**COMP-3700 MIDTERM I STUDY GUIDE**

**Note:** The first midterm will mostly be on class models, state models, and interaction models, so it might be best to focus primarily on lectures 2, 3, and 4.

**Table of Contents:**

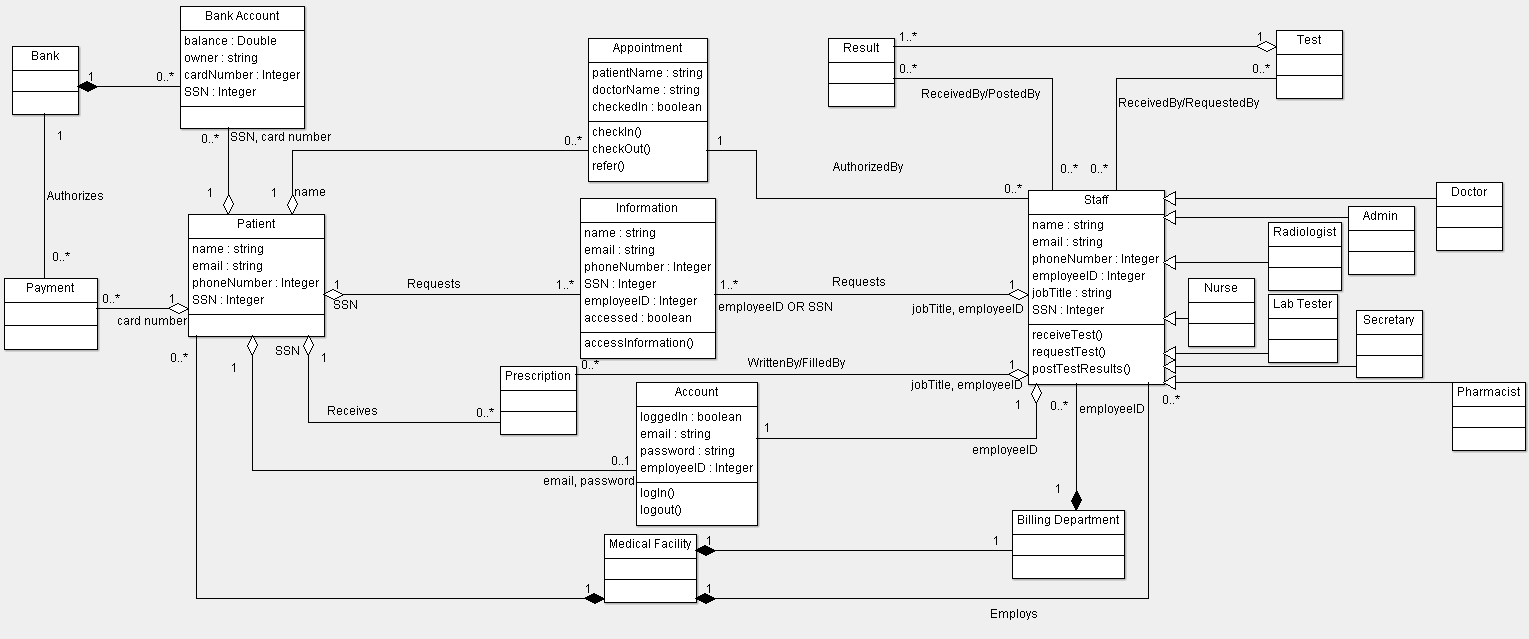
* Lecture 1 spans pages 1 – 6
* Lecture 2 spans pages 7 – 10, with a class diagram example included on page 11
* Lecture 3 spans pages 12 – 15, with a state diagram example included on page 15
* Lecture 4 spans pages 16 – 18, with a use case diagram example on page 19, a sequence diagram example on page 20, and an activity diagram example on page 21 (one with swimlanes and one without)
* Lecture 5 spans pages 22 – 25
* The Midterm I questions are on page 26.
* A sample of a previous Midterm I are on pages 27 – 30.

**LECTURE 1 (INTRODUCTION):**

* Computational methods and models give us the ability to solve problems and design systems that no one of us would be capable of tackling alone
  + Computational thinking is reformulating a seemingly difficult problem into one we know how to solve by reduction, embedding, transformation, or simulation
    - It is also using abstraction and decomposition when attacking a large complex task or designing a large complex system, separation of concerns, and choosing an appropriate representation for a problem or modeling the relevant aspects of a problem to make it tractable
  + It is also using invariants to describe a system’s behavior succinctly and declaratively
  + We can safely use, modify, and influence a large complex system without understanding its every detail
  + Computational thinking thus has the following characteristics:
    - Conceptualizing, not programming. Computer science is not computer programming
    - Thinking like a computer scientist means more than being able to program a computer
    - Requires thinking at multiple levels of abstraction
    - Fundamental, not rote skill. A fundamental skill is something every human being must know to function in modern society. Rote means a mechanical routine
    - A way that humans, not computers, think. Computational thinking is a way humans solve problems; it is not trying to get humans to think like computers. Computers are dull and boring; humans are clever and imaginative
    - It complements and combines mathematical and engineering thinking
    - Computer science inherently draws on mathematical thinking, given that, like all sciences, its formal foundations rest on mathematics
    - Draws on engineering thinking, given that we build systems that interact with the real world
    - The constraints of the underlying computing device force computer scientists to think computationally, not just mathematically
    - Being free to build virtual worlds enables us to engineer systems beyond the physical world
    - Ideas, not artifacts. It’s not just the software and hardware artifacts we produce that will be physically present everywhere and touch our lives all the time, it will be the computational concepts we use to approach and solve problems, manage our daily lives, and communicate and interact with other people
    - Intellectually challenging and engaging scientific problems remain to be understood and solved. The problem domain and solution domain are limited only by our own curiosity and creativity
* Characteristics of Software Design: Design is conscious, design keeps human concerns in the center, design is a conversation with materials, design is creative, design is communication, design has social consequences, & design is a social activity
* **Why Software Design:**
  + Writing a program is easy; developing a software product is hard
  + High-quality software products are robust, efficient and effective. They are also easy to understand, modify and compose with other high-quality software products
  + Software engineers are skilled professionals who follow “best practices”
* As computers become cheaper, smaller, and more powerful, their spread through our technological society becomes more pervasive
* **Why study software design?**
  + In an article in the January/February 1997 issue of I.E.E.E. Software, authors cite staggering cost estimates of software development failures at $81 billion for 1995, and $100 billion for 1996.
  + Several highly visible failures:
    - The cancellation of IBM's $8 billion contract with the FAA
    - The DOD cancellation of a $2 billion contract with IBM to modernize its information systems
    - The failure of the software for delivering real time sports data at the 1996 Olympics
    - The one and one-half year delay in the United Airlines automated baggage handling system at the new Denver airport at a cost of $1.1 million per day, and the list could go on.
  + A reading of Peter Neumann's book, Computer Related Risks, reveal deaths which resulted from radiation overdoses from a computer-based radiation therapy system in the mid-1980s
  + Aids to precision and cross-checking are essential, and this is precisely the objective of software design and modeling.
* **Modeling as a Design Technique:**
  + A model is an abstraction of something for the purpose of understanding it before building it
  + To build complex systems:
    - Abstract different views of the system
    - Build models using precise notations
    - Verify that the models satisfy requirements
    - Gradually add detail to transform models into implementation
* **Why model?**
  + Testing a physical entity before building it
  + Communication with customers
  + Visualization
  + Reduction of complexity
* **Customer Myths:**
  + Myth – A general statement of objectives is sufficient, we will fill in the details later
  + Reality – Poor up-front definition is the major cause of failed software efforts
  + Myth – Project requirements continually change, but change can easily be accommodated because software is flexible
  + Reality – It is true that software requirements do change, but the impact of change varies with the time the change is introduced
* **Practitioner Myth:**
  + Myth – The only deliverable for a successful project is the working program
  + Reality – A working program is only one part of the software configuration. Documentation forms the foundation for a successful development and provides guidance for the software maintenance task
* Building the system: Determine the requirements, create a system design, design individual programs, test the programs in pieces, test the programs together, & deliver the system
* Analysis emphasizes the investigation of the domain and requirements
* Design emphasizes a conceptual solution that fulfills the requirements
* **What is Object-Orientation (OO)?**
  + OO means that we organize software as a collection of discrete objects that incorporate both data structure and behavior
  + Characteristics:
    - Identity – Discrete, distinguishable entities, called objects
    - Classification – Objects with same data structure and behavior are grouped into a class
    - Inheritance – Sharing of attributes and operations among classes based on a hierarchical relationship
    - Polymorphism – Same operation may behave differently for different classes
  + OO Principles:
    - Information Hiding – The principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Protecting a design decision involves providing a stable interface which shields the remainder of the program from the implementation
    - Encapsulation – Separates the external aspects of an object, that are accessible to other objects, from the implementation details, that are hidden from other objects
      * Abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data
    - Abstraction – Refers to focusing on essential aspects of an application while ignoring details
  + OO Analysis: A method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain
  + OO Design: A method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design
  + OO Programming: A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships
  + OO Methodology:
    - System Conception means conceiving an application and tentative requirements
    - Analysis restates requirements by constructing models
    - Domain Model – A description of the real-world objects (e.g., stock, bond, trade, commission in a stockbroker application)
    - Application Model – Parts of the application system that are visible to the user (e.g., object that control the execution and present the results)
    - System Design involves devising a high-level strategy – the system architecture and interaction design
    - Class Design adds details to the analysis model in accordance with the system design strategy
      * Interface design, algorithm, & data structure design
    - Prototyping/Implementation involves translating class design onto a programming language
* **Three Models:**
  + The class model describes the static structure of the objects and their relationships (Class Diagram)
    - The class model defines the context for software development
    - The class diagram is a graph whose nodes are classes and whose arcs are relationships
  + The state model describes aspects that change over time
    - Specifies and implements control with state diagrams
  + The interaction model describes how objects in a system cooperate to achieve broader results
    - Use cases focus on functionality from the perspective of the users
    - The sequence and collaboration diagrams show the objects that interact and the time sequence of their interactions
    - The activity diagram elaborates control and workflow that depict processing steps
* **What is the UML?**
  + UML stands for Unified Modeling Language
  + The UML combines the best of the best from
    - Data Modeling concepts (Entity Relationship Diagrams)
    - Business Modeling (work flow)
    - Object Modeling
    - Component Modeling
  + The UML is the standard language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system
  + It can be used with all processes, across different implementation technologies
* **Design Paradigms:**
  + Function-Driven Paradigms:
    - Process models show the overall process or the subprocesses that are supported by the system
  + Data-Driven Paradigms:
    - Used to describe the logical structure of data processed by the system
    - Entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes
    - Widely used in database design
    - No specific notation provided in the UML but objects and associations can be used
* **Terminology:**
  + Operation – procedure or transformation that an object performs or is subject to
  + Method – An implementation of an operation by a specific class
  + Features – Attributes or operations
  + Development – The software life cycle; analysis, design, and implementation
  + System Architecture – A high-level strategy

**LECTURE 2 (CLASS MODELING):**

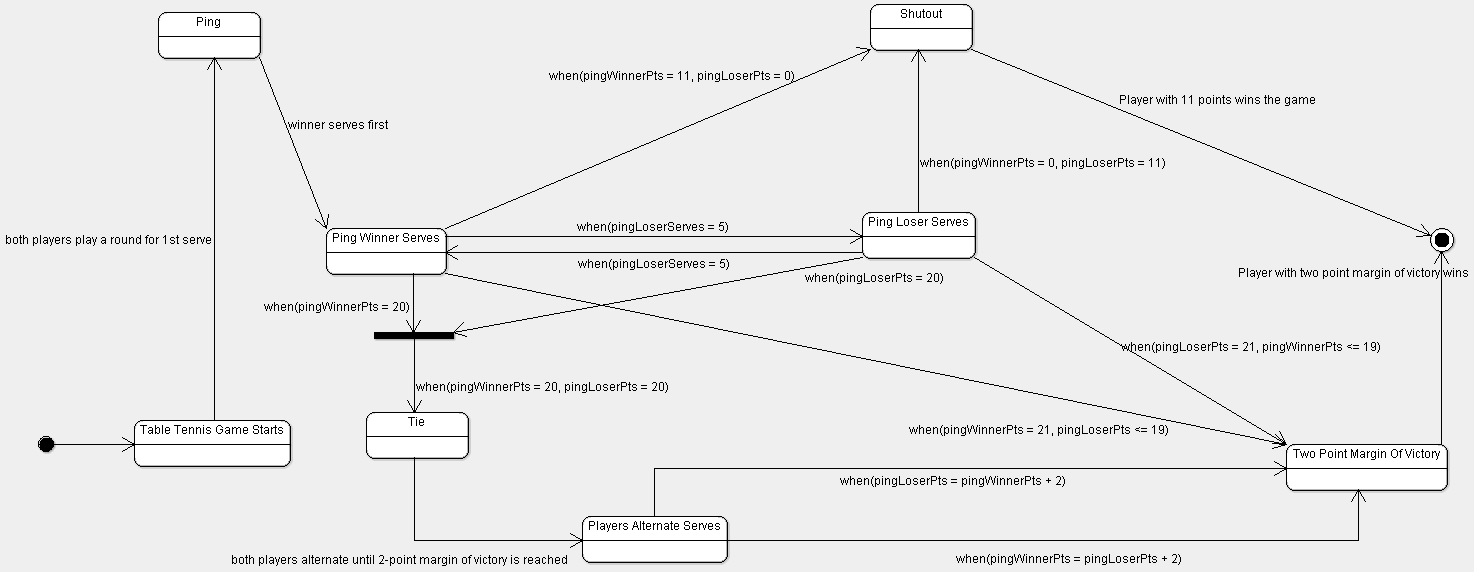
* **Domain Modeling:**
  + Partitions and illustrates the important domain concepts; a classic object-oriented analysis activity
  + **IMPORTANT:** Not software objects, but a “visual dictionary” of domain concepts
  + A domain model does NOT represent software objects
    - It is a model of domain concepts, not of software objects
      * A “visual dictionary” of important words in the domain
    - Uses UML static structure diagram notation
  + **How to Make a Domain Model:**
    - List the candidate conceptual classes using the Conceptual Class category list or linguistic analysis
    - Draw them in a domain model
    - Add the associations necessary to record relations
    - Add the necessary attributes to fulfill the information requirements
* **Objects & Classes:**
  + Object is a concept, abstraction, or thing that has meaning for an application
  + Objects appear as proper nouns in problem descriptions (e.g., Joe Smith, IBM, process 7648)
  + A class describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics
* **Finding Domain Concepts:**
  + Linguistic Analysis: Identify the nouns and noun phrases in textual descriptions; care must be applied with this method: a mechanical noun-to-class mapping isn’t possible, and words in natural languages are ambiguous
  + Specification: Design a library catalog system. The system must support the registration of patrons, checking books in and out patrons, adding and removing of books, and determining which patron has a book
* **Approaches:**
  + Abbott & Booch suggest:
    - Use nouns, pronouns, noun phrases to identify objects and classes
    - Singular -> object, plural -> class
    - Not all nouns are really going to relate to objects
  + Coad & Yourdon suggest:
    - Identify individual or group “things” in the system or problem
  + Ross suggests:
    - People, places, things, organizations, concepts, events
  + Danger signs: class name is a verb, is described as performing something
* **Attributes:**
  + An attribute is a named property of a class that describes a value held by each object of the class
  + Object is to class as value is to attribute
  + One can find attributes by looking for adjectives or by abstracting typical values
  + Show only “simple” relatively primitive types as attributes
  + Connections to other concepts are to be represented as associations, not attributes
* **Operations (During the design phase – not domain modeling):**
  + An operation is a function or procedure that may be applied to or by objects in a class (e.g., hire, fire, payDivident for company)
* **Links and Association Concepts:**
  + A link is a physical or conceptual connection among objects (e.g., Joe works-For IBM)
  + An association is a description of a group of links with common structure and semantics
* **Association End Names:**
  + You can assign association ends not only multiplicity, but also a name
  + Association end names are necessary for associations between two objects of the same class
* **Ordering, Bags, & Sequences:**
  + Sometimes objects on a “many” association end have explicit order
  + A sequence is an ordered collection of elements with duplicates allowed; a bag is an unordered collection of elements with duplicates allowed
* **Association Classes:**
  + An association class is an association that is also a class
  + Many to many associations provide a compelling rationale for association classes
  + Qualified Association: An association in which an attribute called the qualifier disambiguates the objects for a “may” association end
* **Generalization & Inheritance:**
  + Generalization is the relationship between a class (the superclass) and one or more variations of the class
    - Organizes classes by their similarities and differences, structuring the description of objects
    - Use of Generalization: Polymorphism, structuring the description of objects – creating a taxonomy, and reuse of code and data
* **Navigation of Class Models:**
  + Navigation enables exercising a model to uncover hidden flaws and omission
* **Terminology:**
  + Value – A piece of data
  + Class Diagram – Provide a graphic notation for modeling classes and their relationships, thereby describing possible objects
  + Object Diagram – Shows individual objects and their relationships
  + Signature – The number and types of arguments and the type of result value
  + Feature – Generic word for either an attribute or operation
  + Direction – Indicates whether an argument is an input (in), output (out), or an input argument that can be modified (inout)
  + Link – Physical or conceptual connection among objects
  + Association – A description of a group of links with common structure and common semantics
  + Reference – An attribute in one object that refers to another object
  + Multiplicity – Specifies the number of instances of one class that may relate to a single instance of an associated class; consider a constraint on the size of a collection
  + Cardinality – The count of elements that are in a collection
  + Association End – Can contain multiplicity or names or both
  + Set – Objects on a “many” association end; they have no order
  + Bag – A collection of elements with duplicates allowed
  + Sequence – An ordered collection of elements with duplicates allowed
  + Association Class – An association that is also a class
  + Qualified Association – An association in which an attribute called the qualifier disambiguates the objects for a “many” association end
  + Generalization – The relationship between a class (the superclass) and one or more variations of the class (the subclasses); each subclass is said to inherit the features of its superclass (also called the descendant and ancestor, respectively)
  + Generalization Set Name – An enumerated attribute that indicates which aspect of an object is being abstracted by a generalization
  + Override – The means by which a subclass defines a feature with the same name as its superclass feature
  + Null – Special value that denotes that an attribute value is unknown or not applicable
  + Enumeration – A data type that has a finite set of values
  + Multiplicity for an Attribute – Specifies the number of possible values for each instantiation of an attribute
  + Scope – Indicates if a feature applies to an object or a class
  + Extent – The set of objects for a class
  + Visibility – The ability of a method to reference a feature from another class; has the possible values of public, protected, private, and package
  + Public – Any method can freely access it
  + Protected – Only methods of the containing class and its descendants via inheritance can access it
  + Private – Only methods of the containing class can access it
  + Package – Methods of classes defined in the same package as the target class can access it
  + N-ary Associations – Associations among three or more classes
  + Propagation/Triggering – Automatic application of an operation to a network of objects when the operation is applied to some starting object
  + Composition: A has-a relationship symbolized by a black diamond – the first class has a second class as an object. Emulates real-world whole-part relationships (i.e., an engine is a part of a car). When the container is destroyed, the contents are also destroyed (e.g., a university and its departments)
  + Generalization/Inheritance: An is-a relationship symbolized by an arrow – the second class is an object of the first class.
  + Aggregation: A has-a relationship symbolized by an empty diamond – the first class has a second class as an object. Emulates real-world whole-part relationships, BUT the second object can belong to many different objects. When the container is destroyed, the contents are USUALLY not destroyed. Has transitivity (if A is part of B, and B is part of C, then A is part of C) and antisymmetry (if A is part of B, B is NOT part of A).
  + Abstract Class – A class that has no direct instances but whose descendant classes have direct instances
  + Concrete Class – A class that is instantiable; that is, it can have direct instances
  + Abstract Operation – A defined signature for an operation without a corresponding method
  + Multiple Inheritance – Permits a class to have more than one superclass and to inherit features from all parents
  + Delegation – An implementation mechanism by which an object forwards an operation to another object for execution
  + Metadata – Data that describes other data
  + Reification – The promotion of something that is not an object into an object
  + Constraint – A Boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets
  + Derived Element – A function of one or more elements, which in turn may be derived; considered queries with no argument except the target object
  + Query – An operation that merely computes a functional value without modifying any objects
  + Package – A group of elements (classes, associations, generalizations, and lesser packages) with a common theme

**Class Diagram Example:**

**Note:** The words next to association ends are meant to be considered qualifiers. 0..\* or 1..\* is also used instead of \* to show “many” multiplicity.

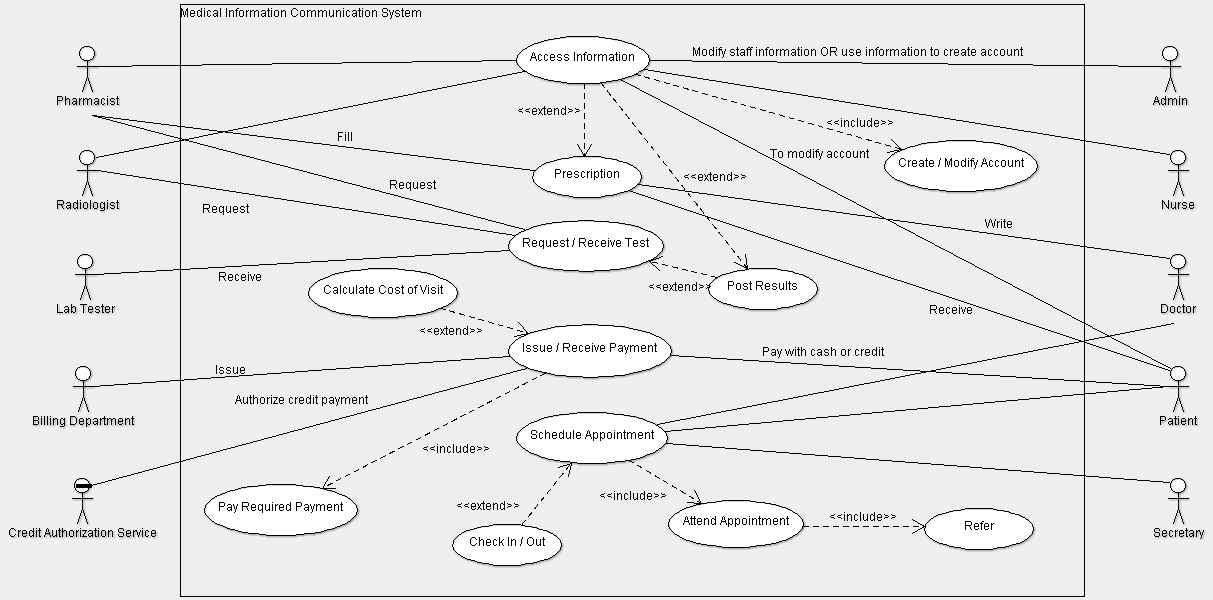
**LECTURE 3 (STATE MODELING):**

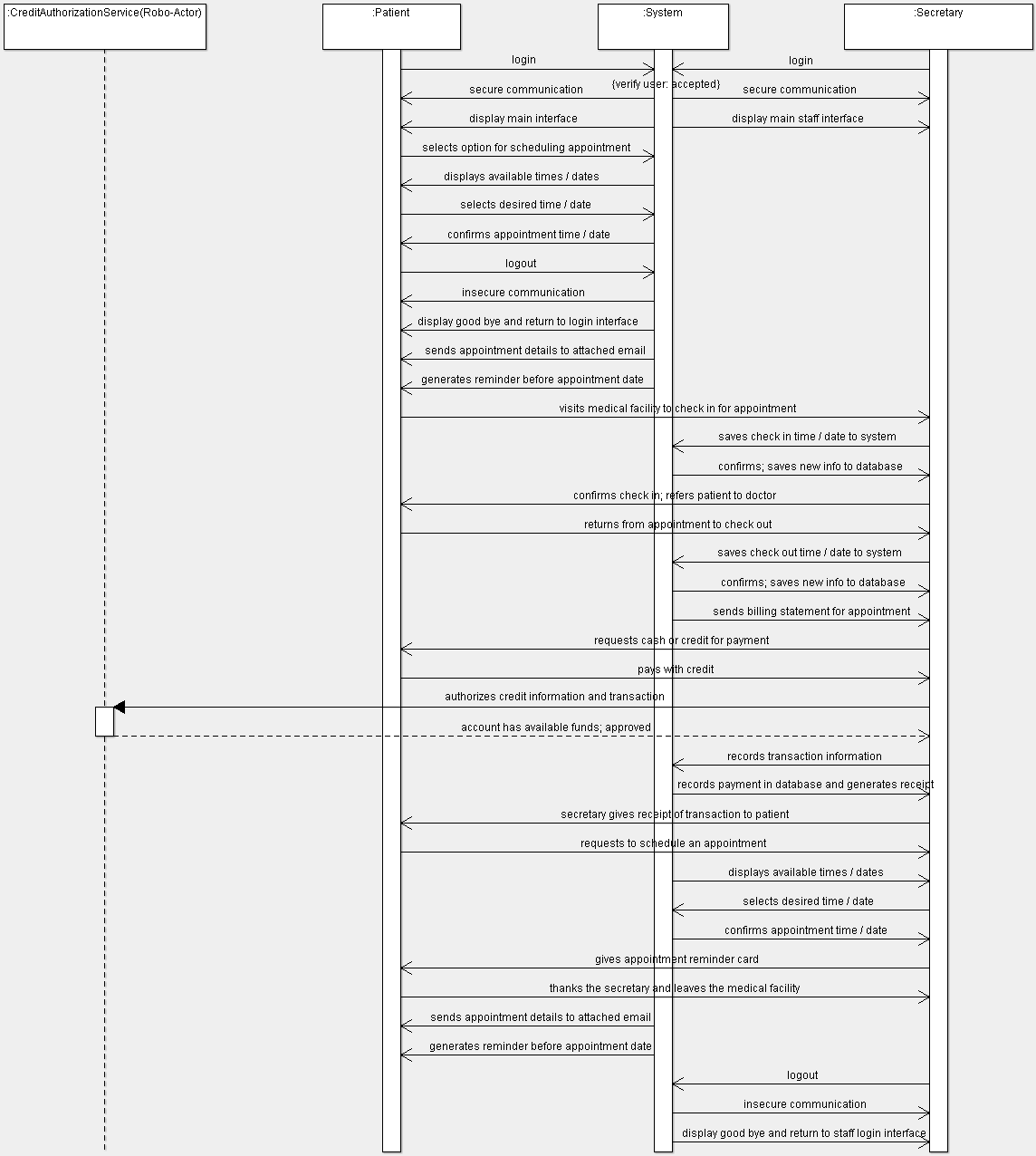
* State models specify the behavior of the system in response to external and internal events
  + They show the system’s responses to stimuli so are often used for modeling real-time systems
* State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another
* Statecharts are an integral part of the UML
* **State Model:**
  + The state model describes the sequence of operations that occur in response to external stimuli
  + Consists of multiple state diagrams, one for each class with temporal behavior
  + Events represents the stimuli
  + States represent the values of objects
* **Events:**
  + An event is an occurrence at a point in time
  + Examples: user depresses left button, flight 123 departs from Chicago
  + Events include error conditions as well as normal occurrences (e.g., motor jammed, transaction aborted)
  + Types of events: Signal, change, time
  + A signal is an explicit one way transmission of information from one object to another
    - It is different than a subroutine call
  + A signal event is the event of sending or receiving a signal
  + A change event is an event that is caused by the satisfaction of a Boolean expression represented by a predicate
    - Expression is continuously tested – whenever the expression changes from false to true the event happens
    - Examples: when (room temperature < heating set point), when (battery power < lower limit)
  + A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval
    - UML notation – when (date = January 16, 2019), after (10 seconds)
* **States:**
  + A state is an abstraction of the values and links of an object
  + Sets of values and links are grouped into a state according to the gross behavior of an object
  + Example: A stack object can be in one of the following states: empty, non-empty, and full
* **Transitions & Conditions:**
  + A transition is a relation between two states
  + Transition depicts an instantaneous change from one state to another
  + Example: When a called phone is answered, the phone line transitions from Ringing state to the Connected state
  + The transition is said to fire upon the change from the source state to the target state
  + A guard condition is a Boolean condition that must be true for a transition to occur
* **State Diagrams:**
  + A state diagram is a graph whose nodes are states and whose arcs are transitions between states
  + A full description of an object must specify what the object does in response to events
* **Completeness & Consistency:**
  + States & Transitions:
    - Completeness requires that in each basic state, for all possible events, there must be a transition defined
    - Completeness requires that each state is targeted by (at least one) transition. Note that initial states have an incoming transition from the initial pseudo-state
    - Consistency of the specification requires that in each state, only a single transition is triggered by a given event
  + Guards:
    - Completeness requires that in each basic state, considering also inherited transitions, guards of transitions triggered by the same event form a tautology
    - Consistency is checked by the following criterion: if there are two or more transitions that are originating from the same state and triggered by the same event, then their guards could not be true at the same time
* **Activities:**
  + An activity is the actual behavior invoked or executed in response to an event
  + Do-Activities are those activities that continue for an extended time
* **Nested State Diagrams:**
  + Problems with Flat State Diagrams:
    - Impractical for large problems
    - Consider n independent Boolean attributes – the flat state diagram would require 2n states
    - Solution: Partition the state into n independent state diagrams – 2n states are required
  + Nested States: Structure states more deeply than just replacing a state with a submachine
    - States can be nested to arbitrary depth
    - A nested state receives the outgoing transitions of its composite state
* Expanding States: One way to organize is to have a high-level diagram with subdiagrams expanding certain states; like macro substitution
* **Concurrency:**
  + The state model implicitly supports concurrency
  + Objects are autonomous entities that can act and change state independent of another
  + Two types:
    - Aggregation Concurrency:
      * A state diagram for an assembly is a collection of state diagrams, one for each part
      * The aggregate state is one state from each diagram
      * Transition for one object may depend on the state of another
    - Concurrency within an Object:
      * Based on partitioning objects into subsets of attributes, each one of which has its own state diagrams
      * Dashed arrow indicates concurrently executing state machines
      * A single event may cause transitions in more than one state machine
      * A transition that forks indicates splitting of control into two concurrent parts
* **Terminology**
  + Event – An occurrence at a point in time
  + Signal – An explicit one-way transmission of information from one object to another
  + Signal Event – The event of sending or receiving a signal
  + Change Event – An event that is caused by the satisfaction of a Boolean expression
  + Time Event – An event caused by the occurrence of an absolute time or the elapse of a time interval
  + State – An abstraction of the values and links of an object
  + Transition – An instantaneous change from one state to another; the transition is said to fire upon the change from the source state to the target state
  + Guard Condition – A Boolean expression that must be true for a transition to occur
  + State Diagram – A graph whose nodes are states and whose directed arcs are transitions between states
  + State Model – Consists of multiple state diagrams, one state diagram for each class with important temporal behavior
  + Effect – A reference to a behavior that is executed in response to an event
  + Activity – The actual behavior that can be invoked by any number of effects
  + Do-Activity – An activity that continues for an extended time
  + Completion Transitions – An unlabeled transition that is triggered by the completion of activity in the source state
  + Race Condition – If an object can receive signals from more than one object, the order in which concurrent signals are received may affect the final state
  + Submachine – A state diagram that may be invoked as part of another state diagram
  + Composite State – Labels the outer contour that entirely encloses the nested states
  + Substates – When a transition fires into two concurrent subdiagrams; the two concurrent substates become active and execute independently

**State Diagram Example:**

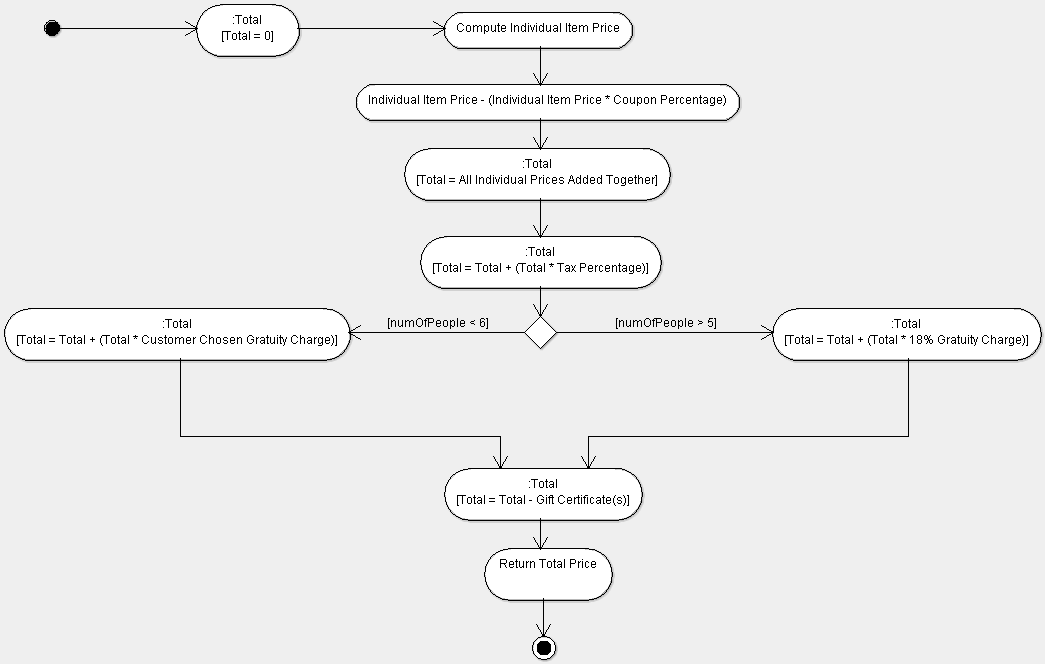
**LECTURE 4 (INTERACTION MODELING):**

* **Interaction Modeling:**
  + Describes how objects interact to produce useful results – holistic view of behavior across objects
  + Interactions can be modeled at different levels of abstraction
    - Use cases describe how a system interacts with outside actors
    - Sequence diagrams describe show the messages exchanged among a set of objects over time
    - Activity diagrams show the flow of control among the steps of a computation – can show both control and data flow
* **Use Case Model - Writing Requirements in Context:**
  + A use case tells a story of actors using a system
    - A use-case is a sequence of actions a system performs that yields an observable result of value to a actor
  + One artifact to express (especially) functional requirements
  + Emphasizes thinking about the valuable objectives-oriented viewpoint of the users
* **Identifying Use Cases:**
  + Major distinct, complete, end-to-end processes of using a system
  + Not usually one step, but a complete story
  + Examples: Rent DVDs, return DVDs, pay fines, etc
* **Use Case Diagram:**
  + A way to conceive and illustrate the use cases
  + Usually created during the initial use case analysis
  + An actor is a direct external user of a system
  + A use case is a coherent piece of functionality that a system can provide by interacting with actors
* **Essential vs. Concrete Use Cases:**
  + Essential use cases defer the details of the UI, and focus on the intentions of the actors, and responsibilities of the system; concrete (AKA real) do not
  + Essential: “The AccountHolder identifies themselves to the ATM”
  + Real: “The AccountHolder inserts their card in the reader. Window A is displayed. They enter their PIN on the numeric keypad, …”
  + As we refine analysis models, we are more inclined to move from essential to concrete use case descriptions
* **Relating Use Cases:**
  + When creating the use case diagram, it can be useful (in terms of comprehension and simplification) to:
    - Factor out shared sub-processes; use the <<includes>> relationship
    - Show precedence order; use the <<extends>> relationship
* **Guidelines for Use Case Models:**
  + First determine the system boundary
  + Ensure that actors are focused
  + Each use case must provide value to users
  + Relate use cases and actors
  + Remember use cases are informal
  + Use cases can be structured
* **Sequence Models:**
  + The sequence model elaborates the themes of use cases
  + Two types of sequence models:
    - A scenario is a sequence of events that occurs during one execution of a system, such as for a use case
    - An interaction diagram shows the participants in an interaction and the sequences of messages among them
      * Sequence diagram uses a fence format. Collaboration diagrams use the graph format
* **Sequence Diagrams:**
  + Each use case requires one or more sequence diagrams to describe its behavior
  + Notation:
    - Object lifetimes
    - Illustrating reply and returns
    - Loops
    - Conditional messages
    - Alternative
    - Iteration over a collection
  + Guidelines:
    - Prepare at least one scenario for one use case
    - Abstract the scenarios into sequence diagrams
    - Divide complex interactions
    - Prepare a sequence diagram for each error condition
* **Activity Models:**
  + Activity diagrams show the sequence of steps that make up a complex process
  + Most useful during the early stages of designing algorithms and workflows
  + Can show both sequential and concurrent flow of control
  + Activities: The steps of an activity diagram are operation, specifically activities from the state model
  + Swimlanes:
    - It is often useful to know which organization is responsible for an activity
    - Lines across swimlane boundaries indicate interactions among different organizations
  + Objectflows:
    - It is often helpful to see relations between an operation and the objects that are its arguments
    - Activity diagram can show inputs to or outputs from the activities
* **Terminology**
  + Actor – A direct external user of a system (i.e., an object or set of objects that communicates directly with the system but that is not part of the system)
  + Use Case – A coherent piece of functionality that a system can provide by interacting with actors
  + Scenario – A sequence of events that occurs during one execution of a system, such as for a use case
  + Sequence Diagram – Shows the participants in an interaction and the sequence of messages among them
  + Lifeline – A vertical line that represents an actor or the system and the entire period during which the object exists (i.e., the lifetime of the object)
  + Activity Diagram – Shows the sequence of steps that make up a complex process, such as an algorithm or workflow
  + Activity Token – Can be placed on an activity symbol to indicate that it is executing
  + Include – Relationship that incorporates one use case within the behavior sequence of another use case; describes behavior that would otherwise have to be described repeatedly, much like a subroutine
  + Extend – Adds incremental behavior to a use case; the extension adds itself to the base and represents the frequent situation in which some initial capability is defined, and later features are added modularly
  + Generalization – Shows specific variations on a general use case, analogous to generalization among classes
  + Activation/Focus of Control – The period of time for an object’s execution, shown as a thin rectangle within sequence diagrams
  + Swimlane – When an activity diagram is divided into columns and lines, this is each column

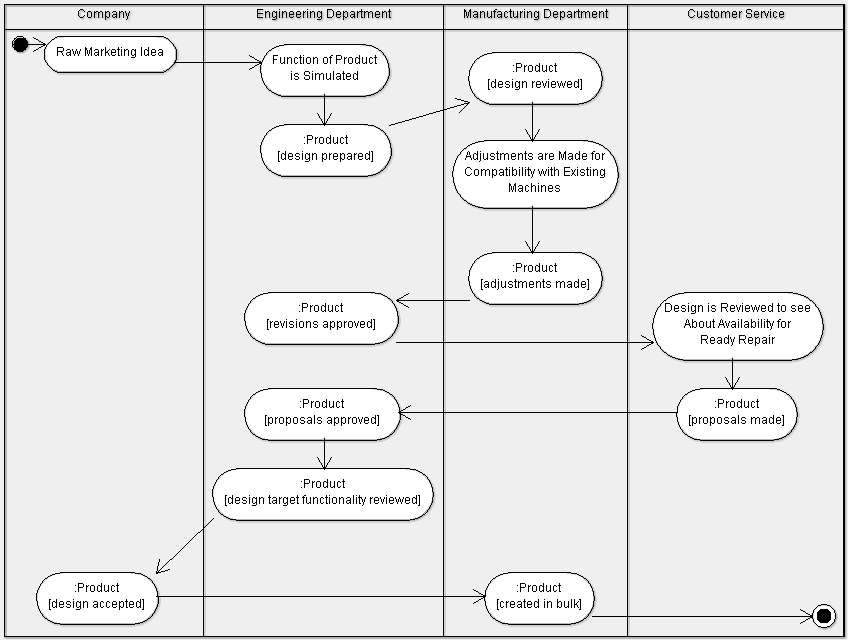
**Use Case Diagram Example:**

**Sequence Diagram Example:**

**Activity Diagram Example:**

****

**Activity Diagram Example (With Swimlanes):**

****

**LECTURE 5 (DOMAIN ANALYSIS):**

* **Development stages:**
  + System conception – Conceive an application and formulate tentative requirements
  + Analysis – Understand requirements by constructing models; specify what needs to be done, not how it is done
  + System design – Devise the architecture and object interaction design
  + Class design – Augment and adjust entity representations from analysis (Algorithms, data structures, class operations, etc.)
  + Implementation – Translate the design into programming code and data structures
  + Testing – Ensure that the application is suitable for actual use and it satisfies the requirements
  + Deployment – Place the application in the field
  + Maintenance – Preserve the long-term viability of the application
  + Incremental Iterative Lifecycle – Iteration Planning -> Requirements Capture -> Analysis & Design -> Implementation -> Test -> Prepare Release
* **Vision:**
  + *Problem Statement* – The problem of [describe problem] affects [the stakeholders affected by the problem], the impact of which is [what is the impact of the problem?]. A successful solution would be [list some key benefits of a successful solution].
  + *Position Statement* – For [target customer] who [statement of the need or opportunity]. The (product name) is a [product category] that [statement of key benefit; that is, the compelling reason to buy]. Unlike [primary competitive alternative], our product [statement of primary differentiation].
  + Stakeholders – Name the stakeholder type, briefly describe the stakeholder, and summarize the stakeholder’s key responsibilities regarding the system being developed; that is, their interest as a stakeholder.
  + User Environment – Detail the working environment of the target user; Think of number of people involved in completing the task, how long a task cycle is, any unique environmental constraints, which system platforms are in use today, and what other applications are in use.
  + The rest involves the product overview (product perspective, assumptions and dependencies, needs and features, alternatives and competition, etc.) and other product requirements.
* **Elaborating a Concept:**
  + Who is the application for? What problems will it solve? Where will it be used? When is it needed? Why is it needed? How will it work?
* **Domain Analysis:**
  + Concerned with devising a precise, concise, and understandable model of the real world.
  + Analysis starts with the concept statement generated during system conception.
* **Class Domain Model:**
  + First step – Construct the class domain model that shows the static structure of the system – Real word classes and their relations to each other
  + Find classes, prepare a data dictionary, find associations, find attributes, organize and simplify classes using inheritance, verify that access paths exist for likely queries, iterate and refine the query, and group classes into packages.
  + Find classes – Take a sentence and extract all the nouns for classes, the eliminate non-useful ones.
    - Redundant classes– If two classes express the same concept, you should keep the most descriptive name
    - Irrelevant classes– A class has little or nothing to do with the problem, eliminate it
    - Vague classes – A class should be specific, not broad
    - Attributes – Names that primarily describe individual objects should be restated as attributes
    - Operations – A name describes an operation that is applied to objects
    - Roles – The name of a class should reflect its intrinsic nature, not the role it plays
    - Implementation Constructs – Eliminate constructs extraneous to the real-world
    - Group classes into good classes and bad classes. Group the bad classes into either vague, redundant, attribute, implementation, or irrelevant.
  + Preparing a Data Dictionary – Prepared for all modeling elements; involves writing a paragraph to precisely describe each class
    - Describes the scope of the class in the context of the problem
  + Keep the Right Associations – Take any associations out that do not contribute to the overall class diagram and refine others
    - Associations between eliminated classes – If certain classes are eliminated, then the relationships they entail are not useful
    - Irrelevant or implementation associations– Eliminate associations that are outside the problem domain or deal with implementation constructs
    - Actions – An association should describe a structural property of the application domain, not a transient event
    - Ternary associations – Decompose associations among three or more classes into binary associations
    - Derived associations – Omit associations that can be defined in terms of other associations (avoid redundancy)
      * Multiple paths between classes sometimes indicate derived associations
    - Misnamed association – Don’t say how or why a situation came about, say what it is
    - Add association end names and add any missing associations
    - Qualified associations – Use qualifiers to reduce multiplicity and to uniquely identify entities
    - Specify multiplicity – For multiplicity values of “many” consider if a qualifier is needed; and check if objects are ordered
    - Use aggregation/composition to depict containment associations
    - Objects – If the independent existence of an element is important, rather than just its value, then it is an object
    - Qualifiers – If the value of an attribute is context-dependent, then consider restating the attribute as a qualifier
  + Finding Attributes – Data properties of individual objects
    - Attributes correspond to nouns followed by possessive phrases
    - Attributes are less likely to be fully described in the problem statement
      * You need to draw on your knowledge of the application domain
      * You can also find attributes in the artifacts of related systems
    - Good news: Attributes seldom affect the basic structure of the problem
    - Omit derived attributes
  + Refining with Inheritance – Organize classes to share common structures
    - Bottom-up generalization – Search for classes with similar attributes, associations, and operations
    - Top-down specialization – Top-down specializations are apparent from the application domain
      * Look for noun phrases composed of various adjectives on the class name
    - Similar associations – When the same association appears more than once, generalize the associated classes
  + Iterating the class model – A class model is rarely correct after a first pass; multiple passes are required instead
  + Grouping Classes into Packages – The last step in class domain modeling is to group classes into packages
    - Package – A group of elements (classes, associations, generalizations) with a common theme
    - To assign classes to packages look for cut points
      * Cut point is a class that is the sole connection between two otherwise disconnected parts of a model
* **Domain State Model**
  + Identify domain classes with state-dependent behavior
  + Find states & events
  + Build state diagrams
  + Evaluate state diagrams
* Identifying Classes with States – Look for classes that can be characterized by a progressive history or that exhibit cyclic behavior
  + Identify the life cycle of an object
  + List the states for each class
  + List the events
* Domain Interaction Model – Interaction model is seldom important for domain analysis
  + During domain analysis the emphasis is on key concepts and structural relations
  + Interaction model is important aspect of application analysis
* **Terminology**
  + Software Development Process – Provides a basis for the organized production of software, using a collection of predefined techniques and notations
  + System Conception – Deals with the genesis of an application
  + Analysis – Focuses on creation of models
  + System Design – The developer makes strategic decisions with broad consequences; an architecture must be formulated along with global strategies and policies to guide the subsequent, more detailed portion of design
  + Architecture – The high-level plan or strategy for solving the application problem
  + Class Design – The developer expands and optimizes analysis models; there is a shift in emphasis from application concepts toward computer concepts
  + Implementation – The stage for writing the actual code
  + Testing – After implementation, the system is complete, but it must still be carefully tested before being commissioned for actual use
  + Training – An organization must train users so that they can fully benefit from an application
  + Deployment – The eventual system must work in the field, on various platforms and in various configurations
  + Maintenance – Once development is complete and a system has been deployed, it must be maintained for continued success
  + Waterfall Development – Dictates that developers perform the software development stages in a rigid linear sequence with no backtracking (capture requirements, construct an analysis model, system design, class design, implementation, testing, and deployment)
  + Iterative Development – The nucleus of a system is developed first (analyzing, designing, implementing, and delivering working code). Then the scope of the system grows, adding properties and behavior to existing objects, as well as adding new kinds of objects. Each iteration includes a full complement of stages: analysis, design, implementation, and testing

**MIDTERM I QUESTIONS:**

1. Given a UML Class Diagram and a constraint, how can you restructure the Class Diagram to satisfy the given constraint? You may have to reconstruct part of the Class Diagram entirely so that the given constraint is satisfied.
2. Given a concept statement, perform linguistic analysis to find class candidates, then organize them into good classes and bad classes.
   1. Create a class diagram from the good classes found with linguistic analysis.
   2. Choose one class and create a Statechart for it. Find constraints within the concept statement to add to the Statechart.
   3. Draw a Use Case Diagram based on the concept statement.
   4. Select a Use Case and come up with a concrete use case, then make a detailed system sequence diagram from it.
3. Draw an Activity Diagram for the type of things that may occur for a given activity.

**Notes:**

* Please keep diagrams detailed enough to answer the question but brief enough to spare time during the 50 minute exam.
* Focus on Class Diagrams, constraints, concept statements, linguistic analysis (and organizing classes into good and bad), Statecharts, Use Case Diagrams, concrete use cases, and sequence diagrams when studying.

**PREVIOUS MIDTERM I SAMPLE:**

