

EXAMINATIONS – 2016
TRIMESTER 2

SWEN222
Software Design

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Instructions: Answer all questions

Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use a separate answer booklet.

Question	Topic	Marks
1.	Object-Oriented Design	30
2.	Functional Design & Contracts	30
3.	Design Patterns I	30
4.	Design Patterns II	30
Total		120

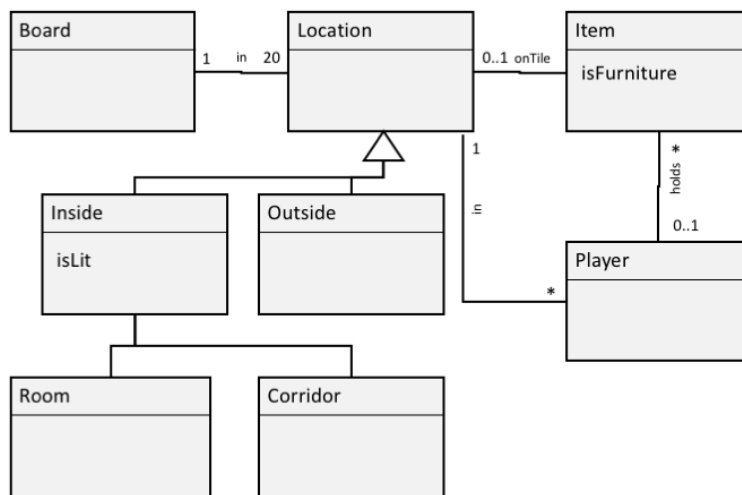
Question 1. Object-Oriented Design

[30 marks]

Consider the following (incomplete) description for a simple board game called *Dungeon*.

“The game board is made up of twenty locations. Each location is adjacent to one or more other locations. Each location is itself made up from one or more tiles which are arranged into a grid where each tile has a specific position. Each location is either a room, a corridor or an outside area. Rooms and corridors are either lit or unlit. An item can be stored in a location on one or more tiles. Every item has a name and description. Some items represent furniture and cannot be moved. All other items can be picked up by one of the players.”

(a) [9 marks] Provide a *class diagram* covering those aspects of the game outlined in the description above. Your diagram should contain at most nine classes.



(b) [9 marks] Provide suitable *Class-Responsibility-Collaborator (CRC)* cards describing the following classes from the game.

Board:	Class: Board <i>Responsibilities</i>	<i>Collaborators</i>
	Store Locations in Grid	Location

Player:	Class: Player <i>Responsibilities</i>	<i>Collaborators</i>
	Know items in Inventory	Item
	Ability to pick up item	
	Know current location	Location
	Know position in room	

Location:	Class: Location <i>Responsibilities</i>	<i>Collaborators</i>
	Know tiles in location	Tile
	Know items store in location	Item
	Know adjacent locations	Location

(c) [12 marks] Provide a straightforward *Java implementation* for each of the three classes in part (b). **You need only consider those aspects of the game described on Page 2.** For example, you do not need to implement player movement, the rules for game over, player input, or other aspects not described. Your classes should include *constructors* and *methods* as necessary.

```

1  public class Board {
2      private Location[] locations;
3
4      public Board(Location[] locations) {
5          this.locations = locations;
6      }
7
8      public Location getLocation(int i) { return locations[i]; }
9      public int numberOfLocations() { return locations.length; }
10 }

1  public class Player {
2      private List<Item> inventory = new ArrayList<Item>();
3      private Location location;
4      private Point position;
5
6      public void setPosition(Location loc, Position pos) {
7          this.location = loc;
8          this.position = pos;
9      }
10
11     public Location getLocation() { return location; }
12     public Point getPosition() { return position; }
13     public void remove(Item item) { inventory.remove(item); }
14     public void add(Item item) { inventory.add(item); }
15 }

1  public class Location {
2      private Location[] adjacents;
3      private Tile[][] tiles;
4      private HashMap<Point, Item> items = new HashMap<Point, Item>();
5
6      public Board(Location[] adjacents, Tile[][] tiles) {
7          this.adjacents = adjacents;
8          this.tiles = tiles;
9      }
10
11     public Location getAdjacentLocation(Direction d) {
12         return adjacents[d.ordinal()];
13     }
14
15     public Location getTile(Point p) {
16         return tiles[p.getY()][p.getX()];
17     }
18
19     public Item getItemAtPosition(Point p) {
20         return items.get(p);
21     }
22
23     public void putItemAtPosition(Point p, Item i) {
24         int endX = p.getX() + i.getWidth();
25         int endY = p.getY() + i.getHeight();
26         for (int x = p.getX(); x < endX; ++x) {
27             for (int y = p.getY(); y < endY; ++y) {
28                 items.put(new Point(x, y), i);
29             }
30         }
31     }
32
33     public void removeItem(Item i) {
34         items.values().removeAll(i);
35     }
36 }

```

Student ID:

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 2. Functional Design & Contracts

[30 marks]

Consider the following class for representing a simple FIFO buffer:

```

1  class QueueBuffer {
2      private int[] items;
3      private int writePos;
4      private int readPos;
5
6      public QueueBuffer(int len) {
7          items = new int[len];
8      }
9
10     public boolean isFull() {
11         return writePos == items.length;
12     }
13
14     public boolean isEmpty() {
15         return readPos == writePos;
16     }
17
18     public void write(int item) {
19         items[writePos] = item;
20         writePos = writePos + 1;
21     }
22
23     public int read() {
24         int item = items[readPos];
25         readPos = readPos + 1;
26         return item;
27     }
28 }

```

(a) [4 marks] An important aspect of the functional programming paradigm is *immutability*. Briefly, discuss what immutability means and whether or not `QueueBuffer` is *immutable*.

An immutable class is one whose instances cannot be *mutated* (i.e. changed) after being constructed. This means its contents do not change for the life of any instance of the class. Class `QueueBuffer` is not immutable because its fields can be changed via the `write()` and `read()` methods. values are

(b) This question is concerned with method *side-effects*.

(i) [4 marks] Briefly, discuss what is meant by the term *side-effect free*.

A method is side-effect free if it does not modify any state which existing before it was called, does not perform any I/O (e.g. printing to `System.out`) and does not call any methods which are not side effect free.

For each of the following methods, briefly discuss whether or not it is *side-effect free*.

(ii) [2 marks] `QueueBuffer.isFull()`

This is side-effect free as it does not change state

(iii) [2 marks] `QueueBuffer.write(int)`

This is not side-effect free as it modifies the fields `items` and `writePos`

(iv) [2 marks] `QueueBuffer.read()`

This is not side-effect free as it modifies the field `readPos`

(c) [8 marks] Rewrite the `QueueBuffer` class to use a functional design.

```

1  class FunctionalQueueBuffer {
2      private int[] items;
3      private int writePos;
4      private int readPos;
5
6      public FunctionalQueueBuffer(int len) {
7          items = new int[len];
8      }
9
10     private FunctionalQueueBuffer(int[] items) {
11         this.items = Arrays.copyOf(items, items.length);
12     }
13
14     public boolean isFull() {
15         return writePos == items.length;
16     }
17
18     public boolean isEmpty() {
19         return readPos == writePos;
20     }
21
22     public FunctionalQueueBuffer write(int item) {
23         FunctionalQueueBuffer r = new FunctionalQueueBuffer(items);
24         r.items[writePos] = item;
25         r.writePos = writePos + 1;
26         return r;
27     }
28
29     public Pair<FunctionalQueueBuffer, Integer> read() {
30         FunctionalQueueBuffer r = new FunctionalQueueBuffer(items);
31         r.readPos = readPos + 1;
32         return new Pair<>(r, items[readPos]);
33     }
34 }

```


(d) The question concerns the original implementation of `QueueBuffer` on page 6. For each method below, provide appropriate *preconditions* and *postconditions*:

(i) [2 marks] `QueueBuffer(int len)`

REQUIRES: `len >= 0`

ENSURES: `isEmpty() && (len == 0 || !isFull())`

(ii) [2 marks] `write(int item)`

REQUIRES: `!isFull()`

ENSURES: `!isEmpty()`

(e) [4 marks] Give an appropriate *class invariant* that you would enforce for the `QueueBuffer` class, and discuss how you would enforce the invariant.

```
items != null &&  
0 <= readPos && readPos <= writePos && writePos <= items.length
```

Student ID:

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID:

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

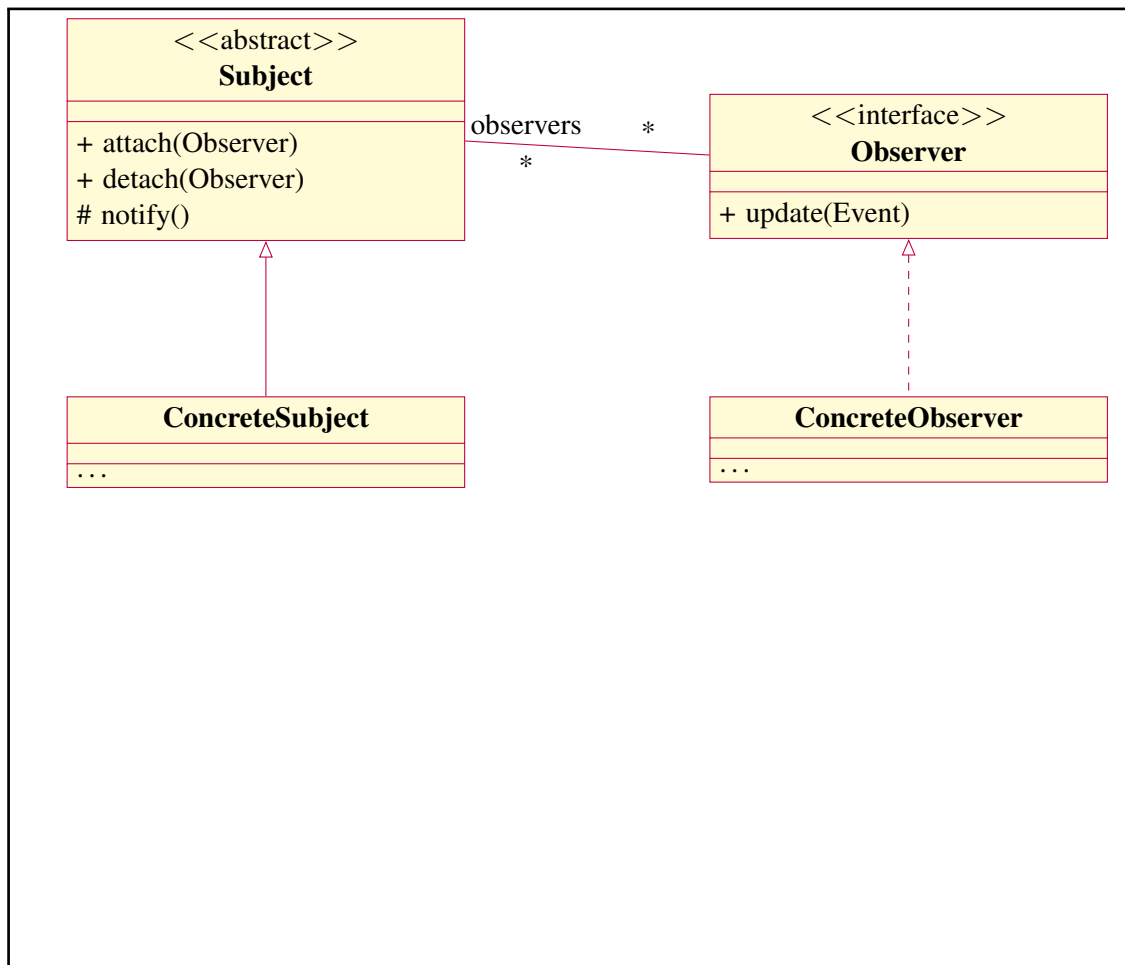
Question 3. Design Patterns I

[30 marks]

(a) This question is concerned with the OBSERVER pattern.

(i) [4 marks] Briefly, describe the *problem* being solved by the OBSERVER pattern.

A *subject* object needs to notify a set of *observer* objects every time an event occurs, but the subject should not be coupled with the implementation of the observer.

(ii) [5 marks] Provide an appropriate *class diagram* which describes the OBSERVER pattern in an abstract sense.

(Question 3 continued)

(iii) [6 marks] A *listener* can be added to certain *items* that produce *events*. For example, *alarms* may produce *activation* and *deactivation* events. Different kinds of listener can respond to these events in different ways.

Sketch an implementation of these classes which uses the OBSERVER pattern.

```

1  abstract class Item {
2      private Set<Listener> listeners;
3
4      ...
5
6      public void attach(Listener listener) {
7          listeners.add(listener);
8      }
9
10     public void detach(Listener listener) {
11         listeners.remove(listener);
12     }
13
14     protected void notify(Event event) {
15         for (Listener listener : listeners) {
16             listener.update(event);
17         }
18     }
19 }

1  interface Event { ... }

1  interface Listener {
2      public void update(Event event);
3  }

1  class Alarm extends Item {
2      public void activate() {
3          notify(new ActivationEvent());
4      }
5
6      public void deactivate() {
7          notify(new DeactivationEvent());
8      }
9
10     class ActivationEvent implements Event { ... }
11     class DeactivationEvent implements Event { ... }
12 }

```

(Question 3 continued on next page)

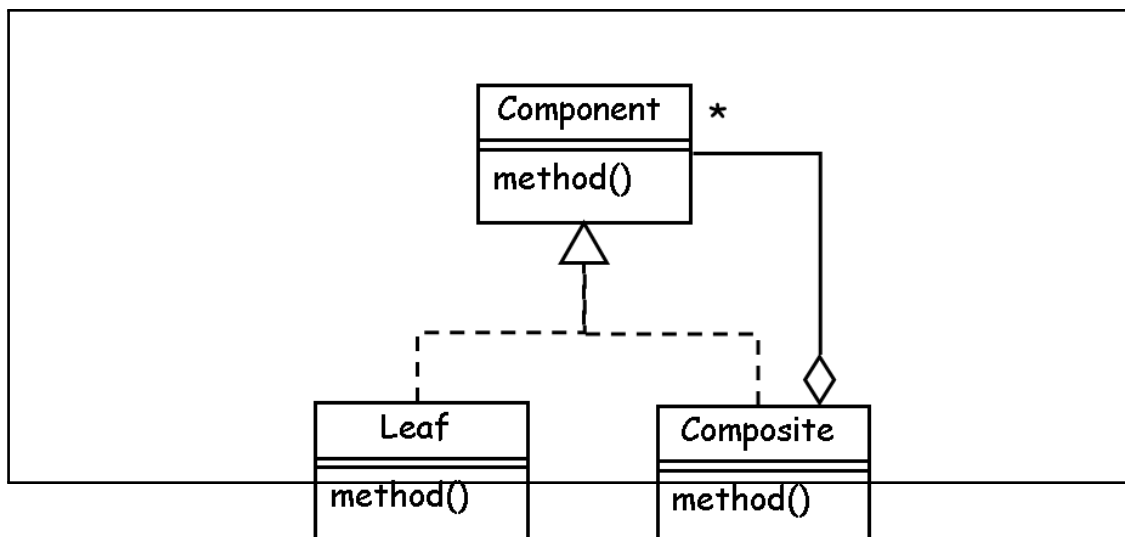
(Question 3 continued)

(b) This question is concerned with the COMPOSITE pattern.

(i) [4 marks] Briefly, describe the *problem* being solved by the COMPOSITE pattern.

A *composite* object needs to store both simple objects and other composite objects in a tree-like hierarchy, but should not be coupled to the implementation of any of these objects. A common interface describes the behaviour of all objects that can appear in the hierarchy.

(ii) [5 marks] Provide an appropriate *class diagram* which describes the COMPOSITE pattern in an abstract sense.



(Question 3 continued)

(iii) [6 marks] An *XML Object* has the form “<tag attrs>...</tag>”, where *tag* is the *object name*, and *attrs* a sequence of zero or more *attributes*. Each attribute has the form “attr=str”, where *attr* is the *attribute name* and *str* a string constant. An XML Object may contain zero or more XML Objects within it. The following illustrates a simple example:

```

1 <project name="wyc">
2   <target name="compile">
3     </target>
4   <target name="test" depends="compile">
5     </target>
6 </project>

```

Sketch a Java implementation for representing XML Objects which uses the COMPOSITE pattern.

```

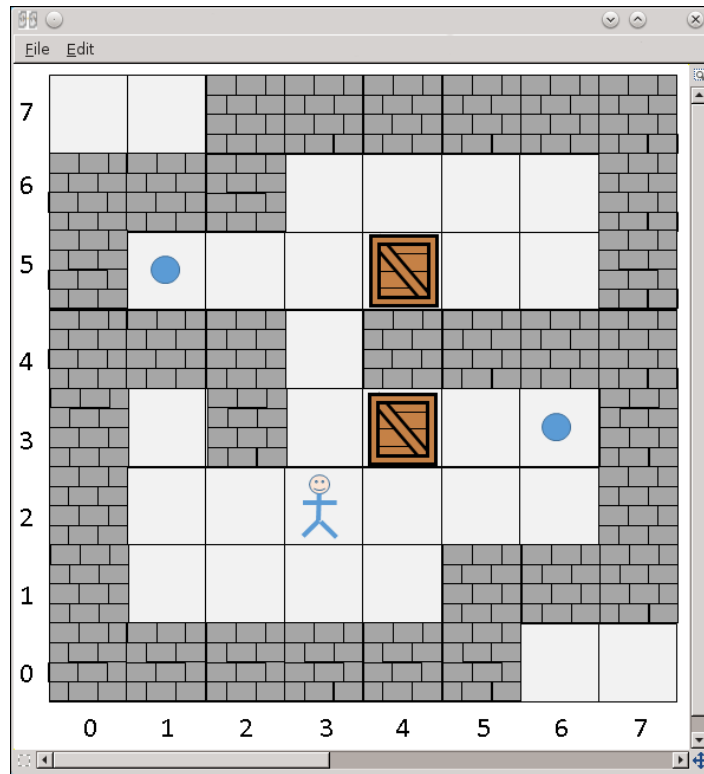
1 class XMLObject {
2     private String name;
3     private Map<String, String> attributes;
4     private List<XMLObject> children;
5
6     ...
7
8     // Some recursive operation
9     public boolean containsName(String search) {
10         if (name.equals(search)) { return true; }
11         for (XMLObject child : children) {
12             if (child.containsName(search)) {
13                 return true;
14             }
15         }
16
17         return false;
18     }
19 }

```

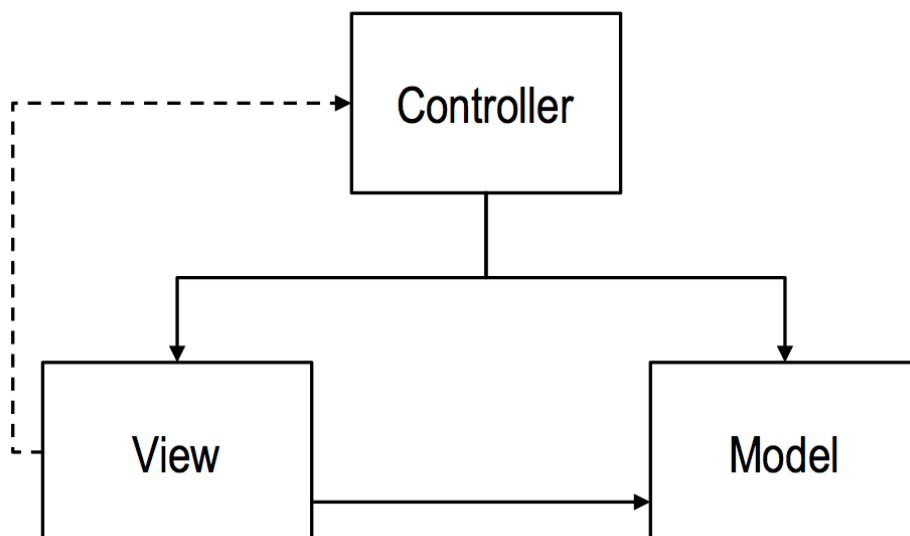
Question 4. Design Patterns II

[30 marks]

You have been asked to design an implementation of the popular game *Sokoban* which uses a *Graphical User Interface (GUI)*. In the game, the player uses the keyboard to move his/her *character* around the board. The goal is to push *crates* into *storage locations* and, when every *crate* is placed in a *storage location*, the game is over. The player's character cannot move through *crates* or *walls* and cannot push more than one *crate* at a time. The following illustrates how the game might look:



You intend to use the *Model/View/Controller* design pattern in your implementation:



(a) [5 marks] Briefly, discuss the features of the game that the *Model* component would be responsible for.

Stores the current state of the game, provides an interface for manipulating this state (to be used by the Controller), and ensures that any attempts to change the state are valid moves by encoding the rules of the game.

(b) [5 marks] Briefly, discuss the features of the game that the *View* component would be responsible for.

Displays the game view to the user given the necessary information about the state in the model, and provides an interface for notifying (the Controller) when the user interacts with the display.

(c) [5 marks] Briefly, discuss the features of the game that the *Controller* component would be responsible for.

Handles communication between the View and the Model by registering for notifications from the View, translating these user interactions into calls on the Model, and notifying the View to update its display when the Model is changed or an invalid interaction occurs.

(d) [5 marks] The DECORATOR pattern is often used when implementing a graphical user interface. Briefly, outline how this pattern might be used in the Sokoban game.

The Decorator pattern can be applied to build composable GUI features by dynamically *wrapping* existing GUI components to provide the same interface with some extended behaviour. For instance, the scrollbar around the main GUI display could be implemented with a decorator wrapped around a simpler window component.

(e) Someone suggested implementing a *command-line* version of the Sokoban game. This would draw the board using a simple text-based user interface, rather than a graphical user interface.

(i) [5 marks] Briefly, discuss how the Model/View/Controller pattern makes it easier to add a text-based user interface.

The part of the program that is specific to the command-line can just be a different implementation of the View with the same interface, and then the Model and Controller do not need to be changed at all. Not only is this easier, it makes it less likely that the two versions of the game will have accidentally different behaviour.

(ii) [5 marks] Briefly, discuss how implementing both command-line and GUI versions of Sokoban might uncover issues with the separation of *Model* and *View*.

If the Model and the View are not sufficiently decoupled then it will be difficult to just replace the View, since the Model will depend on behaviour that is no longer present. This will help identify places where the implementation details of the View have leaked into the Model.

* * * * *

Student ID:

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Student ID:

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.