C. Specialization cohesion
Specialization cohesion addresses the semantic cohesion of inheritance hierarchies. It measures the semantic-relatedness between the inherited attributes and operations to the class itself.

Model Answer:
Poor.
Class Snake inherits from class Position simply because it wants to keep the x and y coordinates of the snake's head in the game board. Semantically, a Snake object is not a special kind of Position. Position is merely a property of a snake in the game.

b) Does the given class design satisfy the *Liskov Substitution Principle*? Explain your answer.

Model Answer:
If we simply consider the properties and operations inherited by class Snake from class Position without considering their semantics, it appears to be Liskov-compliant because all members of a Position object may be "relevant" to a Snake object. Attributes `coordX` and `coordY` are used to keep track of the current position of a snake's head and operations getCoordX and getCoordY may be used to review the snake's current position (as partially defined by its head). There is no need for explicitly "disinherit" any properties.
However, under this interpretation, the position of the snake is said to be the position of the head of the snake, which is an over-simplification. More importantly, while a Snake object might occupy a position in the game board, itself is not a position in the game board. Hence, semantically, a Snake object cannot be considered as a specialised version of a position in the game board, which is what class Position is modelling.
Semantically Snake cannot be considered as a **`derived class`** of Position.

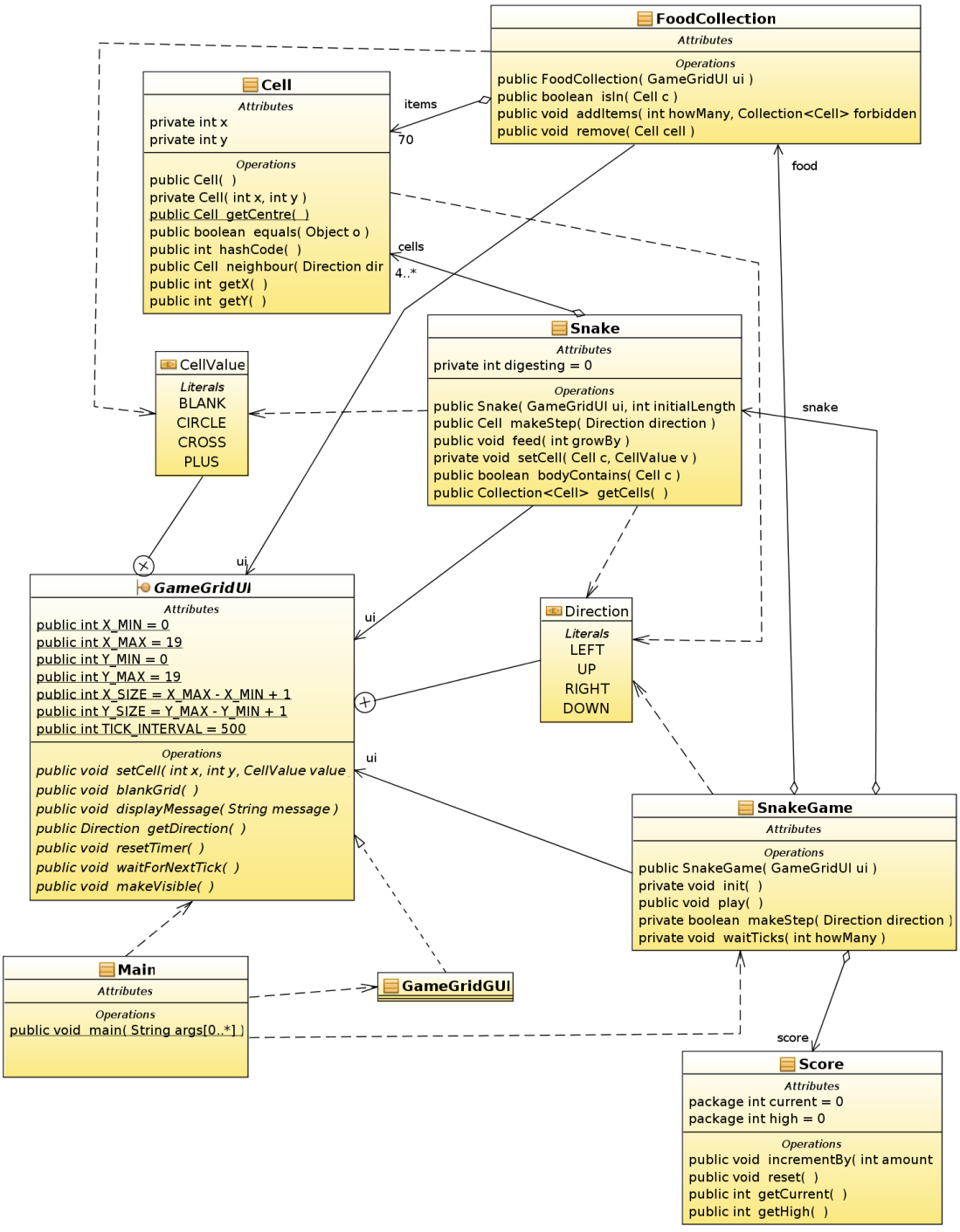
2. *Further Challenge* `(Optional)`

a) Suggest improvements to the detailed design of the given Snake Game Java application so as to:
   - reduce the level of coupling, and
   - increase the level of cohesion.

   You should focus on improving the detailed design as shown on the UML class diagram. There is no need to modify the given Java code to implement your proposed modifications.

Model Answer:

The following shows an improved detailed design of the snake game. It was generated using the UML plugin for the `NetBeans` IDE. The notation used in the diagram does not follow the UML specification closely. Its purpose is to show how the coupling can be reduced while making the classes more cohesive.



An improved Snake Game detailed design.

Appendix

Listing 1: Class Main

```java
package snake;
/** The Main class that contains the main method */
public class Main
{
    public static GameLogic gameLogic;

    /**
     * The main method that starts the game
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        gameLogic = new GameLogic();
        gameLogic.play();
    }
}
```

Listing 2: Class Position

```java
package snake;
/** Class for holding the x,y coordinates of a grid cell */
public class Position
{
    protected int coordX; // The x coordinate of the grid cell
    protected int coordY; // The y coordinate of the grid cell

    /**
     * Constructor: The grid cell location is given by 2 coordinates
     * @param c1 The x coordinate
     * @param c2 The y coordinate
     */
    public Position(int c1, int c2) {
        this.coordX = c1;
        this.coordY = c2;
    }

    /**
     * Getter method for the x coordinate
     * @return The coord1 instance variable
     */
    public int getCoordX() {
        return this.coordX;
    }

    /**
     * Getter method for the y coordinate
     * @return The coord2 instance variable
     */
    public int getCoordY() {
        return this.coordY;
    }
}
```

Listing 3: Class GameGridGUI

```
1  package snake;
2
3  // various import statements omitted
4
5  public class GameGridGUI extends javax.swing.JFrame
6                               implements GameGridUI
7  {
8    // implementation details omitted
9  }
```

Listing 4: Interface GameGridUI

```
1  package snake;
2
3  /**
4   * Template for a user interface featuring:
5   * - a message field
6   * - a 20x20 grid of cells each of which can be set to one of 4 values
7   * - input of a desired direction out of 4
8   * - a ticking timer
9   */
10 public interface GameGridUI
11 {
12     public static final int X_MIN = 0;
13     public static final int X_MAX = 19;
14     public static final int Y_MIN = 0;
15     public static final int Y_MAX = 19;
16     public static final int X_SIZE = X_MAX - X_MIN + 1;
17     public static final int Y_SIZE = Y_MAX - Y_MIN + 1;
18
19     /** Show the UI to the user */
20     void makeVisible();
21
22     /**
23      * Change the look of a certain given cell in the grid.
24      *
25      * @param x column number (0 to 19, 0 on the left)
26      * @param y row number (0 to 19, 0 at the bottom)
27      * @param value one of the predefined shapes to fill this cell
28      */
29     void setCell(int x, int y, CellValue value);
30
31     /** Various shapes that can be displayed in a cell of the grid. */
32     enum CellValue {
33         BLANK, CIRCLE, CROSS, PLUS
34     }
35
36     /** Make all cells in the grid Blank */
37     void blankGrid();
38
39     /**
40      * Display the given string as a message to the player.
41      *
42      * @param message for the user to read
43      */
44     void displayMessage(String message);
45
```

```
46      /**
47       * Return the direction that the player indicated most recently.
48       *
49       * @return the direction selected by the player
50       */
51      Direction getDirection();
52
53      /** A fixed set of directions the player can select from. */
54      enum Direction {
55          LEFT, UP, RIGHT, DOWN
56      }
57
58      /**
59       * The direction selected by resetTimer() until player starts
60       * interacting.
61       */
62      public static final Direction DEFAULT_DIRECTION = Direction.UP;
63
64      /**
65       * Start ticking now with the given interval.  If the interval
66       * indicated by tickInterval is smaller than
67       * MIN_TICK_INTERVAL, MIN_TICK_INTERVAL will be used instead.
68       *
69       * @param tickInterval
70       *  The desired interval between consecutive ticks in milliseconds
71       */
72      void resetTimer(int tickInterval);
73
74      /**
75       * The minimum time between two consecutive timer ticks in
76       * milliseconds.
77       */
78      static int MIN_TICK_INTERVAL = 200;
79
80      /**
81       * Every time this method is called, it waits until certain time.
82       * If the time has already passed, the method returns immediately.
83       * The first time it is called after resetTimer(), it will return
84       * no sooner than TICK_INTERVAL after the call to resetTimer().
85       * The second time it will wait until 2 * TICK_INTERVAL after the
86       * most recent call to resetTimer(), etc.
87       */
88      void waitForNextTick();
89  }
```

Listing 5: Class GameLogic

```
1  package snake;
2  import java.util.Random;
3   /**
4    * A class that holds all the info for a game session. It also
5    * creates a 2D array that holds the state of every cell
6    * on the grid.
7    */
8  public class GameLogic
9  {
10     // A 2D array that holds the state of every cell on the grid
11     public static String[][] gridState =
12                     new String[GameGridUI.X_SIZE][GameGridUI.Y_SIZE];
13
14     private GameGridGUI gameGrid;
15
16     private Snake snake;
17     private int score; // the player's score
18     private int foodX; // x coordinate of a piece of food
19     private int foodY; // y coordinate of a piece of food
20
21     /**
22      * constructor: it also creates an instance of the GUI
23      */
24     public GameLogic() {
25         gameGrid = new GameGridGUI();
26     }
27
28     /**
29      * The method that kick-starts the game. It resets the board, and
30      * then arranges the timing of every movement of the snake. It
31      * also shows the score on the top of the grid.
32      */
33     public void play() {
34         reset();  // reset everything
35
36         // a loop to hold the iteration for every movement
37         while (true)
38         {
39             // wait for the next movement
40             gameGrid.waitForNextTick();
41             // moves the snake
42             snake.tick();
43             // shows the score
44             showScore();
45         }
46     }
47
48     /** Increases the score by 1 */
49     public void increaseScore() {
50         score++;
51     }
52
53     /**
54      * Resets the game board, the score, the snake and the pieces of
55      * food
56      */
57     private void reset() {
```

```
58          // make the game grid visible
59          gameGrid.makeVisible();
60
61          /* populate the gridState array representation with String
62           * objects that correspond to BLANK ("-") */
63          for (int iy = GameGridUI.Y_SIZE - 1; iy >= 0; iy--)
64          {
65              for (int ix = 0; ix < GameGridUI.X_SIZE; ix++)
66              {
67                  String string = new String();
68                  string = "-";
69                  gridState[ix][iy] = string;
70              }
71          }
72
73          // set all grid icons to BLANK
74          gameGrid.blankGrid();
75          // reset the score
76          score = 0;
77          // set up a new snake
78          snake = new Snake(gameGrid);
79
80          // generate a random
81          Random randomGen = new Random();
82          // generate the initial amount of food and place it on the grid
83          generateInitialFood(randomGen, gameGrid);
84      }
85
86      /**
87       * Generates the initial amount of food and places it on the grid
88       * after checking that the cell is not occupied
89       * @param randomGen
90       *   a Random class parameter, used to randomize the food
91       * @param gameGrid
92       *   the game grid on which the food will be placed
93       */
94      public void generateInitialFood(Random randomGen,
95                                      GameGridUI gameGrid)
96      {
97          int i = 0;   // set up a counter
98          for (i = 0; i <= 54; i++)  // creates 55 pieces of food
99          {
100             // assign x coordinate of the food
101             foodX = GameGridUI.X_MIN +
102                     randomGen.nextInt(GameGridUI.X_SIZE);
103
104             // assign y coordinate of the food
105             foodY = GameGridUI.Y_MIN +
106                     randomGen.nextInt(GameGridUI.Y_SIZE);
107
108             // while (foodX,foodY) lies on the body of the snake
109             while ((foodX == 10 & foodY == 10)
110                     || (foodX == 10 & foodY == 9)
111                     || (foodX == 10 & foodY == 8)
112                     || (foodX == 10 & foodY == 7))
113             {
114                 // assign x coordinate of the food
115                 foodX = GameGridUI.X_MIN +
```

```
116                         randomGen.nextInt(GameGridUI.X_SIZE);
117                 // assign y coordinate of the food
118                 foodY = GameGridUI.Y_MIN +
119                         randomGen.nextInt(GameGridUI.Y_SIZE);
120             }
121
122             /* place the food on the grid */
123             GameLogic.gridState[foodX][foodY] = "X";
124
125             // place the icons of the food on the game grid
126             gameGrid.setCell(foodX, foodY, GameGridUI.CellValue.CROSS);
127         }
128     }
129
130     /** Show the current score */
131     public void showScore()
132     {
133         // construct a String with an expression for the current score
134         String msg = String.format("Your score: %d points", score);
135         // show the String on the GUI
136         gameGrid.displayMessage(msg);
137     }
138
139     /** Restart the game */
140     public void restartGame()
141     {
142         // show a message that the game is over
143         gameOverMessage();
144
145         // restart the game after 5 seconds
146         try {
147             Thread.sleep(5000); // in milliseconds
148         }
149         catch (InterruptedException e) // catch the exception
150         {
151             gameGrid.displayMessage(e.getMessage());
152         }
153         gameGrid.resetTimer(200); // reset timer
154         play(); //start the game again
155     }
156
157     /** Inform the player that the game is over. */
158     public void gameOverMessage()
159     {
160         //construct a String with a message confirming that game is over
161         String msg =
162               String.format("GAME OVER -- Your score is : %d points",
163                                     score);
164         //show the message on the GUI
165         gameGrid.displayMessage(msg);
166     }
167
168     /**
169      * Generate a piece of food and place it on the grid
170      * @param randomGen a Random
171      * @param gameGrid the grid UI representation
172      */
173     public void generateFood(Random randomGen, GameGridUI gameGrid)
```

```
174     {
175         // assign x coordinate of the food
176         foodX = GameGridUI.X_MIN +
177                 randomGen.nextInt(GameGridUI.X_SIZE);
178         // assign y coordinate of the food
179         foodY = GameGridUI.Y_MIN +
180                 randomGen.nextInt(GameGridUI.Y_SIZE);
181 
182         // check if the new food location is blank
183         if (GameLogic.gridState[foodX][foodY].equals("-"))
184         {
185             // place it on the grid
186             GameLogic.gridState[foodX][foodY] = "X";
187             // and put its icon on the grid
188             gameGrid.setCell(foodX, foodY, GameGridUI.CellValue.CROSS);
189         }
190         else // generate a new piece of food
191         {
192             generateFood(randomGen, gameGrid);
193         }
194     }
195 }
```

Listing 6: Class Snake

```
1  package snake;
2  
3  import snake.GameGridUI.Direction;
4  import java.util.LinkedList;
5  import java.util.Random;
6  
7  /** The Snake Class represents a snake */
8  public class Snake extends Position
9  {
10     private GameGridUI gameGrid;  // A variable of type GameGridUI
11 
12     // holds instances of Position of every part of the snake
13     private LinkedList<Position> snakeState =
14                                         new LinkedList<Position>();
15 
16     /**
17      * constructor: creates a new Snake instance, puts it on the game
18      * grid with its head on the middle cell and then it draws it.
19      * @param gameGrid
20      */
21     public Snake(GameGridUI gameGrid)
22     {
23         super(10,10);  // set the coordinates for the head of the snake
24 
25         /* associates it with the gameGrid variable that represents
26          * the grid on the GUI */
27         this.gameGrid = gameGrid;
28 
29         drawInitialSnake();  // draws the initial snake
30     }
31 
32     /**
33      * gets the direction and passes the argument to
```

```
34      *  moveToDirection() method
35       */
36      public void tick()
37      {
38          Direction dir = gameGrid.getDirection();
39          moveToDirection(dir);
40      }
41
42      /**
43       * calculates the next grid cell that the head will go. If it is
44       * outside the grid limits, it calls for game over.
45       * @param dir
46       *      the direction as an enumeration type of DOWN, UP,
47       *      LEFT and RIGHT
48       */
49      private void moveToDirection(Direction dir)
50      {
51          switch (dir)
52          {
53              case DOWN: // if DOWN
54              {
55                  //if it is inside the minimum y value
56                  if (coordY > GameGridUI.Y_MIN) {
57                      moveToCoords(coordX, coordY - 1); // move
58                  }
59                  else {
60                      Main.gameLogic.restartGame(); // restart game
61                  }
62                  break;
63              }
64              case UP: // if UP
65              {
66                  // if inside the maximum y
67                  if (coordY < GameGridUI.Y_MAX)
68                  {
69                      //move
70                      moveToCoords(coordX, coordY + 1);
71                  }
72                  else {
73                      // else restart game
74                      Main.gameLogic.restartGame();
75                  }
76                  break;
77              }
78              case LEFT: // if LEFT
79              {
80                  //if inside the minimum x
81                  if (coordX > GameGridUI.X_MIN) {
82                      //move
83                      moveToCoords(coordX - 1, coordY);
84                  }
85                  else {
86                      //else restart game
87                      Main.gameLogic.restartGame();
88                  }
89                  break;
90              }
91              case RIGHT: // if right
```

```
92                {
93                    // if inside the maximum x
94                    if (coordX < GameGridUI.X_MAX) {
95                        // move
96                        moveToCoords(coordX + 1, coordY);
97                    }
98                    else {
99                        // else restart game
100                       Main.gameLogic.restartGame();
101                   }
102                   break;
103               }
104           }
105       }
106
107       /* The actual implementation of the movement
108        * @param newX the x coordinate of the target grid cell
109        * @param newY the y coordinate of the target grid cell
110        */
111       private void moveToCoords(int newX, int newY)
112       {
113           // the new coordinates become the current ones
114           coordX = newX;
115           coordY = newY;
116
117           // if the new grid cell is blank
118           if (GameLogic.gridState[newX][newY].equals("-")) {
119               actionsIfBlank();
120           }
121           // the new coordinate contains food
122           else if (GameLogic.gridState[newX][newY].equals("X")) {
123               actionsIfFood();
124           }
125           // the new coordinates contain part of the snake so it is game
    over
126           else {
127               Main.gameLogic.restartGame();
128           }
129       }
130
131       /* Contains all the necessary actions to draw the snake and
132        * update the grid cells if the next cell is empty
133        */
134       private void actionsIfBlank()
135       {
136           drawPreviousHead(); // draw the previous head as CIRCLE
137           eraseTail(); // erase the tail
138           drawHead(); // draw the new head
139           // update the representation of the snake in the snakeState LinkedList
140           updateSnakeState();
141       }
142
143       /* Contains all the necessary actions to draw the snake and update
144        * the grid cells if the next cell contains food.
145        */
146       private void actionsIfFood()
147       {
148           drawPreviousHead(); // draw the previous head as CIRCLE
```

```
149         drawHead();          // draw the new head
150         // update the representation of the snake in the snakeState LinkedList
151         updateSnakeStateAfterFood();
152         // generate a random
153         Random randomGen = new Random();
154         // use it to generate a piece of food on a random cell
155         Main.gameLogic.generateFood(randomGen, gameGrid);
156         // increase the score by 1
157         Main.gameLogic.increaseScore();
158     }
159
160     /* Draws the initial snake at the start of the game.
161      * The snake consists of a head and three body parts.
162      */
163     private void drawInitialSnake()
164     {
165         /* change the cell info on the gridState representation
166          *  of the head
167          */
168         GameLogic.gridState[coordX][coordY] = "+";
169
170         /* change the cell icon on the gameGrid representation
171          * of the head
172          */
173         gameGrid.setCell(coordX, coordY, GameGridUI.CellValue.PLUS);
174
175         /* change the cell info on the gridState representation
176          * of the body
177          */
178         GameLogic.gridState[coordX][coordY - 1] = "o";
179         GameLogic.gridState[coordX][coordY - 2] = "o";
180         GameLogic.gridState[coordX][coordY - 3] = "o";
181
182         // change the cell icon on the gameGrid representation of the body
183         gameGrid.setCell(coordX, coordY - 1, GameGridUI.CellValue.CIRCLE);
184         gameGrid.setCell(coordX, coordY - 2, GameGridUI.CellValue.CIRCLE);
185         gameGrid.setCell(coordX, coordY - 3, GameGridUI.CellValue.CIRCLE);
186
187         /* assign the coordX.coordY coordinates of every part of the snake
188          * to Position instances */
189         Position head = new Position(coordX, coordY);
190         Position body1 = new Position(coordX, coordY - 1);
191         Position body2 = new Position(coordX, coordY - 2);
192         Position tail = new Position(coordX, coordY - 3);
193
194         // add these Position instances to the snakeState LinkedList
195         snakeState.addFirst(head);
196         snakeState.addLast(bodcoordY1);
197         snakeState.addLast(bodcoordY2);
198         snakeState.addLast(tail);
199     }
200
201     /* Draw the actual head on the grid */
202     private void drawHead()
203     {
204         // change the cell on the gridState representation
205         GameLogic.gridState[coordX][coordY] = "+";
206
```

```
207         // change the cell icon on the gameGrid representation
208         gameGrid.setCell(coordX, coordY, GameGridUI.CellValue.PLUS);
209     }
210 
211     /* Draws a CIRCLE on the cell occupied by the previous head. */
212     private void drawPreviousHead()
213     {
214         /* get the Position of the previous head from the first element
215          * of the snakeState LinkedList */
216         Position previousHead = (Position) snakeState.getFirst();
217 
218         //get the coordX and the coordY variables from the Position instances
219         int previousHeadX = previousHead.getCoordX();
220         int previousHeadY = previousHead.getCoordY();
221 
222         //change the cell on the gridState representation
223         GameLogic.gridState[previousHeadX][previousHeadY] = "o";
224 
225         //change the cell icon on the gameGrid representation
226         gameGrid.setCell(previousHeadX, previousHeadY,
227                          GameGridUI.CellValue.CIRCLE);
228     }
229 
230     /* Erases the Tail from the grid and the logical representation
231      * in gridState LinkedList.
232      */
233     private void eraseTail()
234     {
235         /* get the Position of the previous tail from the last element
236          * of the snakeState LinkedList */
237         Position tail = (Position) snakeState.getLast();
238 
239         // get the coordX and the coordY variables from the Position instances
240         int tailX = tail.getCoordX();
241         int tailY = tail.getCoordY();
242 
243         // change the cell on the gridState representation
244         GameLogic.gridState[tailX][tailY] = "-";
245 
246         // change the cell icon on the gameGrid representation
247         gameGrid.setCell(tailX, tailY, GameGridUI.CellValue.BLANK);
248     }
249 
250     /* Updates the remaining info on the gridState representation by
251      * removing the previous tail Position and adding the new head Position
252      */
253     private void updateSnakeState()
254     {
255         snakeState.removeLast();  // remove the last element
256         // put the coordinates of the new head in a Position instance
257         Position newHead = new Position(coordX, coordY);
258         // add the new head Position as the first element
259         snakeState.addFirst(newHead);
260     }
261 
262     /* Updates the remaining info on the gridState representation in
263      * the case of eating the food by adding the new head Position
264      */
```

```
265     private void updateSnakeStateAfterFood()
266     {
267         // put the coordinates of the new head in a Position instance
268         Position newHead = new Position(coordX, coordY);
269 
270         // add the new head Position as the first element
271         snakeState.addFirst(newHead);
272     }
273 }
```

Tutorial T10

Theme

– Identify and apply design pattern(s) in software design

---

1. Drawing application

Consider the Java classes Canvas, Circle, Square and Triangle in the Appendix. These Java classes model a partial Java application for drawing shapes on a canvas.

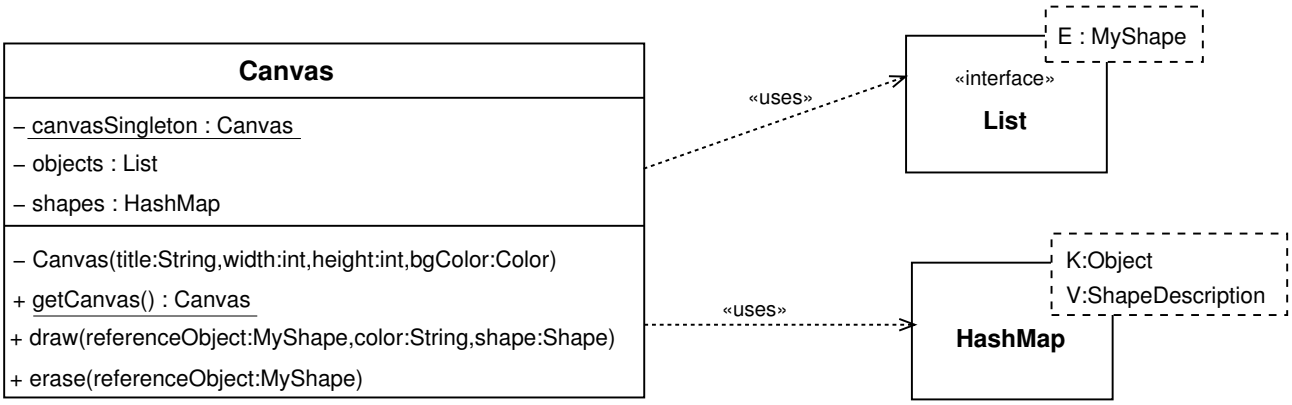
a) State the name of the object-oriented design pattern that has been applied in the design of class Canvas.

Model Answer:
Singleton

b) With reference to the code in the Appendix, explain how the design pattern identified in part (a) affects the design of class Canvas.

Model Answer:
Class Canvas:

– has a private class variable (i.e. `canvasSingleton`) which holds a Canvas object;
– has a *private* constructor;
– has a public class method (i.e. getCanvas) which returns a Canvas object.



c) Explain why there was a need to apply the design pattern that you identified in part (a) to this application.

Model Answer:
Class Canvas was implemented using the *singleton* design pattern so as to enable various Circle, Square and Triangle objects to be created and be drawn on the *same* canvas.

Instances of Circle, Square and Triangle simply obtain a Canvas object and invoke its methods so to make themselves visible in, and invisible from, the canvas.

2. A more advanced drawing application

Assume that class Picasso has been introduced to the given partial Java application in the appendix. Class Picasso models the behaviour of a painter who makes paintings using various geometric shapes such as triangle, circles, etc. A Picasso object is responsible for manipulating drawings on a canvas and it includes the following four operations:

- to make a drawing appear on the canvas
- to erase a drawing from the canvas
- to shift a drawing horizontally across the canvas by a specified distance
- to shift a drawing vertically across the canvas by a specified distance

A Picasso object can make a simple drawing (i.e. a circle, a square or a triangle) on the canvas. It can also make a complex drawing such as a tree (i.e. a green circle on top of a black square) or a scene made up of a forest (i.e. multiple trees) and a sun (i.e. a yellow circle). The same set of operations apply regardless of the complexity of the drawing.

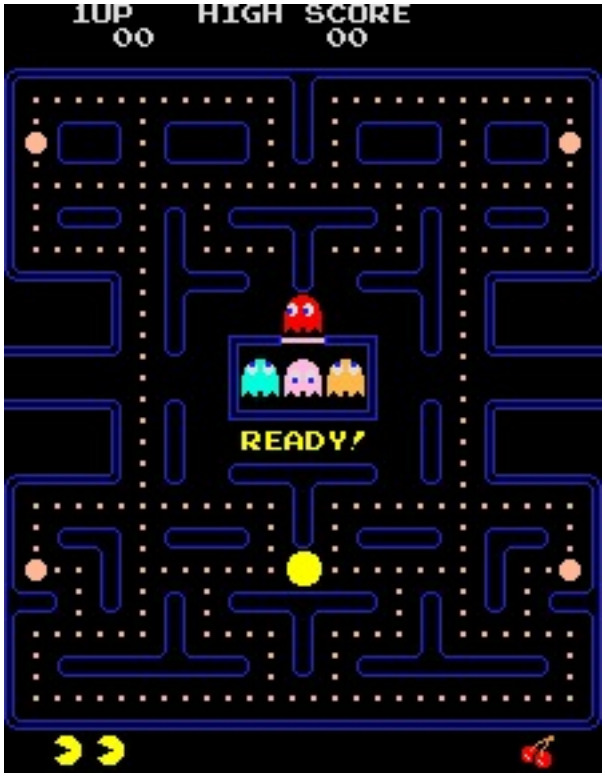
a) State the name of the object-oriented design pattern that would be appropriate to be applied in the design of the classes with which class Picasso is expected to collaborate.

Model Answer:

Composite. (If you want to explore this in more detail, there is an optional challenge question at the end.)

3. A game example

Consider the following partial description of the classic computer game Pac-Man:



The player controls Pac-Man through a maze, eating pac-dots. When all pac-dots are eaten, Pac-Man is taken to the next stage. Regardless of the game stage, the layout of the maze remains the same.

Near the corners of the maze are four larger, flashing dots known as power pellets. Eating a power pellet will enable Pac-Man to enter a temporary power boost.

At each stage, four ghosts (Blinky, Pinky, Inky and Clyde) roam the maze, trying to catch Pac-Man. Each ghost has a distinct personality. Blinky always chases Pac-Man. Pinky always aims for positioning itself in front of Pac-Man. Inky is "fickle" and sometimes heads towards Pac-Man and other times away. Clyde chases Pac-Man when it is at some distance from him, and it moves towards the lower-left corner of the maze when it gets very close to Pac-Man.

If Pac-Man is touched by a ghost, he loses a life. When Pac-Man loses all of his lives, the game ends.

Consider a hypothetical Java implementation of the computer game Pac-Man which includes the classes Maze, Location and Ghost. Class Maze models the maze in which Pac-Man and the ghosts roam. A Maze object contains a collection of Location objects for keeping track of what is in each location in the maze. Class Ghost models the behaviours of the ghosts as described above.

a) *Which* object-oriented design pattern would be applicable to the class Maze, so as to ensure that Pac-Man and the four ghosts will roam within the

same maze? *Justify* your answer.

Model Answer:

The Singleton pattern

Regardless of the game level a player is at, a Pac-Man game has one maze layout only. The same Maze object can therefore be used throughout the game. Having a single Maze object in the game also helps to ensure that Pac-Man and the ghosts will roam in the same maze.

b) Which object-oriented design pattern would be good to model the family of roaming algorithms used by the ghosts so that each Ghost object can take on a different roaming behaviour during its creation?

Model Answer:

the Strategy pattern. (If you want to explore this in more detail, there is an optional challenge question at the end.)

4. OPTIONAL CHALLENGE EXERCISES

a) Making use of the design pattern which you identified in part (2a), make a detailed class design for the partial Java application described in section 2.

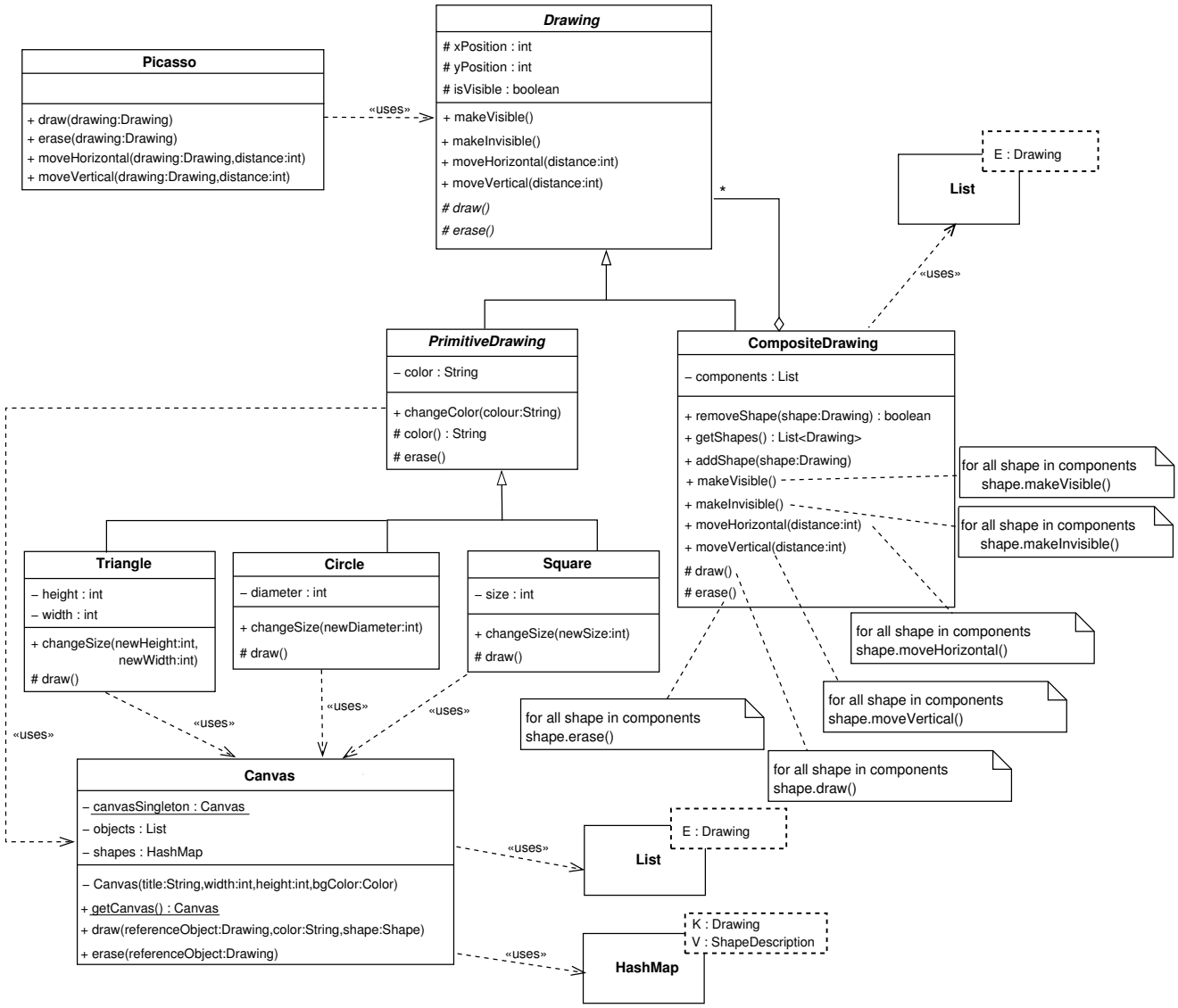
Draw a detailed UML class diagram to illustrate your detailed class design. For those operations that are not included in the given Java classes, use the UML comment notation to outline how these operations should be implemented.

Hint:

Your detailed UML class diagram should include classes Picasso, Circle, Square, Triangle and Canvas. You may suppress the details of class Canvas which are not used by classes Circle, Square and Triangle.

Model Answer:

Example diagram.

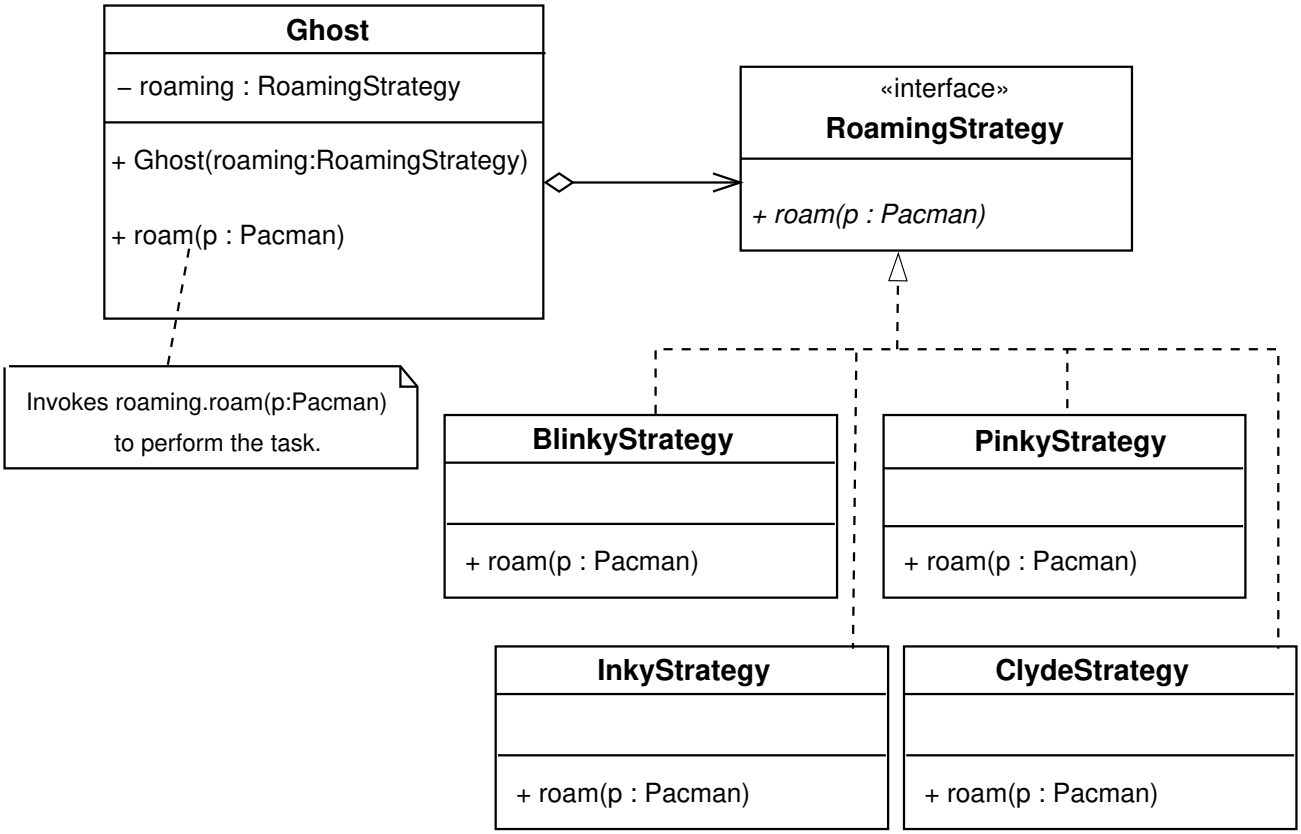


b) *Draw* a detailed UML class diagram to show how the design pattern you identified in question 3b can be applied to class Ghost, so that each Ghost object may be flexibly set up with their specific roaming behaviour.

Your class diagram should include the use of the UML comment notation to outline how the operation(s) that are typical to the identified design pattern would be implemented.

Your class diagram may omit details that are irrelevant to the application of the identified design pattern.

Model Answer:


Ghost
– roaming : RoamingStrategy
+ Ghost(roaming:RoamingStrategy)
+ roam(p : Pacman)
«interface»
RoamingStrategy
+ roam(p : Pacman)
Invokes roaming.roam(p:Pacman)
to perform the task.
BlinkyStrategy
+ roam(p : Pacman)
PinkyStrategy
+ roam(p : Pacman)
InkyStrategy
+ roam(p : Pacman)
ClydeStrategy
+ roam(p : Pacman)

5. Appendix

Listing 1: Class Canvas

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.util.List;
4  import java.util.*;
5  /**
6   * Canvas is a class to allow for simple graphical drawing on a canvas.
7   * This is a modification of the general purpose Canvas, specially made
8   * for the BlueJ "shapes" example.
9   * @author: Bruce Quig
10  * @author: Michael Kolling (mik)
11  * @version: 1.6 (shapes)
12  *
13  * @author: Sylvia Wong (modified by)
14  * @version: 16-02-2011
15  */
16 public class Canvas
17 {
18     private static Canvas canvasSingleton;
19
20     /** Factory method to get the canvas singleton object. */
21     public static Canvas getCanvas() {
22         if(canvasSingleton == null) {
23             canvasSingleton = new Canvas("BlueJ Shapes Demo",
24                                          300, 300,  Color.white);
25         }
26         canvasSingleton.setVisible(true);
27         return canvasSingleton;
28     }
29
30     // --- instance part ---
31
32     private JFrame frame;
33     private CanvasPane canvas;
34     private Graphics2D graphic;
35     private Color backgroundColour;
36     private Image canvasImage;
37     private List<Object> objects;
38     private HashMap<Object,ShapeDescription> shapes;
39
40     /**
41      * Create a Canvas.
42      * @param title  title to appear in Canvas Frame
43      * @param width  the desired width for the canvas
44      * @param height  the desired height for the canvas
45      * @param bgClour  the desired background colour of the canvas
46      */
47     private Canvas(String title, int width, int height, Color bgColour) {
48         frame = new JFrame();
49         canvas = new CanvasPane();
50         frame.setContentPane(canvas);
51         frame.setTitle(title);
52         canvas.setPreferredSize(new Dimension(width, height));
53         backgroundColour = bgColour;
54         frame.pack();
55         objects = new ArrayList<Object>();
```

```
56          shapes = new HashMap<Object,ShapeDescription>();
57      }
58
59      /**
60       * Set the canvas visibility and brings canvas to the front of screen
61       * when made visible. This method can also be used to bring an already
62       * visible canvas to the front of other windows.
63       * @param visible  boolean value representing the desired visibility of
64       * the canvas (true or false)
65       */
66      public void setVisible(boolean visible) {
67          if(graphic == null) {
68              // first time:  instantiate the offscreen image and fill it
69              // with the background colour
70              Dimension size = canvas.getSize();
71              canvasImage = canvas.createImage(size.width, size.height);
72              graphic = (Graphics2D)canvasImage.getGraphics();
73              graphic.setColor(backgroundColour);
74              graphic.fillRect(0, 0, size.width, size.height);
75              graphic.setColor(Color.black);
76          }
77          frame.setVisible(visible);
78      }
79
80      /**
81       * Draw a given shape onto the canvas.
82       * @param  referenceObject  an object to define identity for this shape
83       * @param  color            the color of the shape
84       * @param  shape            the shape object to be drawn on the canvas
85       */
86      public void draw(Object referenceObject, String color, Shape shape) {
87          objects.remove(referenceObject); // in case it was already there
88          objects.add(referenceObject);    // add at the end
89          shapes.put(referenceObject, new ShapeDescription(shape, color));
90          redraw();
91      }
92
93      /**
94       * Erase a given shape's from the screen.
95       * @param  referenceObject  the shape object to be erased
96       */
97      public void erase(Object referenceObject) {
98          objects.remove(referenceObject); // in case it was already there
99          shapes.remove(referenceObject);
100         redraw();
101     }
102
103     /**
104      * Set the foreground colour of the Canvas.
105      * @param  newColour   the new colour for the foreground of the Canvas
106      */
107     public void setForegroundColor(String colorString) {
108         // implementation detail omitted.
109     }
110
111     /**
112      * Wait for a specified number of milliseconds before finishing.
113      * This provides an easy way to specify a small delay which can be
```

```
114         * used when producing animations.
115         * @param  milliseconds  the number
116         */
117        public void wait(int milliseconds) {
118            // implementation detail omitted.
119        }
120
121        /** Redraw all shapes currently on the Canvas. */
122        private void redraw() {
123            erase();
124            for(Object obj : objects) {
125                ((ShapeDescription)shapes.get(obj)).draw(graphic);
126            }
127            canvas.repaint();
128        }
129
130        /** Erase the whole canvas. (Does not repaint.) */
131        private void erase() {
132            Color original = graphic.getColor();
133            graphic.setColor(backgroundColour);
134            Dimension size = canvas.getSize();
135            graphic.fill(new Rectangle(0, 0, size.width, size.height));
136            graphic.setColor(original);
137        }
138
139        /**********************************************************************
140         * Inner class CanvasPane - the actual canvas component contained in the
141         * Canvas frame. This is essentially a JPanel with added capability to
142         * refresh the image drawn on it.
143         */
144        private class CanvasPane extends JPanel {
145            public void paint(Graphics g) {
146                g.drawImage(canvasImage, 0, 0, null);
147            }
148        }
149
150        /**********************************************************************
151         * Inner class ShapeDescription
152         */
153        private class ShapeDescription {
154            private Shape shape;
155            private String colorString;
156
157            public ShapeDescription(Shape shape, String color) {
158                this.shape = shape;
159                colorString = color;
160            }
161
162            public void draw(Graphics2D graphic) {
163                setForegroundColor(colorString);
164                graphic.fill(shape);
165            }
166        }
167    }
```

Listing 2: Class Circle

```
import java.awt.geom.*;
/**
 * A circle that can be manipulated and that draws itself on a canvas.
 * @author    Michael Kolling and David J. Barnes
 * @author S H S Wong (modified by)
 * @version 18-02-2011
 */
public class Circle {
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new circle at a specific position with a specific color.
     * @param diameter of the circle
     * @param x     x position
     * @param y     y position
     * @param colour        colour of the diameter
     */
    public Circle(int diameter, int x, int y, String colour) {
        this.diameter = diameter;
        xPosition = x;
        yPosition = y;
        color = colour;
        isVisible = false;
    }

    /**
     * Make this circle visible. If it was already visible, do nothing.
     */
    public void makeVisible() {
        isVisible = true;
        draw();
    }

    /**
     * Make this circle invisible. If it was already invisible, do nothing.
     */
    public void makeInvisible() {
        erase();
        isVisible = false;
    }

    /** Move the circle horizontally by 'distance' pixels. */
    public void moveHorizontal(int distance) {
        erase();
        xPosition += distance;
        draw();
    }

    /** Move the circle vertically by 'distance' pixels. */
    public void moveVertical(int distance) {
        erase();
        yPosition += distance;
        draw();
```

```
58      }
59
60      /**
61       * Change the size to the new size (in pixels). Size must be >= 0.
62       */
63      public void changeSize(int newDiameter) {
64          erase();
65          diameter = newDiameter;
66          draw();
67      }
68
69      /**
70       * Change the color. Valid colors are "red", "yellow", "blue", "green"
71       *  and "black".
72       */
73      public void changeColor(String newColor) {
74          color = newColor;
75          draw();
76      }
77
78      /* Draw the circle with current specifications on screen. */
79      private void draw() {
80          if(isVisible) {
81              Canvas canvas = Canvas.getCanvas();
82              canvas.draw(this, color,
83                          new Ellipse2D.Double(xPosition, yPosition,
84                                               diameter, diameter));
85              canvas.wait(10);
86          }
87      }
88
89      /* Erase the circle on screen. */
90      private void erase() {
91          if(isVisible) {
92              Canvas canvas = Canvas.getCanvas();
93              canvas.erase(this);
94          }
95      }
96  }
```

Listing 3: Class Square

```
1  import java.awt.*;
2  /**
3   * A square that can be manipulated and that draws itself on a canvas.
4   * @author    Michael Kolling and David J. Barnes
5   * @author S H S Wong (modified by)
6   * @version 18-02-2011
7   */
8  public class Square {
9      private int size;
10     private int xPosition;
11     private int yPosition;
12     private String color;
13     private boolean isVisible;
14
15     /**
16      * Create a new square at a required position with the required color.
17      * @param size  size of the square
18      * @param x     x position
19      * @param y     y position
20      * @param colour        colour of the square
21      */
22     public Square(int size, int x, int y, String colour) {
23                 this.size = size;
24                 xPosition = x;
25                 yPosition = y;
26                 color = colour;
27                 isVisible = false;
28     }
29
30     /**
31      * Make this square visible. If it was already visible, do nothing.
32      */
33     public void makeVisible() {
34         isVisible = true;
35         draw();
36     }
37
38     /**
39      * Make this square invisible. If it was already invisible, do nothing.
40      */
41     public void makeInvisible() {
42         erase();
43         isVisible = false;
44     }
45
46     /** Move the square horizontally by 'distance' pixels. */
47     public void moveHorizontal(int distance) {
48         erase();
49         xPosition += distance;
50         draw();
51     }
52
53     /** Move the square vertically by 'distance' pixels. */
54     public void moveVertical(int distance) {
55         erase();
56         yPosition += distance;
57         draw();
```

```
58      }
59
60      /**
61       * Change the size to the new size (in pixels). Size must be >= 0.
62       */
63      public void changeSize(int newSize) {
64          erase();
65          size = newSize;
66          draw();
67      }
68
69      /**
70       * Change the color. Valid colors are "red", "yellow",
71       *   "blue", "green" and "black".
72       */
73      public void changeColor(String newColor) {
74          color = newColor;
75          draw();
76      }
77
78      /* Draw the square with current specifications on screen. */
79      private void draw() {
80          if(isVisible) {
81              Canvas canvas = Canvas.getCanvas();
82              canvas.draw(this, color,
83                          new Rectangle(xPosition, yPosition, size, size));
84              canvas.wait(10);
85          }
86      }
87
88      /* Erase the square on screen. */
89      private void erase() {
90          if(isVisible) {
91              Canvas canvas = Canvas.getCanvas();
92              canvas.erase(this);
93          }
94      }
95  }
```

Listing 4: Class Triangle

```
1  import java.awt.*;
2  /**
3   * A triangle that can be manipulated and that draws itself on a canvas.
4   *
5   * @author    Michael Kolling and David J. Barnes
6   * @author S H S Wong (modified by)
7   * @version 18-02-2011
8   */
9  public class Triangle {
10     private int height;
11     private int width;
12     private int xPosition;
13     private int yPosition;
14     private String color;
15     private boolean isVisible;
16
17     /** Create a new triangle. */
18     public Triangle(int height, int width, int x, int y, String colour) {
19         this.height = height;
20         this.width = width;
21         this.xPosition = x;
22         this.yPosition = y;
23         color = colour;
24         isVisible = false;
25     }
26
27     /**
28      * Make this triangle visible. If it was already visible, do nothing.
29      */
30     public void makeVisible() {
31         isVisible = true;
32         draw();
33     }
34
35     /**
36      * Make this triangle invisible. If it was already invisible, do nothing.
37      */
38     public void makeInvisible() {
39         erase();
40         isVisible = false;
41     }
42
43     /** Move the triangle horizontally by 'distance' pixels. */
44     public void moveHorizontal(int distance) {
45         erase();
46         xPosition += distance;
47         draw();
48     }
49
50     /** Move the triangle vertically by 'distance' pixels. */
51     public void moveVertical(int distance) {
52         erase();
53         yPosition += distance;
54         draw();
55     }
56
57     /**
```

```
58          * Change the size to the new size (in pixels). Size must be >= 0.
59          */
60         public void changeSize(int newHeight, int newWidth) {
61             erase();
62             height = newHeight;
63             width = newWidth;
64             draw();
65         }
66 
67         /**
68          * Change the color. Valid colors are "red", "yellow",
69          * "blue", "green" and "black".
70          */
71         public void changeColor(String newColor) {
72             color = newColor;
73             draw();
74         }
75 
76         /* Draw the triangle with current specifications on screen. */
77         private void draw() {
78             if(isVisible) {
79                 Canvas canvas = Canvas.getCanvas();
80                 int[] xpoints = { xPosition, xPosition + (width/2),
81                                   xPosition - (width/2) };
82                 int[] ypoints = { yPosition, yPosition + height,
83                                   yPosition + height };
84                 canvas.draw(this, color, new Polygon(xpoints, ypoints, 3));
85                 canvas.wait(10);
86             }
87         }
88 
89         /* Erase the triangle on screen. */
90         private void erase() {
91             if(isVisible) {
92                 Canvas canvas = Canvas.getCanvas();
93                 canvas.erase(this);
94             }
95         }
96 }
```

Tutorial T11

Theme

- writing Java code which conforms to the Java coding conventions
- writing preconditions, post-conditions and invariants
- documenting and verifying design assumptions in Java code using assertions

Key concepts: *Java coding conventions, class invariants, pre/post-conditions, assertions in Java*

T11.1. Writing readable Java code

a) Class `phoneaccount` in Appendix A is very difficult to read because it fails to conform to the Java coding conventions.

- State ***five*** types of issues with the readability of the given class `phoneaccount`.

Model Answer:

(i) Missing appropriate comments for the program. The only documentation comment in the program does not describe the purpose of the class.
(ii) The class name `phoneaccount`should be written as `PhoneAccount`
(iii) The local variable `MIN` does not follow the Java naming convention, i.e.: start with lower case, than mixed case.
(iv) Poor indentation and alignment throughout.
(v) Variable names (e.g. `s`, `tx`, `e` and `c`) are not sufficiently meaningful.
(vi) The method `main` contains too many lines of code.

b) Class `ScrabbleSets` in Appendix B was a solution submitted to the *Java Code Challenge: Scrabble Sets* at DZone (dzone.com). Details of the challenge is available at:

https://dzone.com/articles/java-code-challenge-scrabble-sets

The code is functional and meets the requirements set out in the code challenge.

State ***three*** issues with the **readability** and **maintainability** of class `ScrabbleSets` in Appendix B.

Model Answer:

(i) The logic of method printRemainingTiles is not intuitive due to the use of an

HashMap<Character, Integer> object to model the the fre- quency of each letter tile.

(ii) Excessive use of class scope variable(s) and method(s), making the solution a predominantly procedural program rather than following the object-oriented approach.

(iii) Missing appropriate comments for the program.

(iv) Though the named constant COUNT_BY_LETTERfollows the Java naming convention for named constants, the role it plays in the program is not easy to understand, hence making the code more difficult to understand and to maintain.

(v) Poor indentation and alignment throughout.

T11.2. Precondition & Assertion

Consider this definition of class `NumberTester`. (**ignore the vertical line at left**)

```
1   /**
2    * Test a number between 0-99 to see if it is a prime number.
3    * @author S H S Wong
4    * @version 2011-03-16
5    */
6   public class NumberTester {
7
8       /* All prime numbers between 0 and 99. */
9       private static final int[] PRIMES = {
10            2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
11            43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
11
13      /* Determine if the given number is a prime. */
14      private static boolean isPrime(int number) {
15          // Checks the given number against the list of primes.
16          for (int aPrime : PRIMES) {
17              if (aPrime == number) {
18                  return true;
19              }
20          }
21          return false;
22      }
23
24      /**
25       * Main: run the number tester
26       * @param args
27       */
28      public static void main(String args[])
29      {
30          Scanner input = new Scanner(System.in);
31          System.out.print("Enter a number between 0 and 99 (inclusive): ");
32          int number = input.nextInt();
33
34          String result = "";
35          if (isPrime(number)) {
36              result = " ";
37          }
38          else {
39              result = " not ";
40          }
41          System.out.printf("%d is%sa prime number.\n", number, result);
42      }
43  }
```

Class `NumberTester` has been written with an assumption made in the design of this class. To ensure the program operates correctly, a *precondition* is expected to be applied somewhere within this class.

a) Where should a precondition be introduced?

Model Answer:

A precondition should be introduced to method `isPrime`.

b) Write out the required precondition in plain English.

Model Answer:
The value given parameter (ie number) must be in the range of 0–99 (inclusive).

c) Write an `assert` statement to introduce the required precondition check. Where should this `assert` statement be inserted?

Model Answer:
The assertstatement should be inserted into method `isPrime`:

```
1     /* Determine if the given number is a prime. */
2     private static boolean isPrime(int number) {
3         assert (number >= 0 && number <= 99) :   // precondition
4                 "Number out of range (0-99): " + number;
5
6         // Checks the given number against the list of primes.
7         for(int aPrime : PRIMES) {
8             if (aPrime == number) {
9                 return true;
10            }
11        }
11        return false;
13    }
```

T11.3. More on Assertion

Consider the Java classes `Budget`, `BudgetItem` in Appendix B. These Java classes model a partial Java application for managing a student's weekly spending budget.

Class `Budget` keeps track of a list of budget items. A client can:

- add a budget item (a budget item is expected to be in a known category: utility, rent, food, transport, entertainment)
- record the spending of an existing budget item

Class `BudgetItem` models the detail of each budget item including category of the budget, budget amount and actual expenditure. The budget amount is set when a `BudgetItem` object is created. A client can view the detail and record an expenditure.

a) State what is meant by a:

(i) class invariant

Model Answer:
What must be true about each instance of a class.

(ii) precondition

Model Answer:
What must be true when a method is invoked.

(iii) post-condition

Model Answer:
What must be true after a method completes successfully.

b) The (documentation and block) comments in classes `BudgetItem` and `Budget` clearly state the **class invariants**, **preconditions** and **post-conditions** for these classes whenever appropriate.

Write assert statements for classes `BudgetItem` and `Budget` to verify these design assumptions.

Model Answer:

Listing 1: Class BudgetItem

```
1  public class BudgetItem {
2
3      /* Class Invariants:
4       * category != null
5       * amount >= 0
6       * spending >= 0
7       */
8      private Category category;  // budget category
9      private int amount;         // budgeted amount (in pence)
10     private int spending;       // actual spending (in pence)
11
11     /** Constructor */
13     public BudgetItem(Category category, int budget) {
14         this.category = category;
15         this.amount = budget;
16         spending = 0;
17         assert category != null : "Budget category is " + category;
18         assert budget >= 0 : "Negative budget amount: " + budget;
19         assert spending >= 0 : "Negative spending amount: "
20                                                 + spending;
21     }
22
23     // other methods omitted
24
25     /** Increase the actual spending of this budget item
26      *  by the specified amount.
27      *  Precondition: amount > 0
28      *  Post-condition: spending' = spending + amount
29      */
30     public void recordSpending(int amount) {
31         assert amount > 0 : "Non-positive spending amount: "
32                                                 + amount;
33         int oldSpending = spending;
34         spending = amount;
35         assert spending == (oldSpending + amount) :
36             "Wrong calculation of new spending. Old value: "
37             + oldSpending +
38             " Given amount: " + amount + " New value: " + spending;
39         assert (category != null && budget >= 0 && spending >= 0) :
40             "Inappropriate object state: \n Budget category is "
41             + category +
42             "\n Budget amount is negative: " + budget +
43             "\n Spending amount is negative: " + spending;
```

```
44        }
45    }
```

Listing 2: Class Budget

```
1  import java.util.HashMap;
2  import java.util.Map;
3  /**
4    * Class Budget models a budget.
5    * New budget item can be added to the budget.
6    * Expenditure for each budget item can be recorded within the budget.
7    * @author S H S Wong 8
     * @version 2018-11-22
9    */
10 public class Budget {
11
11       private Map<Category, BudgetItem> budgetDetail;
13
14       // other class detail omitted
15
16       /** Add budget item in a specified category.
17        * preconditions:
18        *      The specified category must be one of the known categories.
19        *      The budget amount must be >= 0.
20        * post-condition:
21        *  The number of budget items must have been increased by 1.
22        */
23       public void addBudgetItem(String category, int amount) {
24           assert amount >= 0 : "negative budget amount: " + amount;
25           int numOfBudgetItems = budgetDetail.size();
26           Category budgetCategory = null;
27           try {
28             budgetCategory = Category.valueOf(category);
29           }
30           catch (IllegalArgumentException e) {
31               assert false :
32                   "Unknown budget item category: " + category;
33               throw e;
34           }
35           catch (NullPointerException npe) {
36               assert false :
37                   "Unknown budget item category: " + category;
38               throw npe;
39           }
40           budgetDetail.put(budgetCategory,
41                    new BudgetItem(budgetCategory, amount));
42           assert budgetDetail.size() == (numOfBudgetItems + 1) :
43               "An existing budget item has been overridden: "
44               + category;
45       }
46
47       /** Record a new spending amount.
48        * preconditions:
49        *      The specified category must be one of the known categories.
50        *      The budget amount must be >= 0.
51        */
52       public void recordSpending(String category, int amount) {
53           assert amount > 0 : "Unacceptable spending amount: "
```

```
54                  + amount;
55          Category budgetCategory = null;
56          try {
57            budgetCategory = Category.valueOf(category);
58          }
59          catch (IllegalArgumentException iae) {
60              assert false :
61                  "Unknown budget item category: " + category;
62              throw iae;
63          }
64          catch (NullPointerException npe) {
65              assert false :
66                  "Unknown budget item category: " + category;
67              throw npe;
68          }
69          if (budgetCategory != null) {
70              assert false :
71                  "Unknown budget item category: " + category;
72          }
73          BudgetItem item = budgetDetail.get(budgetCategory);
74          item.recordSpending(amount);
75      }
76  }
```

c) How can assertion checks can be enabled when running the application?

Model Answer:

Use the JVM argument -ea

T11.4. Appendix A

Listing 3: Class `phoneaccount`

```
/** This program receives data from another system. */
import java.util.Scanner;
 public class phoneaccount
 {
public static void main(String[] args)
{
 final double s = 0.08; // calls cost in pence
 final double tx = 0.05; //texts cost each in pence
 int e;
 int c = 0;
 Scanner in = new Scanner(System.in);
 System.out.print("Enter your phone number: ");
 String phoneNumber = in.nextLine();
 System.out.println("Your remaining credit for " + phoneNumber + " : " + c);
 System.out.print("Enter a whole number to top-up: ");
 int tu = in.nextInt();
 double firstCredit = c + tu;
 System.out.printf("Remaining credit for %s is %.2f\n", phoneNumber, firstCredit);
 if(firstCredit > s && firstCredit > tx) {
 System.out.println("You can now make a call. ");
 System.out.print("A call is in progress. Enter the length of calls in minutes: ");
 int MIN = in.nextInt();
 System.out.println("A text has been sent. ");
 System.out.print("Enter the number of texts sent: ");
 int textSent = in.nextInt();
 double callCost = s * MIN;
 double textCost = tx * textSent;
 firstCredit = firstCredit - callCost - textCost;
 System.out.printf("Remaining credit for %s is %.2f\n", phoneNumber, firstCredit);
 }
 if(firstCredit < s || firstCredit < tx) {
 System.out.println("You don't have enough credit.");}
 System.out.print("Enter a whole number to top-up: ");
 int newTopup = in.nextInt();
 double secondCredit = firstCredit + newTopup;
 System.out.printf("Remaining credit for %s is %.2f\n", phoneNumber, secondCredit);
 if  (secondCredit > s && secondCredit > tx) {
 System.out.println("You can now make a call. A text has been sent. ");
 System.out.print("Enter the length of calls in minutes: ");
 int newMin = in.nextInt();
 System.out.print("Enter the number of texts sent: ");
 int newTextSent = in.nextInt();
 double newCallCost = s * newMin;
 double newTextCost = tx * newTextSent;
 secondCredit = secondCredit - newCallCost - newTextCost;
 System.out.printf("Remaining credit for %s is %.2f\n", phoneNumber, secondCredit);
    }
    if (secondCredit < s || secondCredit < tx) {
    System.out.println("You don't have enough credit.");
    }
    else {System.out.println("Thank you");}
    }
 }
```

T11.5. Appendix B

Listing 4: Class ScrabbleSets

```
1  package software.schwering.javacodechallenge;
2
3  import java.util.HashMap;
4  import java.util.List;
5  import java.util.Map;
6  import java.util.Map.Entry;
7  import java.util.stream.Collectors;
8
9  /* Source: https://dzone.com/articles/java-code-solution-scrabble-sets */
10
11 public class ScrabbleSets {
11
13 private static final Map<Character, Integer> COUNT_BY_LETTER = new HashMap<>();
14
15 static{
16
17 //see http://scrabblewizard.com/scrabble-tile-distribution/
18
19  COUNT_BY_LETTER.put('A', 9);
20  COUNT_BY_LETTER.put('B', 2);
21  COUNT_BY_LETTER.put('C', 2);
22  COUNT_BY_LETTER.put('D', 4);
23 COUNT_BY_LETTER.put('E', 11);
24  COUNT_BY_LETTER.put('F', 2);
25  COUNT_BY_LETTER.put('G', 3);
26  COUNT_BY_LETTER.put('H', 2);
27  COUNT_BY_LETTER.put('I', 9);
28  COUNT_BY_LETTER.put('J', 1);
29  COUNT_BY_LETTER.put('K', 1);
30  COUNT_BY_LETTER.put('L', 4);
31  COUNT_BY_LETTER.put('M', 2);
32  COUNT_BY_LETTER.put('N', 6);
33  COUNT_BY_LETTER.put('O', 8);
34  COUNT_BY_LETTER.put('P', 2);
35  COUNT_BY_LETTER.put('Q', 1);
36  COUNT_BY_LETTER.put('R', 6);
37  COUNT_BY_LETTER.put('S', 4);
38  COUNT_BY_LETTER.put('T', 6);
39  COUNT_BY_LETTER.put('U', 4);
40  COUNT_BY_LETTER.put('V', 2);
41  COUNT_BY_LETTER.put('W', 2);
42  COUNT_BY_LETTER.put('X', 1);
43  COUNT_BY_LETTER.put('Y', 2);
44  COUNT_BY_LETTER.put('Z', 1);
45  COUNT_BY_LETTER.put('_', 2);
46 }
47
48 public static void printRemainingTiles(String tilesInPlay){
49 Map<Character, Integer> remainingCount = new HashMap<>(COUNT_BY_LETTER);
50 tilesInPlay.chars().forEach(i->remainingCount.put((char)i, remainingCount.get((char)i)-1));
51 List<Character> errors = remainingCount.entrySet().stream().filter(e->e.getValue()<0).map(Entry::ge
52 if(errors.isEmpty()){
53 Map<Integer, String> tilesByCount = remainingCount.entrySet().stream().collect(Collectors.groupingBy(E
54  tilesByCount.entrySet().stream().sorted((e1,e2)->e2.getKey().compareTo(e1.getKey())).forEachOrdered(
55 }else{
56 errors.forEach(c->System.out.printf("Invalid input. More %s's have been taken from the bag than possib
57  }
58  }
59  }
```

For full detail of class `ScrabbleSets`, see the Java file in the given Java code ZIP archive for this tutorial.

T11.6. Appendix C

Listing 5: Class BudgetItem

```
/** * Class BudgetItem models an item in a budget.  * Each budget item
must be in one of the predefined categories.        * @author S H S
Wong * @version 2011-03-16 */
 public class BudgetItem {
    /* Class Invariants:
     * category != null
     * amount >= 0
     * spending >= 0
     */
    private Category category;  // budget category
    private int amount;         // budgeted amount (in pence)
    private int spending;        // actual spending (in pence)

    /** Constructor */
    public BudgetItem (Category category, int budget) {
        this.category = category;
        this.amount = budget;
        spending = 0;
    }

    /** Return the category of this budgeted item */
    public Category getCategory() {
        return category;
    }

    /** Return the budgeted amount */
    public int getAmount() {
        return amount;
    }

    /** Return the actual spending of this budget item */
    public int getSpending() {
        return spending;
    }

    /** Increase the actual spending of this budget item

     *  by the specified amount.
     *  Precondition: amount > 0
     *  Post-condition: spending' = spending + amount
     */
    public void recordSpending(int amount) {
        spending = amount;
    }
}

/** Budget item categories */
enum Category {
    UTILITY, RENT, FOOD, TRANSPORT, ENTERTAINMENT
}
```

Listing 6: Class Budget

```
1  import java.util.HashMap;
2  import java.util.Map;
3  /**
4   * Class Budget models a budget.
5   * New budget item can be added to the budget.
6   * Expenditure for each budget item can be recorded within the budget.
7   * @author  S H S Wong
8   * @version 2011-03-16
9   */
10 public class Budget {
11
11     private Map<Category, BudgetItem> budgetDetail;
13
14     /** Constructor */
15     public Budget() {
16         this.budgetDetail = new HashMap<Category, BudgetItem>();
17     }
18
19     /** Add budget item in a specified category.
20      * preconditions:
21      *     The specified category must be one of the known categories.
22      *     The budget amount must be >= 0.
23      * post-condition:
24      *  The number of budget items must have been increased by 1.
25      */
26     public void addBudgetItem(String category, int amount) {
27         Category budgetCategory = Category.valueOf(category);
28         budgetDetail.put(budgetCategory,
29                          new BudgetItem(budgetCategory, amount));
30     }
31
32     /** Record a new spending amount.
33      * preconditions:
34      *     The specified category must be one of the known categories.
35      *     The budget amount must be >= 0.
36      */
37     public void recordSpending(String category, int amount) {
38         Category budgetCategory = Category.valueOf(category);
39         BudgetItem item = budgetDetail.get(budgetCategory);
40         item.recordSpending(amount);
41     }
42 }
```

Tutorial T12

Theme

- apply refactoring to source code

Key concepts: Various kinds of refactoring, e.g. big refactorings, extract superclass, push down, pull up, renaming, etc

1. Applying Refactoring to a Class

Consider the definition of class BorrowRecord in the Appendix. This class has been developed at an earlier sprint of an agile software development process.

Class BorrowRecord is designed to support the following user requirement of a local library:

> A member can borrow a book for a specified number of days.

BorrowRecord models a book borrowing record from a member of the library. When a book is borrowed by a member, a new BorrowRecord object is created to record the borrowing details. The start date of the borrowing period is set to the moment when the new BorrowRecord object is created. After the specified number of days, the borrowing period expires.

A borrow record can be printed to the standard output stream (i.e. the console). A borrow record can also be exported as an XML file.

The focus of the next sprint is to manage the library's stock. Before the next sprint is to commence, you are tasked with refactoring the given class BorrowRecord.

a) Name four types of refactoring that can be applied to class BorrowRecord so as to improve the readability, maintainability and extendability of class BorrowRecord.

Model Answer:

- renaming
- promoting attribute to class
- extract class
- separate domain from presentation

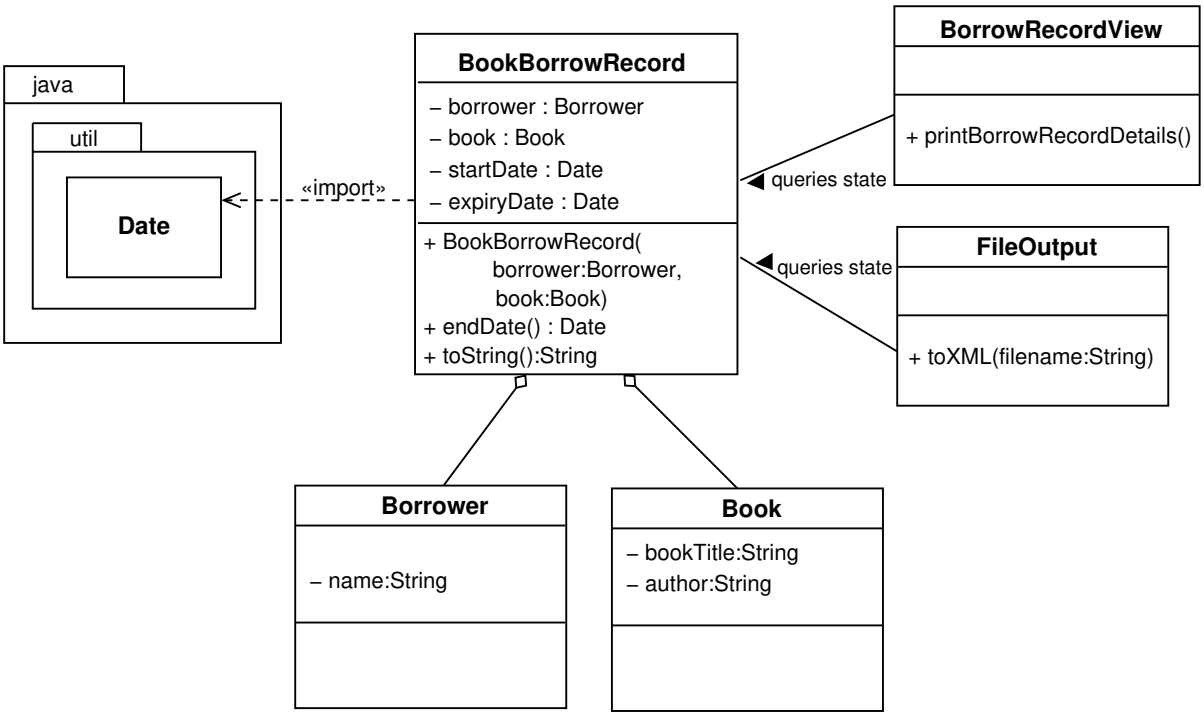
b) Making use of an UML class diagram, describe the design of class BorrowRecord after the four types of refactoring named in part (a) have been applied to this class.

Model Answer:

- Renaming:
  - The class name BorrowRecord is unclear. Rename BorrowRecord to BookBorrowRecord may help clarify the purpose of this class.

  - Rename `book` to `bookTitle`
  - Rename `endDate` to `expiryDate`
- Promoting attribute customer to class Borrower
  - move field `borrower` to the new class Borrower
  - rename `borrower` to `borrowerName` or simply `name`
- Extract class:
  - make a new class Book from fields `book` and `author`;
  - move all relevant getter and setter methods for these fields to the new class.
- Separate domain from presentation:
  - Move method printBorrowRecordDetails() to a BorrowRecordView class. The View class may also contain methods for presenting other model data from the underlying application.
  - Add a getter method for each field in class BorrowRecord.
  - Move method toXML(String) to a FileOutput class. This class may also contain methods for writing other model data in XML.

  Note that some implementation details on how the two view/output classes obtain data from class BorrowRecord have been omitted from the class diagram. There are different ways to implement the association relation between class BorrowRecord and its view/output classes. One way is to include a BorrowRecord field in classes BorrowRecordView and FileOutput. Another way is to apply the *observer* pattern. However, it is arguable whether the *observer* pattern is suitable for class FileOutput since its task (as suggested by method toXML(String)) is simply to write the data of a BorrowRecord object as XML to a file.



2. Refactoring using Eclipse (Optional)

Consider the Java classes Canvas, Circle, Square and Triangle in the Appendix. These Java classes model a partial Java application for drawing shapes on a canvas, and were featured in a previous tutorial.

a) Making use of Eclipse, apply the refactoring *Extract Superclass* to the given Java code.

Model Answer:

Extract Superclass "extracts a common superclass from a set of sibling types. The selected sibling types become direct subclasses of the extracted superclass after applying the refactoring."

Classes Circle, Square and Triangle have a few fields and methods in common, e.g. `xPosition`, `yPosition`, `isVisible`, `color`, erase(), makeVisible(), makeInvisible(), moveVertical(int), moveHorizontal(), etc. A new superclass Shape can be extracted from these classes by moving the common methods it.

As many of the common methods invoke draw(), this method needs to be declared in class Shape as abstract as each sibling class has a different implemenation of draw().

b) Can you think of other type(s) of refactoring that can be applied to the given code so as to improve its readability, maintainability and extendability?

3. Appendix

Listing 1: Class BorrowRecord

```
1  import java.io.FileNotFoundException;
2  import java.io.PrintWriter;
3  import java.io.UnsupportedEncodingException;
4  import java.util.Calendar;
5  import java.util.Date;
6
7  /**
8   * Class BorrowRecord models a library user's borrow record.
9   *
10  * The start date of the borrow record is set to the date
11  * when this BorrowRecord object is created. The end date
12  * of the borrowing is the date when the return date of
13  * the borrowed item.
14  *
15  * @author S H S Wong
16  * @version 06-12-2019
17  */
18 public class BorrowRecord {
19    private String borrower;
20    private String book;
21    private String author;   // The author of the book
22    private Date startDate;  // The date when the book was issued.
23
24    // The end date for the borrowing period, i.e.  the due date.
25    private Date endDate;
26
27    /**
28     * Constructor: creates BorrowRecord object
29     * @param borrower
30     * @param book
31     * @param author
32     */
33    public BorrowRecord(String borrower, String book, String author) {
34       this.borrower = borrower;
35       this.book = book;
36       this.author = author;
37
38       Calendar cal = Calendar.getInstance();
39       this.startDate = cal.getTime();
40
41       // Adds the duration of the borrowing to the current date.
42       cal.add(Calendar.DAY_OF_MONTH, 28);
43       this.endDate = cal.getTime();
44    }
45
46    /**
47     * Returns the end date of the borrowing period.
48     */
49    public Date endDate() {
50       return endDate;
51    }
52
53    /* (non-Javadoc)
54     * @see java.lang.Object#toString()
55     */
```

```
56      @Override
57      public String toString() {
58          return "Borrow Record:\n"
59              + (borrower != null ? "Borrower: " + borrower + "\n" : "")
60              + (book != null ? "Book: " + book + "\n" : "")
61              + (author != null ? "Author: " + author + "\n" : "")
62              + (startDate != null ? "Start Date: " + startDate + "\n" : "")
63              + (endDate != null ? "Return Date: " + endDate + "\n" : "");
64      }
65
66      /**
67       * Print the details of this book borrowing record to
68        * the standard output stream.
69       */
70      public void printBorrowDetails() {
71          System.out.println(this.toString());
72      }
73
74      /**
75       * @return this book borrowing record in XML.
76       * @throws FileNotFoundException, UnsupportedEncodingException
77       */
78      public void toXML(String filename)
79              throws FileNotFoundException, UnsupportedEncodingException {
80          PrintWriter writer = new PrintWriter(filename, "UTF-8");
81          String xml =
82          "<?xml version='1.0' encoding='UTF-8'?>\n"
83          + "<BorrowRecord>\n"
84          + (borrower != null ? "<borrower>" + borrower + "</borrower>\n" : "")
85          + (book != null ? "<book>" + book + "</book>\n" : "")
86          + (author != null ? "<author>" + author + "</author>\n" : "")
87          + (startDate != null ? "<startDate>" + startDate + "</startDate>\n" : "")
88          + (endDate != null ? "<endDate>" + endDate + "</endDate>\n" : "") + ""
89          + "</BorrowRecord>";
90          writer.println(xml);
91          if (writer != null) writer.close();
92      }
93  }
```

Listing 2: Class Canvas

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.util.List;
4  import java.util.*;
5  /**
6   * Canvas is a class to allow for simple graphical drawing on a canvas.
7   * This is a modification of the general purpose Canvas, specially made
8   * for the BlueJ "shapes" example.
9   * @author: Bruce Quig
10  * @author: Michael Kolling (mik)
11  * @version: 1.6 (shapes)
12  *
13  * @author: Sylvia Wong (modified by)
14  * @version: 16-02-2011
15  */
16 public class Canvas
17 {
18     private static Canvas canvasSingleton;
19
20     /** Factory method to get the canvas singleton object. */
21     public static Canvas getCanvas() {
22         if(canvasSingleton == null) {
23             canvasSingleton = new Canvas("BlueJ Shapes Demo",
24                                          300, 300,  Color.white);
25         }
26         canvasSingleton.setVisible(true);
27         return canvasSingleton;
28     }
29
30     // --- instance part ---
31
32     private JFrame frame;
33     private CanvasPane canvas;
34     private Graphics2D graphic;
35     private Color backgroundColour;
36     private Image canvasImage;
37     private List<Object> objects;
38     private HashMap<Object,ShapeDescription> shapes;
39
40     /**
41      * Create a Canvas.
42      * @param title  title to appear in Canvas Frame
43      * @param width  the desired width for the canvas
44      * @param height  the desired height for the canvas
45      * @param bgClour  the desired background colour of the canvas
46      */
47     private Canvas(String title, int width, int height, Color bgColour) {
48         frame = new JFrame();
49         canvas = new CanvasPane();
50         frame.setContentPane(canvas);
51         frame.setTitle(title);
52         canvas.setPreferredSize(new Dimension(width, height));
53         backgroundColour = bgColour;
54         frame.pack();
55         objects = new ArrayList<Object>();
56         shapes = new HashMap<Object,ShapeDescription>();
57     }
```

```
58
59      /**
60       * Set the canvas visibility and brings canvas to the front of screen
61       * when made visible. This method can also be used to bring an already
62       * visible canvas to the front of other windows.
63       * @param visible  boolean value representing the desired visibility of
64       * the canvas (true or false)
65       */
66      public void setVisible(boolean visible) {
67          if(graphic == null) {
68              // first time:  instantiate the offscreen image and fill it
69              // with the background colour
70              Dimension size = canvas.getSize();
71              canvasImage = canvas.createImage(size.width, size.height);
72              graphic = (Graphics2D)canvasImage.getGraphics();
73              graphic.setColor(backgroundColour);
74              graphic.fillRect(0, 0, size.width, size.height);
75              graphic.setColor(Color.black);
76          }
77          frame.setVisible(visible);
78      }
79
80      /**
81       * Draw a given shape onto the canvas.
82       * @param  referenceObject  an object to define identity for this shape
83       * @param  color            the color of the shape
84       * @param  shape            the shape object to be drawn on the canvas
85       */
86      public void draw(Object referenceObject, String color, Shape shape) {
87          objects.remove(referenceObject); // in case it was already there
88          objects.add(referenceObject);    // add at the end
89          shapes.put(referenceObject, new ShapeDescription(shape, color));
90          redraw();
91      }
92
93      /**
94       * Erase a given shape's from the screen.
95       * @param  referenceObject  the shape object to be erased
96       */
97      public void erase(Object referenceObject) {
98          objects.remove(referenceObject); // in case it was already there
99          shapes.remove(referenceObject);
100         redraw();
101     }
102
103     /**
104      * Set the foreground colour of the Canvas.
105      * @param  newColour   the new colour for the foreground of the Canvas
106      */
107     public void setForegroundColor(String colorString) {
108         // implementation detail omitted.
109     }
110
111     /**
112      * Wait for a specified number of milliseconds before finishing.
113      * This provides an easy way to specify a small delay which can be
114      * used when producing animations.
115      * @param  milliseconds  the number
```

```
116         */
117     public void wait(int milliseconds) {
118         // implementation detail omitted.
119     }
120
121     /** Redraw all shapes currently on the Canvas. */
122     private void redraw() {
123         erase();
124         for(Object obj : objects) {
125             ((ShapeDescription)shapes.get(obj)).draw(graphic);
126         }
127         canvas.repaint();
128     }
129
130     /** Erase the whole canvas. (Does not repaint.) */
131     private void erase() {
132         Color original = graphic.getColor();
133         graphic.setColor(backgroundColour);
134         Dimension size = canvas.getSize();
135         graphic.fill(new Rectangle(0, 0, size.width, size.height));
136         graphic.setColor(original);
137     }
138
139     /************************************************************************
140      * Inner class CanvasPane - the actual canvas component contained in the
141      * Canvas frame. This is essentially a JPanel with added capability to
142      * refresh the image drawn on it.
143      */
144     private class CanvasPane extends JPanel {
145         public void paint(Graphics g) {
146             g.drawImage(canvasImage, 0, 0, null);
147         }
148     }
149
150     /************************************************************************
151      * Inner class ShapeDescription
152      */
153     private class ShapeDescription {
154         private Shape shape;
155         private String colorString;
156
157         public ShapeDescription(Shape shape, String color) {
158             this.shape = shape;
159             colorString = color;
160         }
161
162         public void draw(Graphics2D graphic) {
163             setForegroundColor(colorString);
164             graphic.fill(shape);
165         }
166     }
167 }
```

Listing 3: Class Circle

```
import java.awt.geom.*;
/**
 * A circle that can be manipulated and that draws itself on a canvas.
 * @author    Michael Kolling and David J. Barnes
 * @author S H S Wong (modified by)
 * @version 18-02-2011
 */
public class Circle {
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new circle at a specific position with a specific color.
     * @param diameter of the circle
     * @param x     x position
     * @param y     y position
     * @param colour        colour of the diameter
     */
    public Circle(int diameter, int x, int y, String colour) {
        this.diameter = diameter;
        xPosition = x;
        yPosition = y;
        color = colour;
        isVisible = false;
    }

    /**
     * Make this circle visible. If it was already visible, do nothing.
     */
    public void makeVisible() {
        isVisible = true;
        draw();
    }

    /**
     * Make this circle invisible. If it was already invisible, do nothing.
     */
    public void makeInvisible() {
        erase();
        isVisible = false;
    }

    /** Move the circle horizontally by 'distance' pixels. */
    public void moveHorizontal(int distance) {
        erase();
        xPosition += distance;
        draw();
    }

    /** Move the circle vertically by 'distance' pixels. */
    public void moveVertical(int distance) {
        erase();
        yPosition += distance;
        draw();
```

```
58      }
59
60      /**
61       * Change the size to the new size (in pixels). Size must be >= 0.
62       */
63      public void changeSize(int newDiameter) {
64          erase();
65          diameter = newDiameter;
66          draw();
67      }
68
69      /**
70       * Change the color. Valid colors are "red", "yellow", "blue", "green"
71       *  and "black".
72       */
73      public void changeColor(String newColor) {
74          color = newColor;
75          draw();
76      }
77
78      /* Draw the circle with current specifications on screen. */
79      private void draw() {
80          if(isVisible) {
81              Canvas canvas = Canvas.getCanvas();
82              canvas.draw(this, color,
83                          new Ellipse2D.Double(xPosition, yPosition,
84                                               diameter, diameter));
85              canvas.wait(10);
86          }
87      }
88
89      /* Erase the circle on screen. */
90      private void erase() {
91          if(isVisible) {
92              Canvas canvas = Canvas.getCanvas();
93              canvas.erase(this);
94          }
95      }
96  }
```

Listing 4: Class Square

```
1  import java.awt.*;
2  /**
3   * A square that can be manipulated and that draws itself on a canvas.
4   * @author    Michael Kolling and David J. Barnes
5   * @author S H S Wong (modified by)
6   * @version 18-02-2011
7   */
8  public class Square {
9      private int size;
10     private int xPosition;
11     private int yPosition;
12     private String color;
13     private boolean isVisible;
14
15     /**
16      * Create a new square at a required position with the required color.
17      * @param size  size of the square
18      * @param x     x position
19      * @param y     y position
20      * @param colour        colour of the square
21      */
22     public Square(int size, int x, int y, String colour) {
23                 this.size = size;
24                 xPosition = x;
25                 yPosition = y;
26                 color = colour;
27                 isVisible = false;
28     }
29
30     /**
31      * Make this square visible. If it was already visible, do nothing.
32      */
33     public void makeVisible() {
34         isVisible = true;
35         draw();
36     }
37
38     /**
39      * Make this square invisible. If it was already invisible, do nothing.
40      */
41     public void makeInvisible() {
42         erase();
43         isVisible = false;
44     }
45
46     /** Move the square horizontally by 'distance' pixels. */
47     public void moveHorizontal(int distance) {
48         erase();
49         xPosition += distance;
50         draw();
51     }
52
53     /** Move the square vertically by 'distance' pixels. */
54     public void moveVertical(int distance) {
55         erase();
56         yPosition += distance;
57         draw();
```

```
58      }
59
60      /**
61       * Change the size to the new size (in pixels). Size must be >= 0.
62       */
63      public void changeSize(int newSize) {
64          erase();
65          size = newSize;
66          draw();
67      }
68
69      /**
70       * Change the color. Valid colors are "red", "yellow",
71       *   "blue", "green" and "black".
72       */
73      public void changeColor(String newColor) {
74          color = newColor;
75          draw();
76      }
77
78      /* Draw the square with current specifications on screen. */
79      private void draw() {
80          if(isVisible) {
81              Canvas canvas = Canvas.getCanvas();
82              canvas.draw(this, color,
83                          new Rectangle(xPosition, yPosition, size, size));
84              canvas.wait(10);
85          }
86      }
87
88      /* Erase the square on screen. */
89      private void erase() {
90          if(isVisible) {
91              Canvas canvas = Canvas.getCanvas();
92              canvas.erase(this);
93          }
94      }
95  }
```

Listing 5: Class Triangle

```
1  import java.awt.*;
2  /**
3   * A triangle that can be manipulated and that draws itself on a canvas.
4   *
5   * @author    Michael Kolling and David J. Barnes
6   * @author S H S Wong (modified by)
7   * @version 18-02-2011
8   */
9  public class Triangle {
10     private int height;
11     private int width;
12     private int xPosition;
13     private int yPosition;
14     private String color;
15     private boolean isVisible;
16
17     /** Create a new triangle. */
18     public Triangle(int height, int width, int x, int y, String colour) {
19         this.height = height;
20         this.width = width;
21         this.xPosition = x;
22         this.yPosition = y;
23         color = colour;
24         isVisible = false;
25     }
26
27     /**
28      * Make this triangle visible. If it was already visible, do nothing.
29      */
30     public void makeVisible() {
31         isVisible = true;
32         draw();
33     }
34
35     /**
36      * Make this triangle invisible. If it was already invisible, do nothing.
37      */
38     public void makeInvisible() {
39         erase();
40         isVisible = false;
41     }
42
43     /** Move the triangle horizontally by 'distance' pixels. */
44     public void moveHorizontal(int distance) {
45         erase();
46         xPosition += distance;
47         draw();
48     }
49
50     /** Move the triangle vertically by 'distance' pixels. */
51     public void moveVertical(int distance) {
52         erase();
53         yPosition += distance;
54         draw();
55     }
56
57     /**
```

```
58          * Change the size to the new size (in pixels). Size must be >= 0.
59          */
60         public void changeSize(int newHeight, int newWidth) {
61             erase();
62             height = newHeight;
63             width = newWidth;
64             draw();
65         }
66
67         /**
68          * Change the color. Valid colors are "red", "yellow",
69          * "blue", "green" and "black".
70          */
71         public void changeColor(String newColor) {
72             color = newColor;
73             draw();
74         }
75
76         /* Draw the triangle with current specifications on screen. */
77         private void draw() {
78             if(isVisible) {
79                 Canvas canvas = Canvas.getCanvas();
80                 int[] xpoints = { xPosition, xPosition + (width/2),
81                                   xPosition - (width/2) };
82                 int[] ypoints = { yPosition, yPosition + height,
83                                   yPosition + height };
84                 canvas.draw(this, color, new Polygon(xpoints, ypoints, 3));
85                 canvas.wait(10);
86             }
87         }
88
89         /* Erase the triangle on screen. */
90         private void erase() {
91             if(isVisible) {
92                 Canvas canvas = Canvas.getCanvas();
93                 canvas.erase(this);
94             }
95         }
96     }
```