**BCT 2309: OBJECT ORIENTED SYSTEMS AND DESIGN**

**ASSIGNMENT-2020**

**Hezion Kwena M**

**SCT212-0222/2017**

**COMPUTER TECHNOLOGY YEAR 3.1**

Q1.

Discuss the following well known process models as relates to software development.

1. Agile model

The meaning of Agile is swift or versatile. **Agile process model** refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

1. V-model model

The V-model is a Software Development Life Cycle model where execution of processes happens in a sequential manner in a V-shape. It is also known as **Verification and Validation model**. The V-Model is an extension of the waterfall model and is based on the association of a testing phase for each corresponding development stage. This means that for every single phase in the development cycle, there is a directly associated testing phase. This is a highly-disciplined model and the next phase starts only after completion of the previous phase.

1. Cleanroom model

**Cleanroom software engineering** is a theory-based team-oriented process for development and certification of high-reliability software systems under statistical quality control. A principal objective of the Cleanroom process is development of software that exhibits zero failures in use. The philosophy behind cleanroom software engineering is to avoid dependence on costly defect-removal processes by writing code increments right the first time and verifying their correctness before testing. Its process model incorporates the statistical quality certification of code increments as they accumulate into a system.

1. Spiral model

**Spiral model** is one of the most important Software Development Life Cycle models, which provides support for **Risk Handling**. In its diagrammatic representation, it looks like a spiral with many loops. Here, the exact number of loops of the spiral is unknown and can vary from project to project. **Each loop of the spiral is called a Phase of the software development process.** The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks.

Q2.

1. Explain the term **robustness analysis** as relates to software development

Robustness analysis helps you to bridge the gap from Use Cases and Domain Classes, and the model-view-control (MVC) software architecture. In a nutshell, it’s a way of analyzing your use case model and identifying the first-guess set of objects for each use case. These are classified into boundary objects, entity objects, and controllers.

-**Boundary Object**. This represents the interfaces between the actors and the system

Depending upon the type of the actor, a boundary class is required to provide a user interface, external system (legacy system) interface or device interface.

-**Control Objects**. This represent the use case logic and coordinates the other classes

It separates the interface classes from business logic classes. In ASP.NET, the code-behind classes can be regarded as Control classes.

-**Entity Object**. This manages the information the system needs to provide the required functionality

Usually, entity classes are business-specific. They are information experts and encapsulate business knowledge. Most of the time, entity classes are persistent classes that can be used to generate database schema directly

**Robustness Diagram – 4 Connection Rules**

Keep in mind that both boundary objects and entity objects are nouns and that controllers are verbs. Nouns can’t talk to other nouns, but verbs can talk to either nouns or verbs. Here I listed the four basic connection rules which should always be mind:

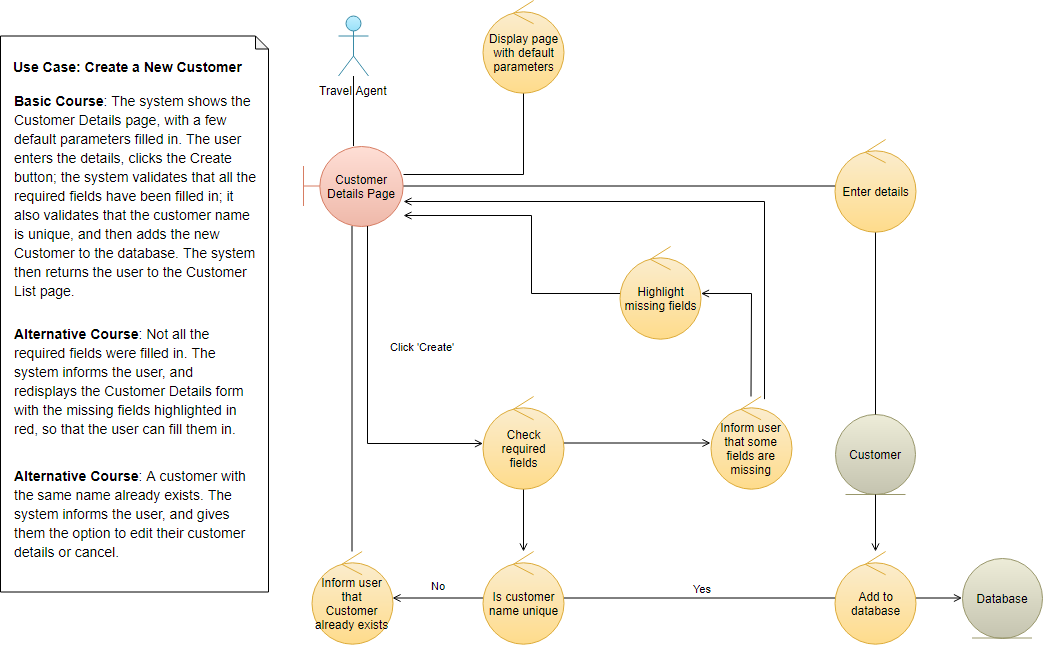
1. Actors can only talk to boundary objects.
2. Boundary objects can only talk to controllers and actors.
3. Entity objects can only talk to controllers.
4. Controllers can talk to boundary objects, entity objects, and other controllers, but not to actors.

**ROBUST ANALYSIS DIAGRAM EXAMPLE**

**Create a New Customer**

The initial use case description is put beside the diagram in the left as a label. We can use it as a basis for creating the Robustness Analysis Diagram; making it traceable with the original description.

The robustness diagram is often be used for modeling use case scenarios as well, typically represented by several sequence diagrams.



1. Describe three types of **analysis classes** as used in OOSAD and clearly cite suitable examples. Illustrate their corresponding notations for each type of analysis classes.

**Analysis classes** represent an early conceptual model for 'things in the system which have responsibilities and behavior'. They eventually evolve into classes and subsystems in the Design Model. Analysis classes may be categorized as one of the following:

* Boundary classes
* Control classes
* Entity classes

#### Boundary Class  [To top of page](https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/modguide/md_acls2.htm#Top)

A **boundary class** is a class used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as the interface).

Boundary classes model the parts of the system that depend on its surroundings. Entity classes and control classes model the parts that are independent of the system's surroundings. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes.

Boundary classes also make it easier to understand the system because they clarify the system's boundaries. They aid design by providing a good point of departure for identifying related services. For example, if you identify a printer interface early in the design, you will soon see that you must also model the formatting of printouts.

Common boundary classes include windows, communication protocols, printer interfaces, sensors, and terminals. You should model boundary classes according to what kind of boundary they represent. Communication with another system and communication with a human actor (through a user interface) have very different objectives.

A system may have several types of boundary classes:

* **User interface classes** - classes which intermediate communication with human users of the system
* **System interface classes** - classes which intermediate communication with other system
* **Device interface classes** - classes which provide the interface to devices (such as sensors), which detect external events

**Example**

* In an Automated Teller Machine, withdrawal of funds must be verified through the ATM Network, an actor (which in turn verifies the withdrawal with the bank accounting system). An object called ATM Network Interface can be identified to provide communication with the ATM Network.

#### Control Class  [To top of page](https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/modguide/md_acls2.htm#Top)

A **control class** is a class used to model control behavior specific to one or a few use cases. Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case specific behavior.

Control classes can contribute to understanding the system because they represent the dynamics of the system, handling the main tasks and control flows.

When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

Note that a control class does not handle **everything** required in a use case. Instead, it coordinates the activities of other objects that implement the functionality. The control class delegates work to the objects that have been assigned the responsibility for the functionality.

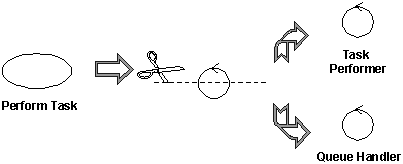
More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control objects include programs such as transaction managers, resource coordinators, and error handlers.

Control classes provide behavior that:

* Is surroundings-independent (does not change when the surroundings change),
* Defines control logic (order between events) and transactions within a use case.
* Changes little if the internal structure or behavior of the entity classes changes,
* Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
* Is not performed in the same way every time it is activated (flow of events features several states).

**Example: Managing a Queue of Tasks**

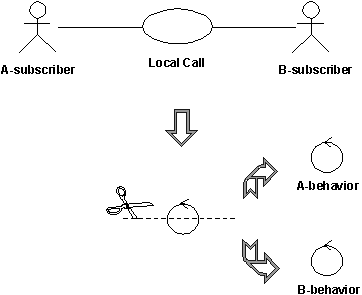
* You can identify a control class from the use case Perform Task in the Depot-Handling System. This control class handles a queue of Tasks, ensuring that Tasks are performed in the right order. It performs the next Task in the queue as soon as suitable transportation equipment is allocated. The system can therefore perform several Tasks at the same time.
* The behavior defined by the corresponding control object is easier to describe if you split it into two control classes, Task Performer and Queue Handler. A Queue Handler object will handle only the queue order and the allocation of transportation equipment. One Queue Handler object is needed for the whole queue. As soon as the system performs a Task, it will create a new Task Performer object, which will perform the Task. We thus need one Task Performer object for each Task the system performs.



The principal benefit of this split is that we have separated queue handling responsibilities (something generic to many use cases) from the specific activities of task management, which are specific to this use case. This makes the classes easier to understand and easier to adapt as the design matures. It also has benefits in balancing the load of the system, as many Task Performers can be created as necessary to handle the workload.

**Example: Call Management**

Consider the use case **Local Call**. Initially, we can identify a control class to manage the call itself.



The control class handling local phone calls in a telephone system can quickly be divided into two control classes, **A-behavior** and **B-behavior**, one for each actor involved.

In a local phone call, there are two actors: **A-subscriber** who initiates the call, and **B-subscriber** who receives the call. The **A-subscriber**lifts the receiver, hears the dial tone, and then dials a number of digits, which the system stores and analyzes. When the system has received all the digits, it sends a ringing tone to **A-subscriber**, and a ringing signal to **B-subscriber**. When **B-subscriber**answers, the tone and the signal stop, and the conversation between the subscribers can begin. The call is finished when both subscribers hang up.

Two behaviors must be controlled: What happens at A-subscriber’s place and what happens at B-subscriber’s place. For this reason, the original control object was split into two control objects, **A-behavior**and **B-behavior**.

**NB-**You do not have to divide a control class if:

* You can be reasonably sure that the behavior of the actors related to the objects of the control class will never change, or change very little.
* The behavior of an object of the control class toward one actor is very insignificant compared with its behavior toward another actor, a single object can hold all the behavior. Combining behavior in this way will have a negligible effect on changeability.

#### Entity Class  [To top of page](https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/modguide/md_acls2.htm#Top)

An **entity class** is a class used to model information and associated behavior that must be stored. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system.

An entity object is usually not specific to one use-case realization; sometimes, an entity object is not even specific to the system itself. The values of its attributes and relationships are often given by an actor. An entity object may also be needed to help perform internal system tasks. Entity objects can have behavior as complicated as that of other object stereotypes. However, unlike other objects, this behavior is strongly related to the phenomenon the entity object represents. Entity objects are independent of the environment (the actors). Entity objects represent the key concepts of the system being developed.

**Example of entity classes**

Typical examples of entity classes in a banking system are **Account** and **Customer.** In a network-handling system, examples are **Node** and **Link**.

If the phenomenon you wish to model is not used by any other class, you can model it as an attribute of an entity class, or even as a relationship between entity classes. On the other hand, if the phenomenon is used by any other class in the design model, you must model it as a class.

Entity classes provide another point of view from which to understand the system because they show the logical data structure, which can help you understand what the system is supposed to offer its users.

1. Explain the importance of robustness analysis

**1. Better project analysis**

Robust analysis involves further study of what has already been analyzed. Before any design is finalized, all available data from the analysis stage are reviewed to make them more effective and useful. Although ideas may already have been drawn out and studied for a particular project, the robust analysis stage will perform the function of the quality checker if a particular idea is good enough to be pushed through the next stage.

**2. Better design**

Since robust analysis is done before any product design is finalized, it will most certainly result to better options and better designs. With the help of reviewing what has already been analyzed in the previous stage, the design architecture can be better planned and this could yield a better result.

**3. Achieve consistency**

Robustness analysis is implemented to be able to find the best way to create and design a particular product. By testing and reviewing various scenarios on the product involved, consistency can easily be achieved. After all, robustness analysis literally translates to review of a product’s effectiveness, strength, or health. By being able to test products before the actual and final design, the end-result will be a more consistent and robust product. Through sequence diagrams, robust analysis will also provide easy traceability of the system and how it actually works.