# SOEN 341 *Software Process*

# Midterm 1

# Solutions and Marking

**Problem 1: Software process models (20%).**

(a) The Waterfall Model has been criticized for being “document driven”.

- What does this mean?

  **Answer:** The Waterfall Model requires the creation of a series of documents, each of which must be completed before the next is started. A limited amount of feedback is allowed: documents in the preceding phase may be modified while the documents of the current phase are being prepared. Milestones of the project record completion of documents: hence the description “document driven”.

- Why is it considered to be a disadvantage of the Waterfall Model?

  **Answer:** Mistakes made in early phases cannot easily be changed because many documents would have to be altered. Consequently, the Waterfall Model cannot deal with changing requirements, although such changes are common in practice. Clients have to judge progress on the basis of documents that they may not completely understand, and do not see working software until late in the development life cycle.

(b) Boehm’s Spiral Model is often called a “risk driven” model.

- What does this mean?

  **Answer:** Each phase of the development includes an analysis of the current risks facing the project. A “risk” is simply something that might cause problems later: it does not imply that there is necessarily something “risky” or dangerous about the project. During each phase, some part of the software that addresses the risks is developed or prototyped, enabling clients to see progress in terms of working software rather than documents.

- Why is it considered to be an advantage of the Spiral Model?

  **Answer:** Addressing perceived problems as soon as possible reduces the possibility of encountering serious problems at a later stage when it may be difficult or impossible to solve them. If a risk discovered early on shows that the project is infeasible, the project can be cancelled without large losses. Changing requirements can be introduced as “risks” for the next development phase. The Spiral Model reduces the chances of wasting effort and having to redo work.

**Marking:** 5 marks for each of the four questions.

**Problem 2: Use cases (20%).**

(a) During which phases of software development are use cases:

- written?

  **Answer:** Use cases are written during domain analysis and requirements gathering.

- used?

  **Answer:** Use case are used during the design and testing phases. Designers can check that the design supports each use case. Testers can write tests based directly on use cases. Use cases are not employed much during implementation, since they should have been incorporated into the design, and implementors need to see only the design.

(b) What features of a software system *cannot* be specified with use cases?

**Answer:** Functions performed by the system that are not a direct consequence of user actions. For example, any operation that is executed at fixed times, rather than on-demand, such as a nightly back-up.

Also, quality requirements (a.k.a. non-functional requirements) cannot be specified by use cases. Thus a use case cannot be used to specify the appearance of the user interface, a timing requirement, etc.

(c) Explain the role of a precondition in a use case.

**Answer:** A *precondition* is an assertion about the state of the system that must be true before the use case is executed.

The precondition is an *obligation* for the system: the system must be in a certain state before the use case is allowed to proceed. E.g., a client must be logged-in to perform a transaction.

A precondition is a *statement*, not an action. “Client is logged-in” is a precondition but “the system has logged-in the client” is not a precondition.

(d) Explain the role of a postcondition in a use case.

**Answer:** A *postcondition* is an assertion about the state of the system that is guaranteed to be true after the use case has been executed.

A postcondition is a *guarantee* that the system will be in a certain state after the use case completes. The implementation of the use case must ensure that the postcondition becomes true.

Postconditions are sometimes written in the future tense. E.g., “The client will be logged-in to the system”.

Like preconditions, postconditions are not actions: they just state some property that must become true as a result of the use case.

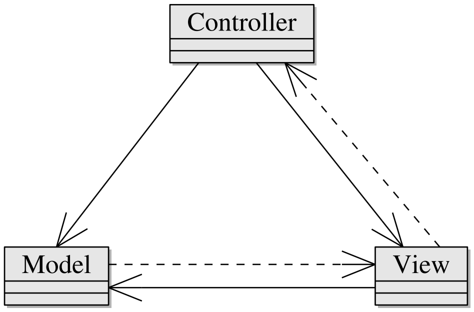
**Marking:** 5 marks for each part.

**Problem 3: Software architecture (20%).**

(a) Give a brief outline of the Model-View-Controller (MVC) architecture.

**Answer:** The MVC architecture has three main components, as shown in the diagram below:

- The *Model* contains the application code (a.k.a. "business logic") and provides the functionality of the system. The Model may be asked to do things and queried about its state, but does not know the sources of these requests.
- The *view* provides information about the model to the user, typically via a Graphical (or other) User Interface. The View sends requests for information to the model.
- The *Controller* coordinates the activities of the other components. In a typical cycle, the Controller receives a request from the user to do something (this request may come from the UI). It sends a request to the Model to perform the action, and then it sends a signal to the View indicating that the state of the model may have changed.



(b) List some of the ways in which MVC respects good design principles.

**Answer:**

- **Low coupling:** The coupling between all components, and especially between the Model and the View, is kept to a minimum to simplify development and maintenance.
- **High cohesion:** Each component performs a simple, well-defined task. The Model, for example, does not know anything about how it will be seen by the user. Thus each component can be cohesive.
- **Divide and conquer:** The problem is split into two parts (application logic and user interface), with a smaller component for control. The computation and viewing parts are both simpler than the complete application.
- **Reuse:** Models and views can be reused in other applications, at least in principle. Reuse is more feasible for simple components than for the application as a whole.
- **Testability:** It is easier to test individual components (e.g., the Model has the correct functionality and the View display data as required) than it is to test the entire system together.
- **Portability:** A change in display hardware requires only a change in the View component; a change in the processor (e.g., to multicore) requires only a change to the Model component.
- **Separation of Concern:** Work on the application logic is distinct from work on the UI.

**Marking:** 10 marks for each part. For part (b), each principle with an explanation of how it applies gets 2 or 3 marks, with a maximum of 10. Although the question only asks for a list, it would be too easy simply to list all of the design principles and claim full marks. Thus a list of principles without explanations has a maximum mark of 8.

---

**Problem 4: Requirements (10%).**

The requirements document for a large computer-controlled machine contained the following statement:

> *It shall be possible to start the machine only*
> *if the safety guard is in the closed position.*

The requirement was implemented in the following way:

- The machine is equipped with a lamp and a photocell. When the guard is in the closed position, it blocks the beam from the photocell.
- The software has a function `beamOn()` which returns `true` when the photocell is illuminated.
- The software contains a method with this structure:

```
void start() {
    if (beamOn()) {
        print("Start failed: safety guard is not closed.");
        return;
    }
    // start machine
    ....
}
```

(a) Identify any potential problems with this system.

**Answer:** What could go wrong with the system as described? The most likely cause of failure is probably a burned-out bulb. Unfortunately, the software cannot distinguish the two situations "guard not closed" and "lamp burned out" because, in each case, no light will reach the photocell. In engineering parlance, the system is not *fail safe*.

(b) Suggest a better way of implementing the requirement. (You must use a lamp and a photocell.)

**Answer:** The system should not allow the machine to start if the lamp has burned out. Thus the logic should be reversed; the guard should interrupt the beam when it is *open*. If the lamp burns out, the software will assume (incorrectly) that the guard is open, and will not start the machine. It has failed, but in a way that does not endanger the operator.

**Discussion:**

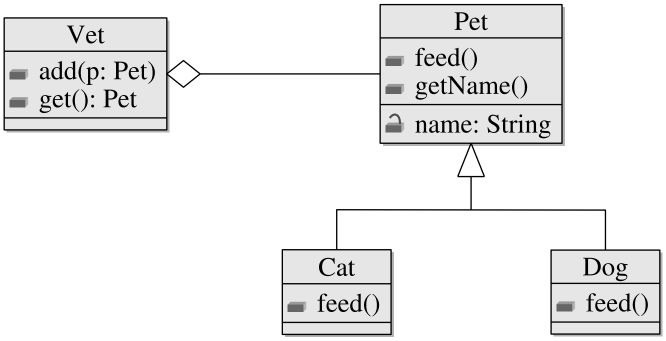
- "Fail safe" means that a failure of some component should not make a system more dangerous. As far as possible, systems should always fail in a way that prevents them from violating safety requirements. Of course, this is not always possible: an aircraft can hardly be expected to "fail safely" if a wing falls off.
- The second most likely cause of failure (after "lamp burned out") is "blocked beam". The same principle applies: if the beam is blocked, it should not be possible to start the machine.
- A lamp and a photocell seems a rather fragile mechanism. Perhaps we should check the position of the guard "directly". But what does this mean? There must be some physical mechanism that communicates the guard position to the software. For example, we could use a microswitch. But microswitches can also fail, and again we would have to design the mechanism in a fail safe way.

  If we can check the position of the guard without the lamp and photocell, we don't need the lamp and photocell, except possibly for redundancy.
- Testing that the lamp is on introduces similar difficulties. How do we test that the lamp is on? Perhaps another photocell — but that still leaves us with the problem of a blocked beam, failed photocell, etc. We could measure the current flowing through the lamp, but lamps sometimes short-circuit when they burn out, and still draw current.
- The requirement implies that it should *not be possible* to start the machine with the guard open. Asking the operator to ensure that the guard is closed before starting does not meet this requirement.
- The `return` statement ensures that the machine will not start if `beamOn()` returns `true`. The ellipsis (`....`) after the comment allows for further actions while the machine is running.
- Safety can be improved by redundancy: we could use two lamps and two photocells, etc. But the fail-safe principle is still the key to a good solution.

**Marking:** 5 marks for reporting the "fail safe" issue and another 5 marks for reversing the logic to correct it. 3/5 for lamp checks or testing the guard directly. 3/5 for asking the operator. 4/5 for a double check such as `(lampOn() && beamOn())` (even though, for the reasons given above, this is not a good solution).

**Problem 5: Class diagrams (15%).**

Explain the features of the class diagram below. If a feature is repeated (e.g., `feed()`), explain it once only. You do not have to guess the meaning of the names; just explain their roles in the diagram (e.g., class, public method, etc.).



**Answer:**

- This is a UML class diagram.
- There are four classes: `Vet`, `Pet`, `Cat`, and `Dog`.
- `Vet` is an aggregation of `Pet`s.
- The hollow diamond indicates that destroying a `Vet` does not mean that we have to destroy the `Vet`'s `Pet`s.
- Classes `Cat` and `Dog` both inherit (triangle) from class `Pet`.

- There four are methods: `add()`, `get()`, `feed()`, `getName()`.
- We do not know for certain what the methods do, but the names suggest:
  1. `add()` adds a new `Pet` to the `vet`'s collection. It has one parameter of type `Pet`.
  2. `get()` obtains a `Pet` from the `Vet`'s collection. It has no parameters but `returns` a `Pet`.
  3. `feed()` provides food for a `Pet`
  4. `getName()` returns the name of a `Pet`
- All of the methods are `public`, as indicated by a padlock with no locking bar.
- The method `feed()` is defined in classes `Cat` and `Dog`: these versions presumably override the definition in class `Pet`.

- Only one variable is shown: `name` in class `Pet` has type `String`. Since it is `protected` (open padlock), it can be accessed by the child classes `Cat` and `Dog`, but not by other classes.

The diagram has more detail than we would expect in a domain model but, considered as a class diagram, is quite abstract. There are no multiplicities, parameter types and return types are omitted except in a few cases, and there are probably missing instance variables, too.

**Marking:** 15 marks for getting all of the features listed above. One mark off (approximately) for each feature omitted or described erroneously.

---

**Problem 6: User-centered design (15%).**

(a) What are the disadvantages of having users involved in user-interface design (UID)?

**Answer:** Users:

- may not know what they really want
- may be prejudiced by UIs that they are familiar with but which are inappropriate for the application
- may not realize the full potential of the UI
- may have differing levels of expertise
- are probably not experts in UI design

There is also the possibility that user involvement will lead to changing requirements, increasing costs, etc.

(b) Explain the term *affordance* as it applies to UID.

**Answer:** The *affordance* of a UI control is the set of actions that the control can initiate. The affordance of a `File` button, for example, typically includes `Open`, `Close`, `Save`, `Save As`, etc.

Affordance is also sometimes used to mean the functions suggested by the appearance of a *control.* For example, a scroll bar is designed to suggest that text may be scrolled by dragging the scroll icon.

Donald Norman, who introduced the term into software development, admits that he should have distinguished "real affordance" (the functionality that the control actually has) and "perceived affordance" (the functionality that its appearance suggests).[1]

The term "affordance" has nothing to do with "affording" in the monetary sense (as in "We cannot afford to purchase this system").

(c) What are the possible effects of "undo" for a UI control?

**Answer:**

- All effects of the most recent action are cancelled and the system reverts to its previous state. A second 'undo' cancels the first 'undo' and returns the system to its state before the first 'undo'.
- A 'multi-undo' allows the user to step backwards through a number of actions. For example, the sequence $A_1\,A_2\,A_3\,A_4\,U\,U$, in which $A_i$ is an action and $U$ is 'undo', is equivalent to $A_1\,A_2$.
- A 'menu-undo' presents the user with a menu of actions that can currently be undone. The menu displays actions for which 'undo' is possible (e.g., delete a paragraph) but not actions that cannot be undone (e.g., format a disk).

[1] `http://www.jnd.org/dn.mss/affordances_and.html`

Additional points that may be made:

- Undo may lead to inconsistencies in the state, memory leaks, etc. However, this should not happen if undo is correctly implemented. If undoing an action would lead to an inconsistent state, the application should not allow undo.
- Undo, especially multi-undo, may be expensive and slow because resources are required to support it.
- Undo may be difficult to implement.

**Marking:** 5 marks for each part. Answers suggested that the wording of the questions was not clear enough, so generous part-marks were awarded.