Software Engineering Spring 2017

Final Review

*Lecture 7*

**Requirement Engineering Process:**

\*\* The processes use for RE vary depending on the application domain, the people involved, and the organization developing the process.

Common Generic Activities:

* Requirements Elicitation
  + Involves technical staff working with customers to find out about the application domain, services that system should provide, and the system’s operation constraints.
  + May involve end-users, engineers, managers, domain experts, trade unions, etc. these people are called *stakeholders*.
* Requirements Analysis
* Requirements Specification
* Requirements Validation

\*\*Spiral View of the Requirements Engineering Process\*\*

4.12 ReqEngSpiral.eps

Requirement Elicitation:

\*\* Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.

Stage includes:

* Requirements discovery
  + Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
* Requirements classification and organization
  + Groups related requirements and organizes them into coherent clusters.
* Requirements prioritization and negotiation
  + Prioritising requirements and resolving requirements conflicts.
* Requirements specification
  + Requirements are documented and input into the next round of the spiral.

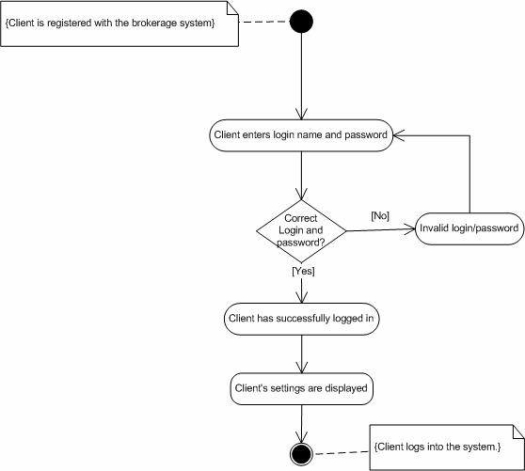
4.13 RequirementsElicitation.eps

**Interviewing:**

* Formal and Informal interviews with stakeholders are a part of most RE processes.
* In these interviews, the requirements engineering team asks questions about the system they currently use and the system to be developed. The team derives requirements from the answers to these questions.
* Types of interviews:
  + Closed interviews based on pre-determined list of questions.
  + Open interviews where various issues are explored with stakeholders.

**Drawing an Activity Diagram:**

* Activity diagrams represent the dynamic (behavioral) view of a system
* Activity diagram is used to represent the flow across use cases or to represent flow within a particular use case.
* In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes.
* Activity diagram contains activities, transitions between activities, decision points, synchronization bars, swim lanes and many more…

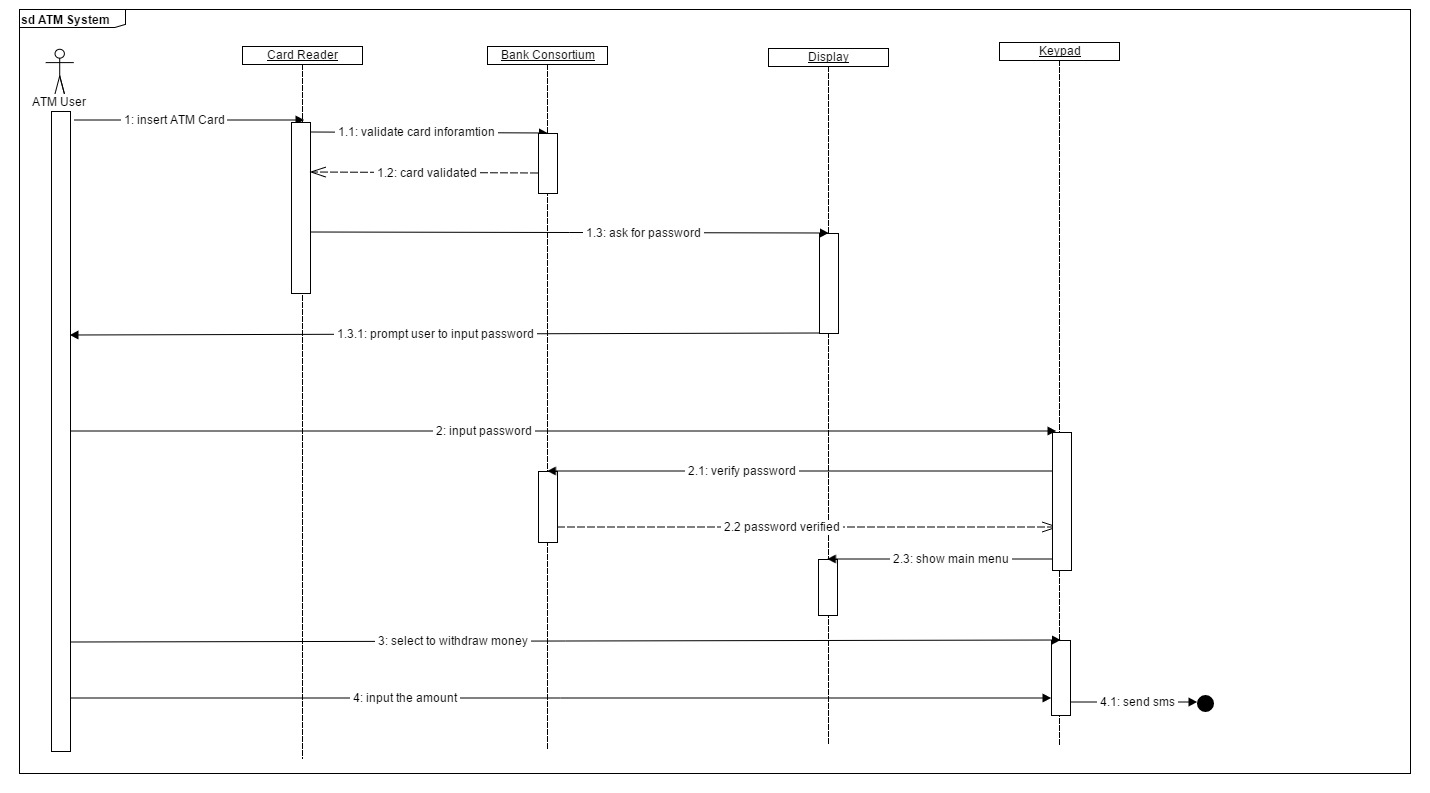


*Lecture 8*

**Sequence Diagram:**

* Sequence diagrams are part of the UML and are used to model the interactions between the objects within a system.
* A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
* The reason the sequence diagram is so useful is because it shows the interaction logic between the objects in the system in the time order that the interactions take place.
* In design and implementation process, the sequence diagram can clarify the life-time cycle of each classes.

\*\* Sequence Diagram for ATM System \*\*

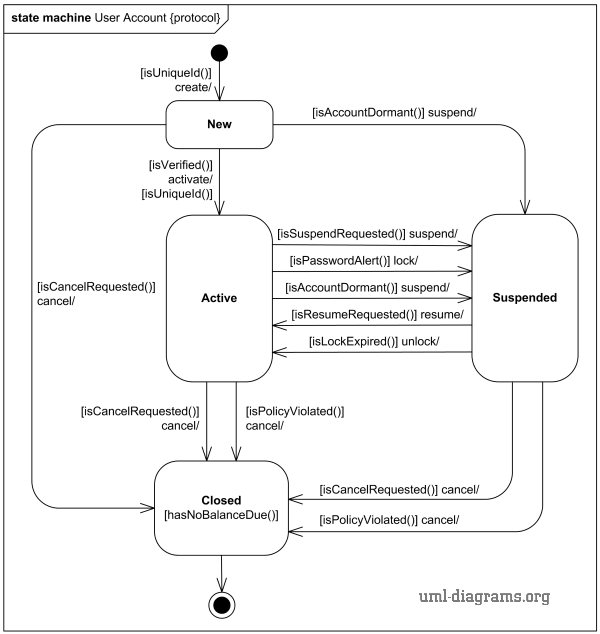


Messages:

* Synchronous Message
  + All communications from one object to another are called messages and they are represented by message arrows in sequence diagram.
  + Represented by a line and a solid arrow
  + A message defines a particular communication between Lifelines of an interaction.
  + Synchronous Message is used when a sending individual suspends execution after sending the message. That is what normally occurs when an operation is called.
* Asynchronous Message
  + Represented by a line and an arrow
  + The asynchronous message arrow is used when a sending individual continues execution after sending message.
  + The name is bracket is the value which need to be passed with the message.
* Synchronous Message Return
  + Represented by a dashed line and an arrow
  + The synchronous return message arrow shows the return of control from a synchronous message or to create a new a new entity.

*Lecture 9*

**State Machine Diagram:**



\*\* State Diagram of Microwave Oven \*\*

5.16 MWOvenStateDiag.eps

Overview of architectural design:

* Architectural design is about dividing the system into subsystems with minimal dependencies between them.

Architectural Views:

* Logic view
  + Shows the key abstractions in the system as objects or object classes.

(e.g. class diagram)

* Development view
  + Shows how the software is decomposed for development.

(e.g. Implementation diagram)

* Process view
  + Shows how, at run-time, the system is composed of interacting processes.

(e.g. sequence diagram)

* Physical view
  + Shows the system hardware and how software components are distributed across the processors in the system. (e.g. Deployment diagram)

*Lecture 10*

**Coupling:**

* In software engineering, coupling is the degree of interdependence between a software module.
* Tight coupling is when a group of classes are highly dependent on one another.
  + This scenario arises when a class assumes too many responsibilities, or when one concern is spread over many classes rather than having its own class.
* A loosely-coupled class can be consumed and tested independently of other (concrete) classes.
  + Interfaces are a powerful tool to use for decoupling. Classes can communicate through interfaces rather than other concrete classes.

**Cohesion:**

* The attraction that a group of similar objects share. In Java, cohesion is the relationship between members within a single class to perform a specific functionality
* The cohesiveness of a class can be measured from highly cohesive to low cohesion.

Levels of Cohesions:

* **Coincidental cohesion (worst)**
  + Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a “Utilities” class).
* **Logical cohesion**
  + Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing even though they are different by nature (e.g. grouping all mouse and keyboard input handling routines).
* **Temporal cohesion**
  + Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).
* **Procedural cohesion**
  + Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).
* **Communicational/informational cohesion**
  + Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).
* **Sequential cohesion**
  + Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).
* **Functional cohesion (best)**
  + Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module

**“SOLID” Principle:**

* In computer programming, SOLID (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) is an abbreviation introduced by Robert C. Martinin the early 2000s that stands for five basic principles of object-oriented programming.
* The intention is that these principles, when applied together, will make it more likely that a programmer will create a system that is easy to maintain and extend over time.
* The principles of SOLID are guidelines that can be applied while working on software to remove rigidly code by providing a framework through which the programmer may refactor the code until it is both legible and extensible.
* It is part of an overall strategy of agile and Adaptive Software Development.

*Lecture 11*

**High level Architecture Design Pattern:**

???

**The Model-View-Controller (MVC) Patter:**

* It is a software design pattern for implementing user interface on computers.
* It divides a given software application into three interconnected parts
  + The Model component manages the system data and associated operations on that data.
  + The View component defines and manages how the data is presented to the user.
  + The Controller component manages user interaction and passes these interactions to the View and the Model.

\*\*\* Advantages \*\*\*

* **Faster development process**
* **Ability to provide multiple views**
* **Modification does not affect the entire model**

**Low-Level Design Pattern:**

* What is design pattern
  + A design pattern is a way of reusing abstract knowledge about a problem and its solution.

**Strategy Pattern:**

* The strategy pattern is a software design pattern that enables an algorithm's behavior to be selected at runtime.
  + Defines a family of algorithms
  + Encapsulates each one
  + Make them interchangeable.
* Strategy pattern lets the algorithm vary independently from clients that use it.
* Avoid duplicated code.
* Eliminated technique that one class to affect others.
  + A super class change code shouldn’t break code in a subclass and vice versa.

**Factory Pattern:**

* In Object-Oriented programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.
* When a method returns one of several possible classes that share a common super class
  + Create a new enemy in a game
  + Random number generator picks a number assigned to a specific enemy
  + The factory returns the enemy associated with that number
* The class is chosen at run time.

*Lecture 12*

**Test Case:**

* A test case contains:
  + A sequence of Steps describing actions to be performed.
  + Test data to be used.
  + An expected response for each action performed.
* Test case are written based on Business and Functional/Technical requirements, use cases and Technical design documents.
* Test relationship between requirements and test case should be 1:1 or 1:N or N:N relationship.
* The level of detail specified in test case will vary depending on Organization, Projects.
* Construction of Test Case also help in:
  + Requirement validation
  + make tester to think through different possible Positive and Negative scenarios.
  + tester will verify the application is working as expected.
  + Number of test cases to be created depends on the size, complexity and type of testing being performed.

Equivelance Partitioning:

\*\* Equivalence partitioning or equivalence class partitioning(ECP) is a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived.

Example: Testing grading function in blackboard.

**Test Strategy:**

* Black Box:
  + Usual process
    - Test planning- may start when the spec is circulated
    - Acceptance/qualification testing – check whether the program is stable enough to be tested
    - Function and system test – check it again spec
    - Beta testing – get user feedback
    - Release testing
      * Check all things that will go to the customer/manufacture
    - Final acceptance testing by the customer
* White Box:
  + White-Box/Glass-Box/Structural testing
    - Logic-driven: examine internal program structure
  + Technique
    - Basis path testing, data flow testing
    - Part of coding stage
  + Path Testing
    - Path: a sequence of operations that runs from start to an exit point
    - No credit for repeated testing of the same path using different data
    - Coverage criteria
* Hybrid / Grey Box:
  + What is Hybrid/Gray-box testing?
    - Combine behavioral/structural strategies
    - Example: derive tests from both specification and code coverage
    - Unit and low-level components are often tested by the structural strategies
    - Big components and system testing is dominated by behavioral strategies
    - Hybrid strategy area useful at all levels
* Regression:
  + What is this term “Regression”?
    - Regression means errors that occurs due to some action or activities in a system.
    - In IT world a "regression" means the return of a bug
  + Regression testing is testing done to check that a system update does not re-introduce errors that have been corrected earlier.

**Debugging Strategy:**

* Brute Force:
  + Most commonly used and least efficient method
  + Used when all else fails
  + Involves the use of memory dumps, run-time traces, and output statements
  + Leads many times to wasted effort and time
* Backtracking:
  + Can be used successfully in small programs
  + The method starts at the location where a symptom has been uncovered
  + The source code is then traced backward (manually) until the location of the cause is found
  + In large programs, the number of potential backward paths may become unmanageably large
* Cause Elimination:
  + Involves the use of induction or deduction and introduces the concept of binary partitioning
    - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
    - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
  + Data related to the error occurrence are organized to isolate potential causes
  + A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
  + Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
  + If initial tests indicate that a particular cause hypothesis shows premise, data are refined in an attempt to isolate the bug.

**That’s it! Good Luck!!**