THE UNIVERSITY OF MELBOURNE

School of Computing and Information Systems

SWEN30006 Software Modelling and Design

Answers: 2018 End of Semester 1

Note that answers are indicative and variations are possible.

Question 1. [15 marks]

a. Representational gap [5]

- Definition [2]: The representational gap is the difference between the model of the problem domain (real world) and the model of the design (software system), that is the differences in the objects, their names, attributes, and relationships.

- Importance [3]: Maintaining a low representation gap makes the design easier to comprehend and easier to update corresponding to changes in the domain. However, to meet stringent non-functional requirements (e.g. re performance), the rep. gap may need to be larger, e.g. by combining domain objects in a single design object to reduce retrieval overheads. This should happen where strictly required.

b. Coupling [5]

- Definition [2]: Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

- Importance [3]: Low coupling is a design aim or GRASP principle (or pattern). The problem low coupling addresses is: How to support low dependency, low change impact, and increased reuse? The solution involves assigning responsibilities so that coupling remains low. This is a critical design principle which should be used to evaluate alternative design options.

c. Cohesion [5]

- Definition [2]: Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

- Importance [3]: High cohesion is a design aim or GRASP principle (or pattern). The problem it addresses is: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling? The solution involves assigning responsibilities so that cohesion remains high. This is a critical design principle which should be used to evaluate alternative design options.

Question 2. [10 marks]

a. Description class. [5]

A description class is a class that captures information about an item, service or other element independent of the existence of any instances of those elements. It reduces redundant or duplicated information, while ensuring that there is no loss of information if all instances of the described element are removed.

For example, a ProductDescription class (with attributes like price, catalogueId, description) could be associated with a Product class. Every product has an associated description, and many products will share the same description. Some descriptions may have no products.

(A simple class diagram could be provided to illustrate this:

[Product]--0..*-------- described by ----------1– [Product Description]

)

b. Composition [5]

Composition, also known as **composite aggregation**, is a strong kind of whole-part relationship. In UML it is modelled as a form of association shown using a filled diamond at the composite end. A composition relationship implies that 1) an instance of the part (such as a *Square*) belongs to only *one* composite instance (such as one *Board*) at a time, 2) the part must *always belong* to a composite (no free-floating *Squares),* and 3) the composite is responsible for the creation and deletion of its parts—either by itself creating/deleting the parts, or by collaborating with other objects. Related to this constraint is that if the composite instance is destroyed/removed, its parts must either be removed, or associated with another composite instance. For example, if a physical paper Monopoly game board is destroyed, we think of the squares as being destroyed as well (a conceptual perspective).

Question 3. [10 marks]

a. Describe the Information Expert pattern and provide an example to illustrate its application. [5]

Problem
What is a general principle of assigning responsibilities to objects?
A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. If we've chosen well, systems tend to be easier to understand, maintain, and extend, and our choices afford more opportunity to reuse components in future applications.
Solution
Assign a responsibility to the information expert—the class that has the *information* necessary to fulfill the responsibility.

For example, who should be responsible for providing the grand total for a Sale. You expect the Sale object to know the details (list of line items making up the Sale) and so the Sale is the obvious place to assign that responsibility.

b. Describe the Creator pattern and explain why particular Creator options are given priority over others. [5]

Problem
Who should be responsible for creating a new instance of some class?
The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.
Solution
Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.

B is a *creator* of A objects.
If more than one option applies, usually prefer a class B which *aggregates* or *contains* class A.

A Sale contains its SalesLineItems and so Sale is the obvious place to assign responsibility for creating SalesLineItems. Composition is a strong relationship, a class can only be aggregated by one composite, and creation/destruction is part of that relationship. Containment is a weaker form of relationship, but still a strong relationship. With either of these relationships you already have strong coupling so this addition responsibility is cohesive and doesn't add significantly to coupling.

Question 4. [7 marks]

In the context of mapping a design to code:

a. [2] The approach for translating design classes to code is to work from least coupled to most coupled. Briefly justify this guideline.

Classes need to be implemented (and ideally, fully unit tested) from least-coupled to most-coupled. This way, when mapping a class, most or all of its dependent classes are already mapped, and so the current class can be defined in terms of classes already mapped.

b. [5] Explain how design associations with multiplicities greater than one can be translated to code, and what should guide the choice of representation. Provide an example.

One-to-many relationships are common. For example, a *Sale* must maintain visibility to a group of many *SalesLineItem* instances. In OO programming languages, these relationships are usually implemented with the introduction of a **collection** object, such as a *List* or *Map*, or even a simple array.

For example, the Java libraries contain collection classes such as *ArrayList* and *HashMap*, which implement the *List* and *Map* interfaces, respectively. Using *ArrayList*, the *Sale* class can define an attribute that maintains an ordered list of *SalesLineItem* instances.
The choice of collection class is of course influenced by the requirements; key-based lookup requires the use of a *Map*, a growing ordered list requires a *List*, and so on.
As a small point, note that the *lineItems* attribute is declared in terms of its interface.
Guideline: If an object implements an interface, declare the variable in terms of the interface, not the concrete class. E.g. **private List lineItems = new ArrayList();**

Many-to-many can be represented using a different collection valued attribute in the object at each end of the association (this requires all updates being applied to both collections), or by an instance of a table class, referenced in the object at each end of the association.

Question 5. [10 marks]

Question 5 Part 1. [4 marks]

Briefly describe architectural analysis and why it is important.

Architectural analysis can be viewed as a specialization of requirements analysis, with a focus on requirements that strongly influence the "architecture." For example, identifying the need for a highly-secure system.
The essence of architectural analysis is to identify factors that should influence the architecture, understand their variability and priority, and resolve them.

Why is architectural analysis important? It's useful to:

- reduce the risk of missing something centrally important in the design of the systems
- avoid applying excessive effort to low priority issues
- help align the product with business goals

Question 5 Part 2. [6 marks]

a. An architecturally significant functional requirement.
Examples given in lecture below (others possible): It needs to be a functional requirement which touches elements across the system/architecture, such that, if it was not considered initially, would require substantial effort and design change to support later.

Licensing	Printing
Auditing	Reporting
Localization	Security
Mail	System management
Online help	Workflow

b. An architecturally significant non-functional requirement.
Examples given in lecture below (others possible): it needs to be a non-functional requirement which is a quality of the system as a whole, and which would be difficult or impossible to achieve if not accounted for initially.

Usability: e.g. aesthetics and consistency in the UI.

Reliability: e.g. availability (the amount of system "up time"), accuracy of system calculations, and the system's ability to recover from failure.

Performance: e.g. throughput, response time, recovery time, start-up time, and shutdown time.

Supportability (or maintainability or extendability) – design/code properties.

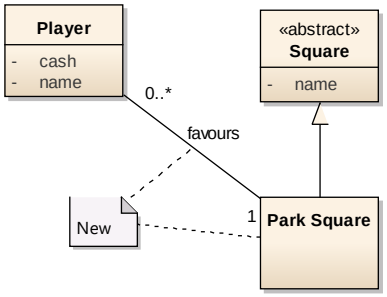
Question 6. [18 marks]

This question relates to the domain model below. This is the iteration-3 domain model for the Monopoly case study from Applying UML and Patterns, 3rd Ed. by Larman.

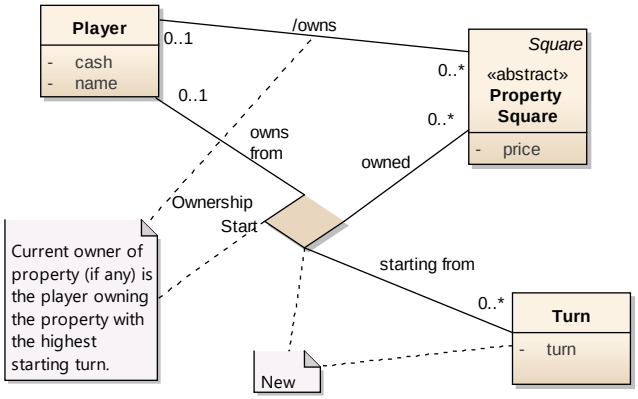
a) [2] Is-On multiplicity constraint 0..8 should change to 0..1.

b) [2] No change required.

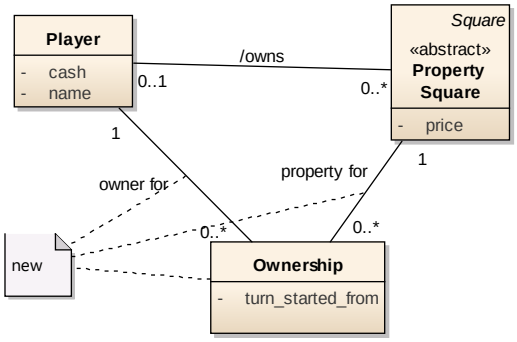
c) [6] Add a new class Park Square as a subclass of Square (not Property Square). Add a new association "favours" from Player to Park Square with multiplicity * (or 0..8) at Player and 1 at Park Square.



d) [8] Add a new association. E.g. Ternary association Owner, Property, [Starting] Turn (new class). Also should (ideally) modify existing owns to /owns (derived) and have a comment saying how it can be derived from the new one (existing owner is the one with the highest turn.



Alternate version:



Question 7. [27 marks]

Question 7 Part 1. [5 marks]

Protected Variations

[1] Problem

How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

[1] Solution

Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Note: The term "interface" is used in the broadest sense of an access view; it does not literally only mean something like a Java interface.

Two points of change are worth defining:

- **[1] variation point**—Variations in the existing, current system or requirements, such as the multiple tax calculator interfaces that must be supported.
- **[1] evolution point**—Speculative points of variation that may arise in the future, but which are not present in the existing requirements.

[1] This is an example of the former.

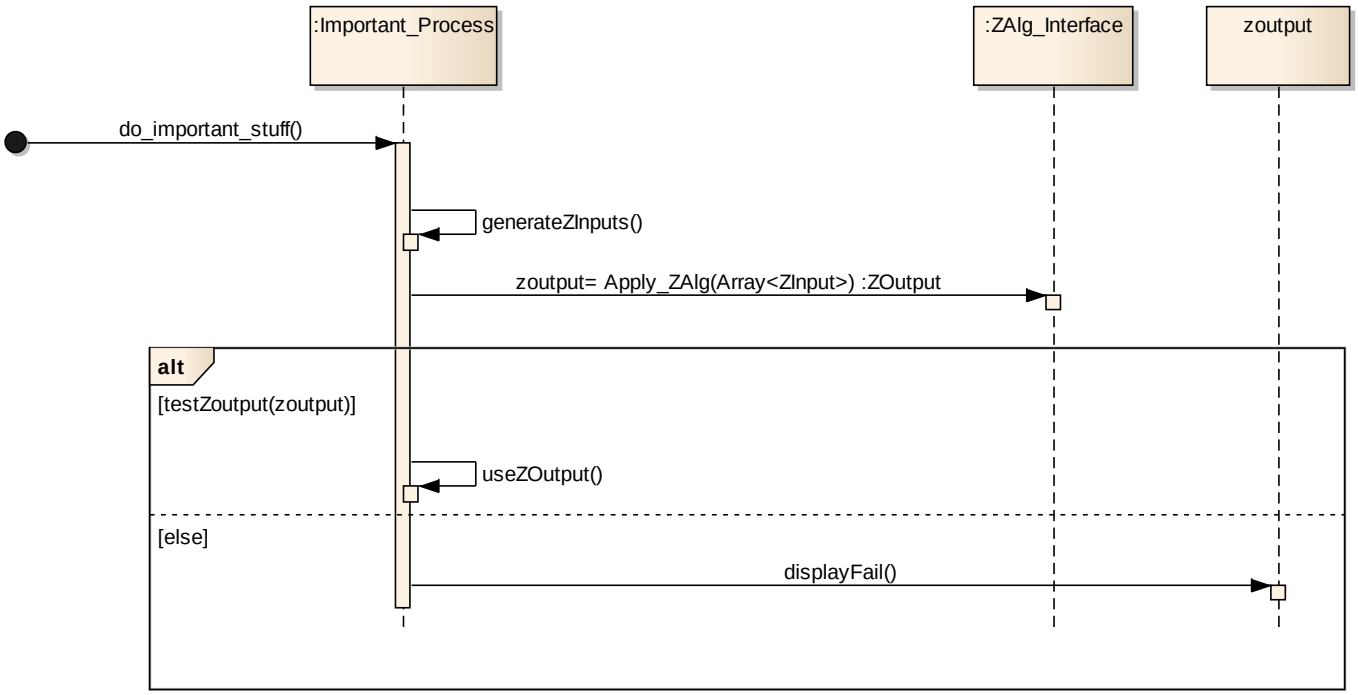
Question 7 Part 2. [7 marks]

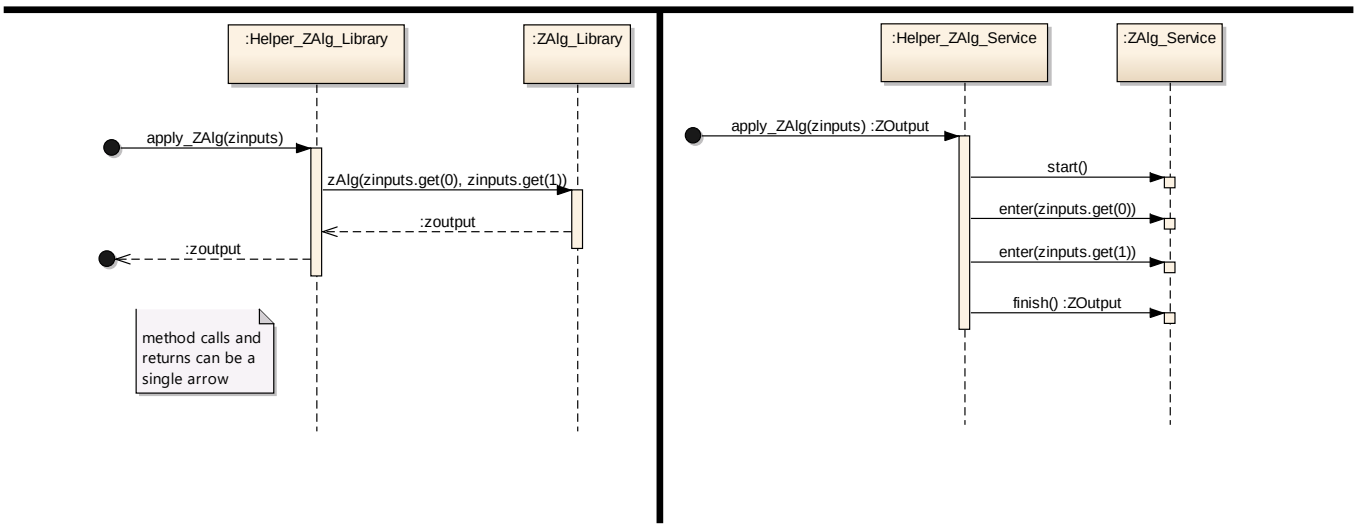
[3] Name:	**Adapter**
[1] Problem:	How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
[1] Solution: (advice)	Convert the original interface of a component into another interface, through an intermediate adapter object.

[2] The two "Helper_" classes represent the adaptors in this example, the components referred to are ZAlg_Library and ZAlg_Interface.

Question 7 Part 3. [15 marks]

We expect one diagram (top) in terms of ZAlg_Interface, and 2 diagrams (bottom) for each of the polymorphic sub-cases. Code was provided so the diagrams should match the code, as per below. No information was provided as to whether the condition "testZOutput(zoutput)" in do_important_stuff() succeeds or fails so the solution should include a frame that represents both cases, again matching the code.



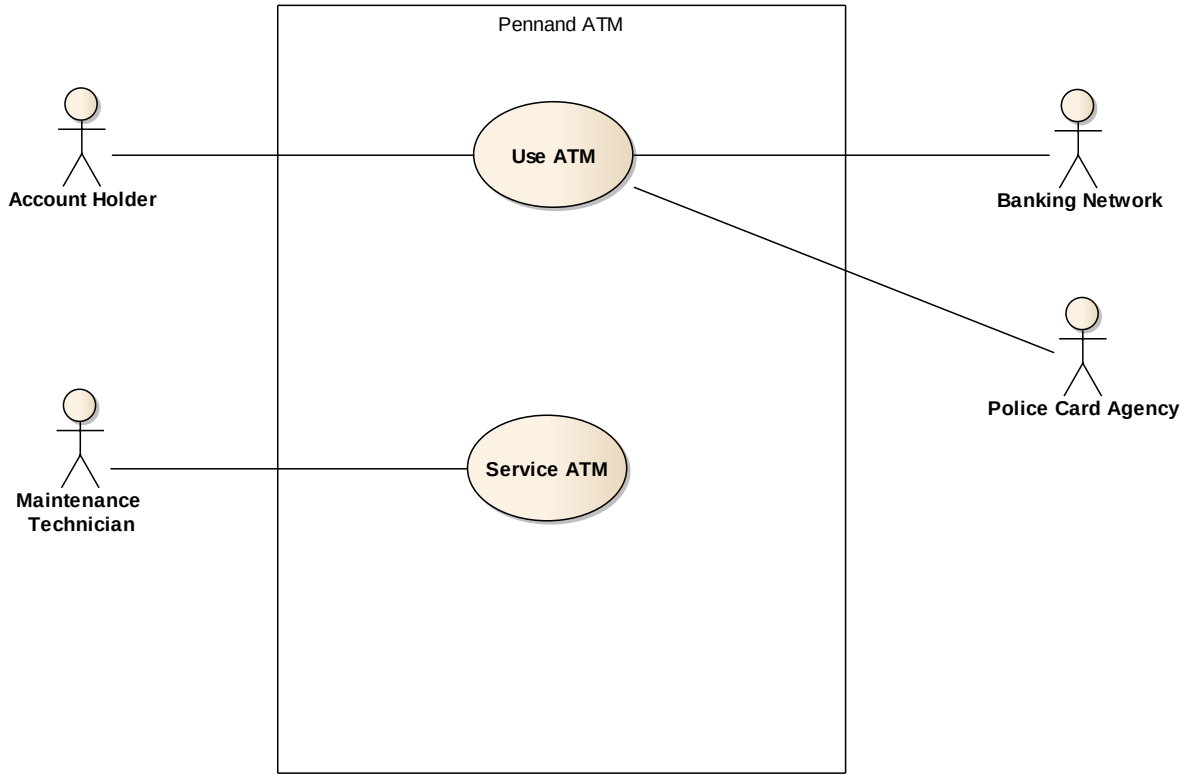


Question 8. [23 marks]

The following questions refer to the passage below which describes the Pennand ATM (Automated Teller Machine). The sentence numbering is provided to make it easier for you to track the various elements of the description.

Question 8 Part 1. [5 marks]

Draw a use case diagram, covering the Pennand ATM description above.



Question 8 Part 2. [18 marks]

Draw a state machine diagram for the Pennand ATM, based on the description above.