Table of contents

OOAD Principles

- A Class should be as shy as possible

- Class should be highly cohesive

- How to decide what should be a class?
  - Nouns.
  - Value is a group of items.
  - Functions associated with an item.

- Many small classes are better than a few large classed
  - Many simple classes means each class
    - encapsulates less of overall system intelligence
    - is more reusable
    - is easier to implement
  - A few complex classes means that each class
    - encapsulates a large portion of system intelligence
    - is less likely to be reusable
    - is more difficult to implement

- Prefer Value objects over Entity objects

- Composition
  - Advantages
    - Contained objects are accessed by the containing class solely through their interfaces
    - "Black-box" reuse, since internal details of contained objects are not visible
    - Good encapsulation
    - Fewer implementation dependencies
    - Each class is focused on just one task
    - The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type
  - Disadvantages
    - Resulting systems tend to have more objects
    - Interfaces must be carefully defined in order to use many different objects as composition.

- Generalization
  - Advantages
    - New implementation is easy, since most of it is inherited
    - Easy to modify or extend the implementation being reused
  - Disadvantages
    - Breaks encapsulation, since it exposes a subclass to implementation details of its super class
    - "White-box" reuse, since internal details of super classes are often visible to subclasses
    - Subclasses may have to be changed if the implementation of the super class changes

        - Implementations inherited from super classes cannot be changed at runtime

- Prefer Composition & Interfaces over Inheritance
    - If inheritance is to be used, try to inherit directly from an abstract class

- Use inheritance ONLY if up casting is needed.
    - While using inheritance, the Liskov's Substitution Principle must not be violated.
    - It should be illegal for a derived class to override a base class method with a NOP method, that is, a method that does nothing.

- DRY or once and once only: Duplication is bad.

- Code should be self documenting.
    - This is possible by choosing the right variable and function names.
    - Comments are secondary because they tend to lie.

- Names
    - Use Nouns or Noun phrases for classes and variables
    - Use commonly used names like isX, getX & setX

- Avoid method overloading

- Avoid literal constants other than "", null, 0 and 1.

- An interface should be designed so that it is easy to use and difficult to misuse.

- Ask for help, not information
    - Never ask an object for information that you need to do something; rather, ask the object that has the information to do the work for you.

- Keep things that vary separately from things that are common.

- Use error handling code for conditions you expect to occur; use assertions for conditions that should never occur
    - If a global debug flag is set, then do a lot more error checking / write detailed log. This flag will usually be false in production mode.
    - Log errors for technical support personnel.

- Keep the command and queries segregated

- Single Responsibility Principle
  A class should have only one reason to change.

- Open Closed Principle
  A class should be open for extension but closed for modifications.

- Liskov Substitution Principle
  All derived classes must be substitutable for their base class.

- Dependency Inversion Principle (DIP)
  Program to an interface and not to an implementation.

- Interface Segregation Principle
  Interfaces should be as fine-grained as possible.

- The Release/Reuse Equivalency Principle
  The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package.

- The Common Reuse Principle
  Classes that are not reused together should not be grouped together

- The Common Closure Principle
  Classes that change together, belong together in a package.

- Acyclic Dependencies Principle (ADP)
  No cycles in the package diagram of a particular layer.

- Stable Dependencies Principle
  A package should only depend upon packages that are more stable than it is.

- Stable Abstractions Principle
  Packages that are maximally stable should be maximally abstract.

- Law of Demeter or Principle of least knowledge
  Each unit should only talk to its friends; don't talk to strangers.

- Samurai Principle
  Throw exception if any error occurs.

- Guidelines for all OO projects
  - Develop iteratively
  - Use component architecture
  - Use UML
  - Continuously verify quality (TDD)
  - Continuous Integration

References for OO Design

1. Object Oriented Principles Introduction by Bob Martin - http://www.infoq.com/presentations/principles-agile-oo-design
2. Strategic domain-driven design – Effective modeling for large projects http://www.parleys.com/display/PARLEYS/Strategic+Domain-Driven+Design+-+Effective+Modeling+for+Larger+Projects?showComments=true

3. Interview with Eric Evans - http://www.parleys.com/display/PARLEYS/Eric+Evans+JavaPolis+2006+interview
4. Improving application design with a rich domain model - http://www.parleys.com/display/PARLEYS/Improving+Application+Design+with+a+Rich+Domain+Model?showComments=true
5. Domain driven Design part 1 - http://www.infoq.com/presentations/model-to-work-evans
6. Domain driven Design part 2 - http://www.infoq.com/presentations/strategic-design-evans
7. Getting the Value objects right for the domain: http://www.parleys.com/display/PARLEYS/Get+Value+Objects+right+for+Domain+Driven+Design?showComments=true
8. Making Roles Explicit: http://www.infoq.com/presentations/Making-Roles-Explicit-Udi-Dahan
9. Domain Driven Design - http://www.infoq.com/presentations/DDD-Entities-Repositories-Jimmy-Nilsson

UML Introduction

Why Objects?

Procedural paradigm works as long as the problem is simple. Object oriented paradigm is needed to break the large problem into smaller ones in the context of objects. A central distinction between object-oriented (OO) and structured analysis is “division by objects rather than division by functions”.

Object orientation leads to faster development, higher reuse, easier maintenance, better quality and better architecture. These benefits are available only if careful analysis and design is done and not just by using an OO language.

Object Oriented Analysis.

- There is an emphasis on finding and describing the concepts of the problem domain.
- Focus on "What" the system must do.
- Do the right thing.
- Work effectively.

Object Oriented Design.

- There is an emphasis on defining software objects and how they collaborate to fulfill requirements. A critical, fundamental ability in OOAD is to skillfully assign responsibilities to classes.
- Focus on "How" the system will do it.
- Do the thing right.
- Work efficiently.

The boundary between analysis and design is blurred.

Modeling

60% of the programs written every year are discarded within a few months of completion, not because the program doesn't work, but because it doesn't do anything useful as per the customer. Most programmers embark on the journey of software development without a clear idea of where they're going.

Communication is a huge part of programming. Writing better Java won't do us any good, if we are building the wrong thing. As we advance in our programming career, we will find that more and more of our job demands effective communication. It doesn't really matter where our requirements come from—our team lead, an analyst, or the customer. In each case, our first job is to accurately gather each requirement and focus our customer on what you can reasonably achieve with your resources at hand. We should never plow the customer under with enormous requirements and arcane design documents. If the customers can't understand our documents, they can't focus on the real issues.
Keep the requirements simple and clear.

A model is

- Partial
- For a certain purpose

- A system of abstractions
- A cognitive tool

There are several models in play in a system.

The model is a great tool for communication between developers and users, and the better the communication is between those groups, the better the software will become, both in the short and the long run.
Models are not right are wrong; they are more or less useful.
Another way of expressing it is that technicalities (such as user interface fashion) come and go. Core business lasts. And when the core business changes, we want to change the model and the software.

Guidelines:

- A picture is worth thousand words. Models do not replace the text documentation, but they make them readable and also reduce the overall size.
- Coding without thinking about design increases overall development time by at least a factor of three, and results in much buggier code.
- Knowing an object-oriented language and having access to an OO library is necessary but not sufficient in order to create object software.
- Analysis and design provide software "blueprints", illustrated by a modeling language like the Unified Modeling Language (UML). Blueprints serve as a tool for thought and as a form of communication with others.
- A model is a simplification of reality. A model helps us to visualize the system. It specifies the structure of the system. It is useful for constructing the system. It documents the decisions made.
- Every model may be expressed at different levels of precision.
- Every non-trivial system is best approached through a set of nearly independent models.

What is UML?

- The UML is the standard language for specifying, visualizing, constructing, and documenting all the artifacts of a software system.
- UML is a set of modeling conventions that is used to specify or describe a software system in terms of objects.
- The objective of UML is to assist software development.
- UML is not a methodology.
- UML is not a visual programming language.

The word unified means

- Commonly accepted notations of OO method were combined.
- It works seamlessly throughout the life cycle of the project.
- It works across application domains.
- It works across implementation languages and platforms.
- It can be accommodated in almost any methodology.

UML diagrams consist of

- Use-Case Model Diagrams
- Static Structure Diagrams

- Class diagrams
- Object diagrams

- Interaction Diagrams
  - Sequence diagrams
  - Communication diagrams
  - Interaction overview diagram
  - Timing diagram
- State Diagrams
  - State chart diagrams
  - Activity diagrams
- Physical or Implementation Diagrams
  - Component diagrams
  - Deployment diagrams
  - Package diagram
- Composite Structure diagram

How is UML used?

Two primary methodologies while developing software are: predictive and adaptive.

Predictive	Adaptive
Attempts to Plan and predict activities and resources.	Accepts change as inevitable driver and encourages flexibility.
First define all requirements, then do detailed design and then implementation.	Iterative & Incremental development
Waterfall methodology	Agile Unified Process, Scrum, Extreme Programming, Lean
Risky and usually proves estimates wrong.	Low risk but cost and time are not known clearly at start.

Any methodology that advises in some form that "tries to define most of the requirements and then move to design and implementation" is basically predictive. Predictive tries hard to polish, stabilize and freeze the requirements; this is a losing battle. The only thing constant in this world is "change".
The hard and stiff breaks; the supple prevails.

Common *misunderstandings* regarding adaptive process are

1. We should carefully define complete models that are translated into code during construction.
2. Most requirements must be defined before starting design / implementation.
3. Most of the design must be done before coding starts.
4. A long time is spent doing requirements and design before programming starts.
5. An iteration of four months is better than an iteration of four weeks.
6. UML is a means to define designs in great detail and programming is just a simple mechanical translation of these designs into code.
7. It needs many documents.

8. We should plan a project in detail from start to finish and predict activities in each iteration.
9. Accurate plans can be made before elaboration phase is complete.

UML is generally used with an iterative development process. Iterative process manages complexity and plans for change, during software development.

Problems with UML

It is not uncommon for people to believe that no matter what task they may be engaged in, mere usage of UML somehow legitimizes their efforts or guarantees the value of the artifacts produced.
A project can succeed without UML. A project can fail with UML.

Here is a list of potential problems with UML.

- It is that it is very easy to create a large number of fancy UML diagrams that are of little help to the project. A diagram should be created only if it adds value to the goal of the project.
- The quality of an UML diagram cannot be accurately measured. It is mostly a subjective decision.
- Drawing good UML diagrams is an iterative process. It is time-consuming.
- *Any fool can write code that a computer can understand and execute. The goal is to write code that humans can understand.* Using UML does not guarantee this; though it is an aid.
- *We do not model the world; we model our perception of the world.* Can we trust our models?
- Sadly, most decisions in our industry are not based on the requirements of one or more applications. They're made for a variety of emotional, political, and social reasons. Some are valid and others less so. How then can UML help?
- Harmful is knowing how to read and draw UML diagrams, but not being an expert in design and patterns. Important is object and architectural design skills, not UML diagrams, drawing, or CASE tools. UML is only a tool for achieving better design. It has the potential to help creation of a better design. This potential may or may not be realized. *UML or any other technology never guarantees quality; people instill quality.*

Most problems in software projects are related to people and not to UML. UML is only a simple notation that can be used for analysis and design. Problems are created because people do not use UML properly. *People with awry faces generally blame the looking glass.* Usually when UML is misused, people create huge and unnecessary documentation using UML.

General Diagramming guidelines

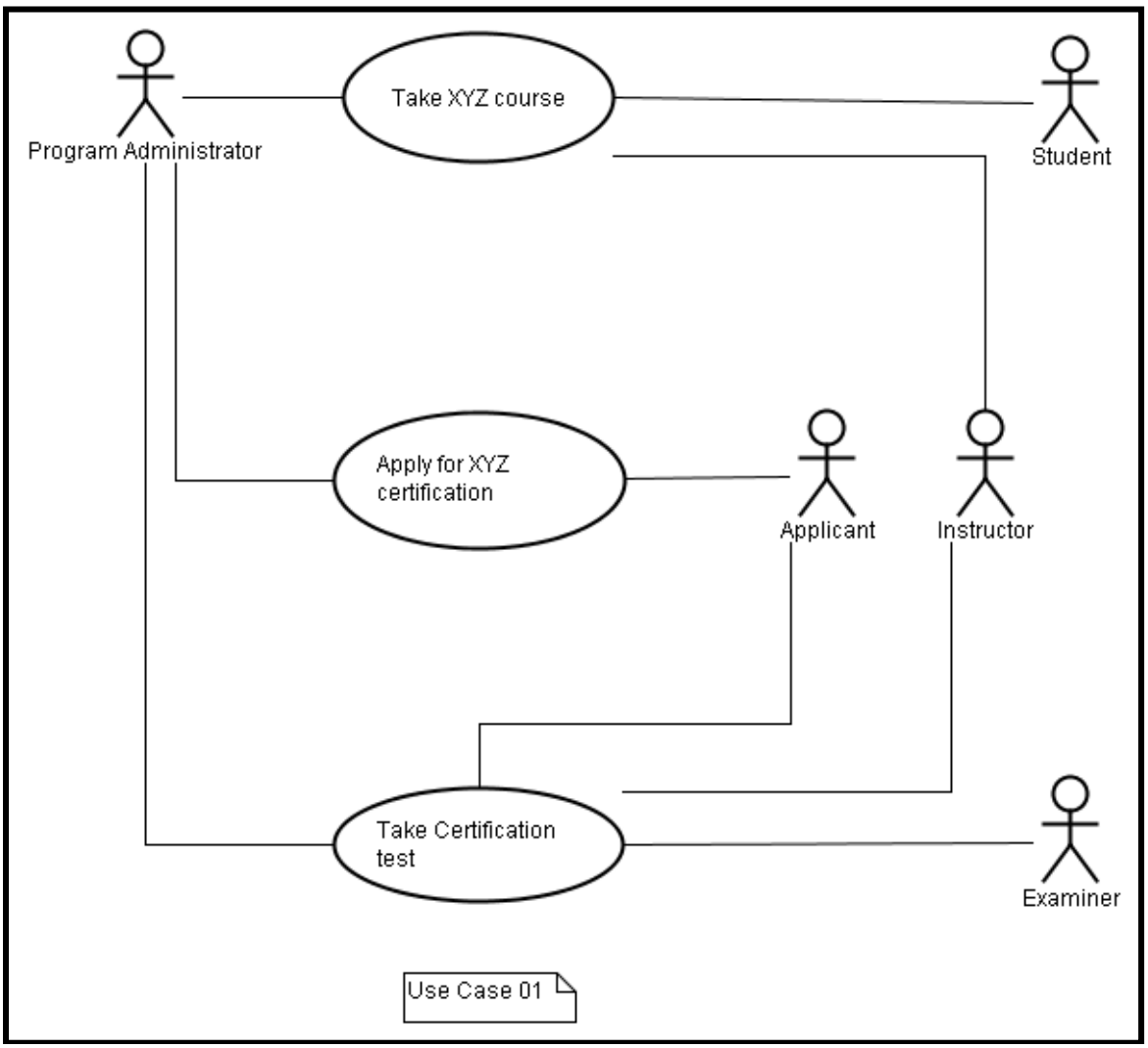
UML is not about using a fancy tool worth thousands of dollars. UML diagrams can be drawn on paper with pencil or with free tools like Jude, Omondo, etc.
Some guidelines to consider are mentioned below.

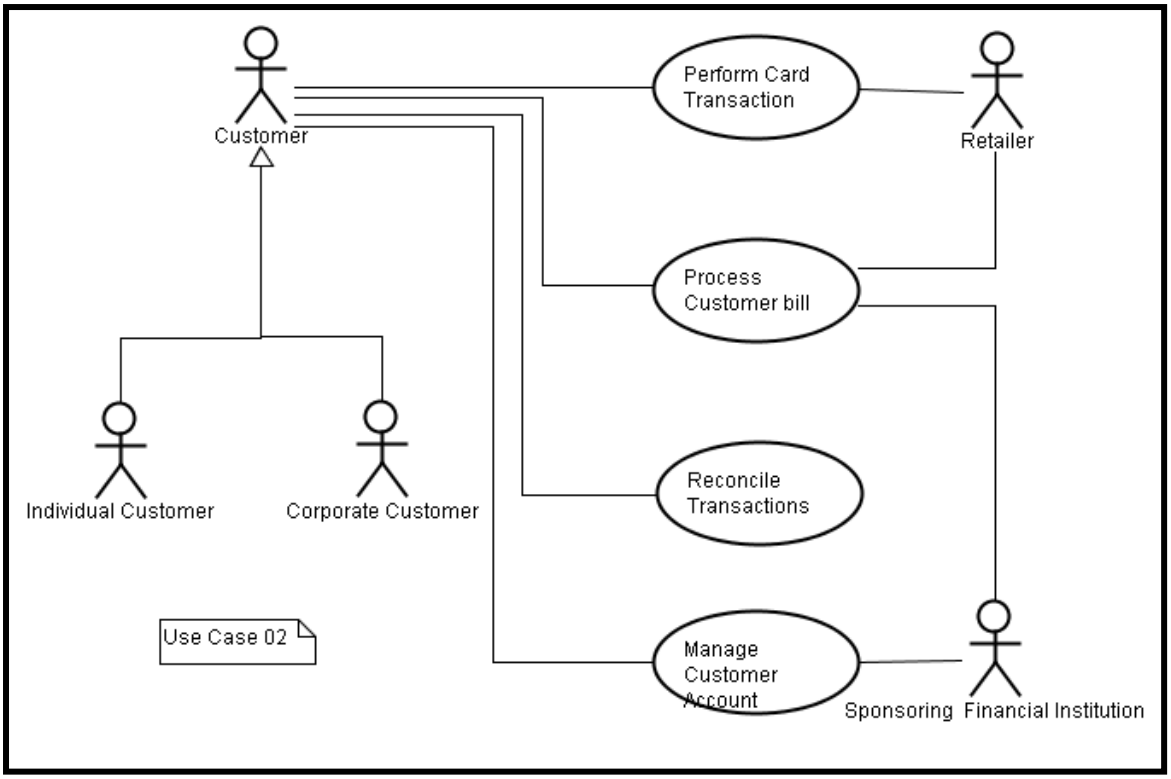
- Avoid crossing lines as far as possible. If crossing lines are needed, they should jump one another.

- Avoid diagonal or curved lines.
- Keep items of equal size.
- Show only what has to be shown. Less is more!
- Prefer well known notations over esoteric notations.
- Reorganize large diagrams into several small ones.
- Keep lot of white space in the diagrams.
- Focus on content first, appearance last.
- Organize diagrams left to right, top to bottom.
- Set and follow effective naming convention.
- Apply common domain terminology in names.
- Indicate unknowns with a question mark.
- Use Stereotypes sparingly.
- Prefer Notes to OCL (Object Constraint Language).

Use Case Diagrams

The indispensable first step to getting the things we want in life: Decide what we want.
Similarly, software cannot be written unless we know what we are trying to do.





A Use Case diagram gives an overview diagram indicating major usage requirements. It is used for analysis of the requirements of an existing/new system
A requirement is a design feature, property, or behavior of a system. It states what needs to be done, but not how it is to be done. It is a contract between the customer and the developer.

Actor is something with behavior such as person, computer system or organization. Actor is represented by a Stick figure of a person. Actor is a role that a user plays with respect to the system. Actors exist outside the system boundaries.

A use case is a single task, performed by the end user of a system, which has some useful outcome. A use case represents a functional requirement of the system. It is represented by an oval and functionality is written within the oval. Functionality is usually expressed as a verb or a verb phrase.

A key attitude in use case work is to focus on the question “How can using the system provide observable value to the user or fulfill their goals?” rather than merely thinking of system requirements in terms of a “laundry list” of features or functions.

A Use Case must return an observable result of value to a particular actor. Use cases drive the whole development process.
Use cases are not object-oriented. Use cases represent an external view of the system.

Less used Use Case symbols:

- Include Relationship represents some partial behavior that is common across several use cases.
- Generalization is used when we have one use case that is similar to another use case but does a bit more.

- An extend relationship is similar to Generalization but it has more rules to it. It is used to describe a variation of normal behavior. The extended use case has extension point's section.

A narrative is associated with each use case. Major portion of the time is to be spent on writing this narrative and not on drawing the Use case diagram. The diagram only provides a context for the narrative. *In fact, the diagram is optional; the narrative is the essence.*

An example of a use case narrative:

Use Case: Book a Party
SME: Ching, Chef at headquarters.
Estimate: 4 person-days

Description: The chef can book a party for an existing client. The steps are

1. The client calls to book the chef for the party. The chef does a search on his calendar for availability.
2. Once the chef and client agree on a date, the chef looks at the client's previous engagements and preferences.
3. The chef and the client discuss and agree on the menu items. The chef books the event with that menu.
4. The chef tells the client the cost based on date, menu and number of guests. After the conversation, the chef generates a letter to confirm the date, time, menu, guests, cost, and deposit and payment schedule. The letter requests the client to make the deposit.
5. The Cashier wants to ensure that the right payments are made at right time.

Open Issues: None.

Guidelines

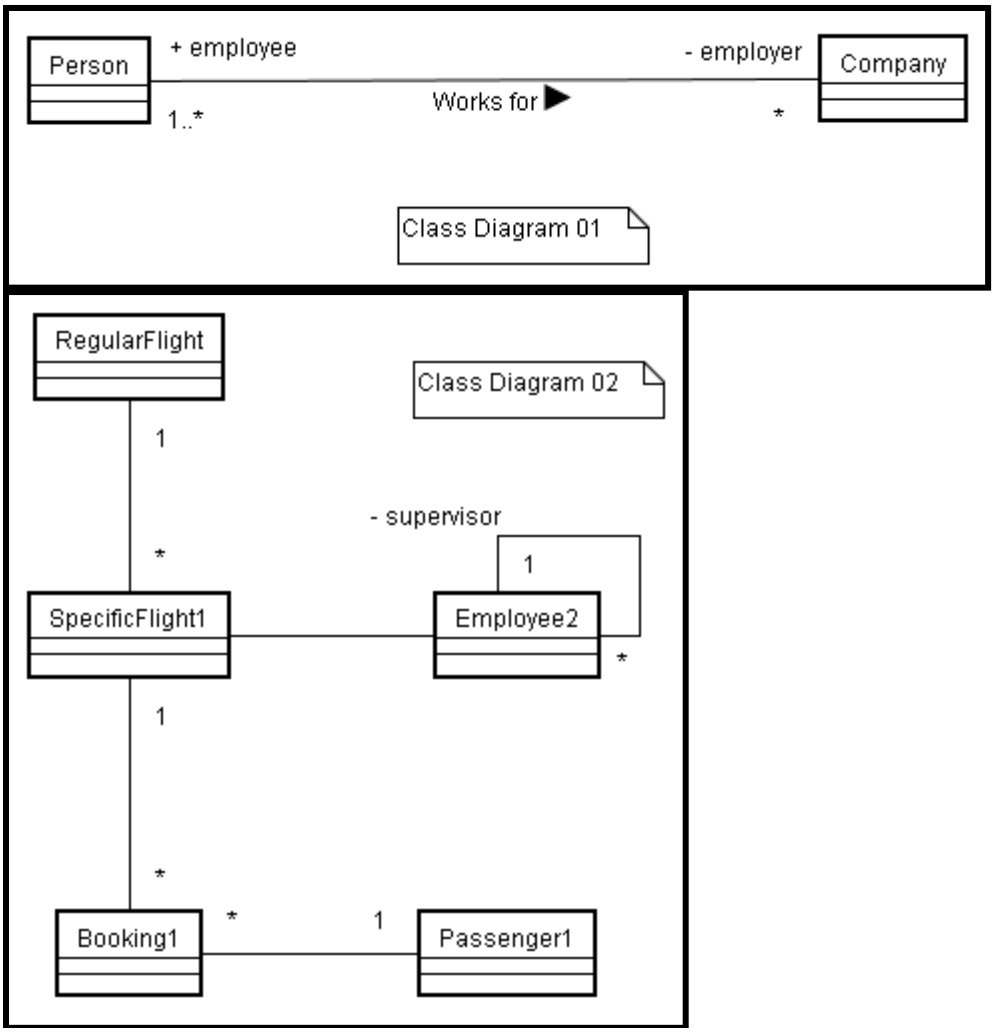
- Casual and readable use cases are useful, whereas unreadable use cases are effectively useless.
- Use cases are for capturing the major functional requirements and not all the requirements.
- Use cases specify what the system must do and *not* how it will do it.
- Do not go into details. The use case narrative is not pseudo code. Focus on the actor's intent and not how it will be implemented.
- Coding cannot start after a use case is designed. Design must follow after a use case is prepared.
- Newcomers either have too many use cases that are doing too little or few use cases doing too much.
- Using programmers to write Use Cases is discouraged, because a programmer tends to write a Use case as pseudo code. Using SME (Subject Matter Experts) to write Use Cases is discouraged because many of the business requirements are obvious to the SME. Usually SME along with analyst writes the Use cases.
- CRUD (Create-Retrieve-Update-Delete) based Use cases are useless.
- Use cases usually focus on the "Happy Path" or basic flow. The "happy path" is that sequence of steps where everything works perfectly and the goal of the use case is achieved.
- Avoid conditions and branching in the basic flow. Keep them for alternate flows.

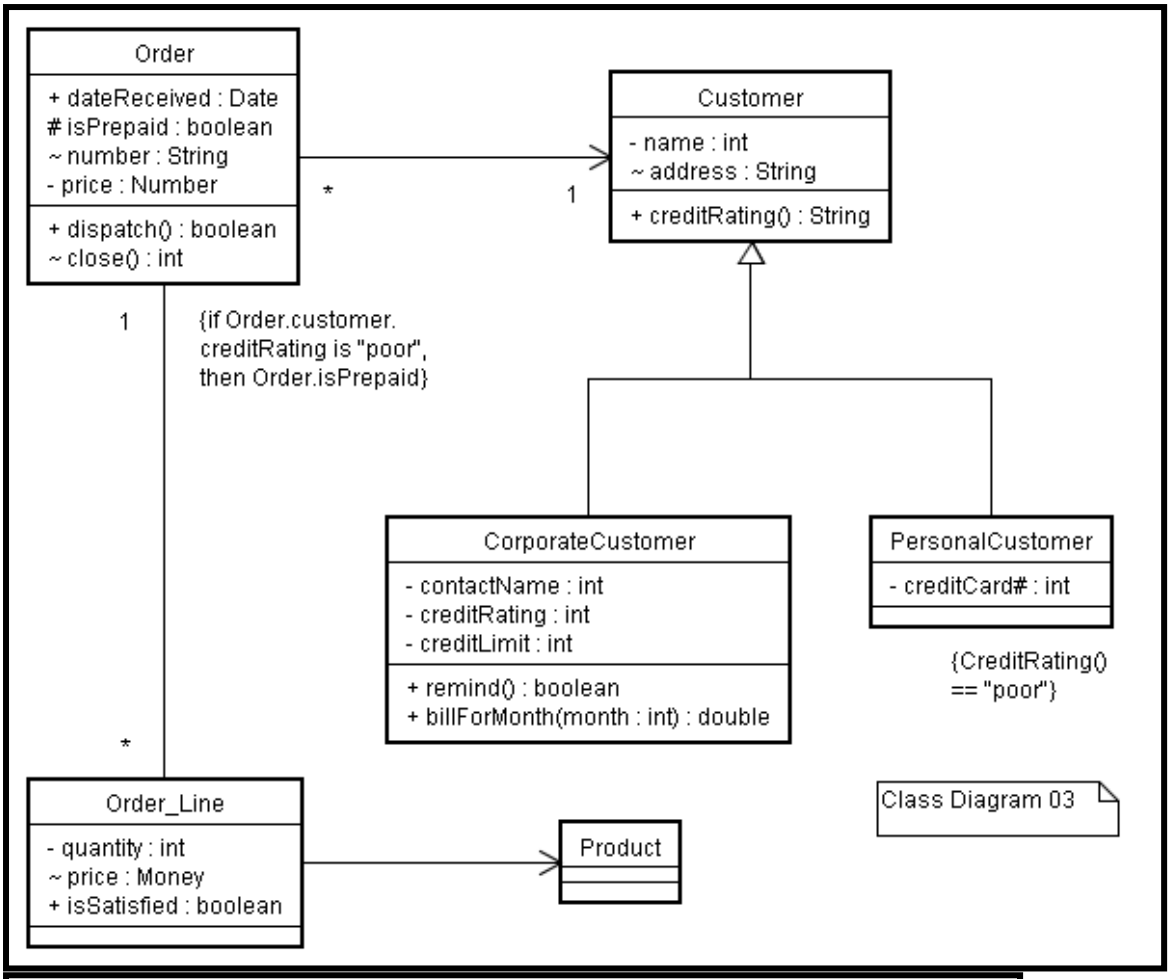
- Use cases represent an external view of the system. There need not be a correlation between use cases and classes inside the system.
- Keep diagrams simple. One of the primary purposes of use cases is to communicate with the users - the customers - of the system. Do not worry at this stage about how the classes will be drawn and how they will interact.
- Do not to show the customer the final GUI along with initial Use Cases. Avoid being bogged down with details.
- If a requirement is not relevant or very obvious, do not bother making a Use Case.
- A use case, by itself, does not describe the flow of events needed to carry out the use case. Flow of events can be described using Use Case description, pseudo code, or activity diagrams.
- Use Case names in Ovals should begin with a strong Verb e.g. withdraw, register, deliver, etc. Avoid weak verbs like process, perform, do, etc.
- Name use cases using domain terminology.
- Keep important items at top left corner e.g. the primary actor.
- Actor names should be singular, business relevant nouns. Actor names should not be job titles.
- Actors do not interact with each other.
- Use Actor "Time" for scheduled events
- Arrowheads on links are *not* for information/data flow. They only indicate the direction in which the request for service is sent. It points from the initiator to the initiated action. Arrowheads can cause misunderstanding and hence it is better to avoid them.
- Avoid more than two levels of Use Case diagrams.
- Use Include, Extend & Generalization for Use Cases association sparingly. Leave the details for later stages.
- Avoid meaningless system Boundary boxes.
- Avoid adverbs (e.g. very, more, rather, etc) in Use Case Narrative.
- Have a glossary of terms for the project.

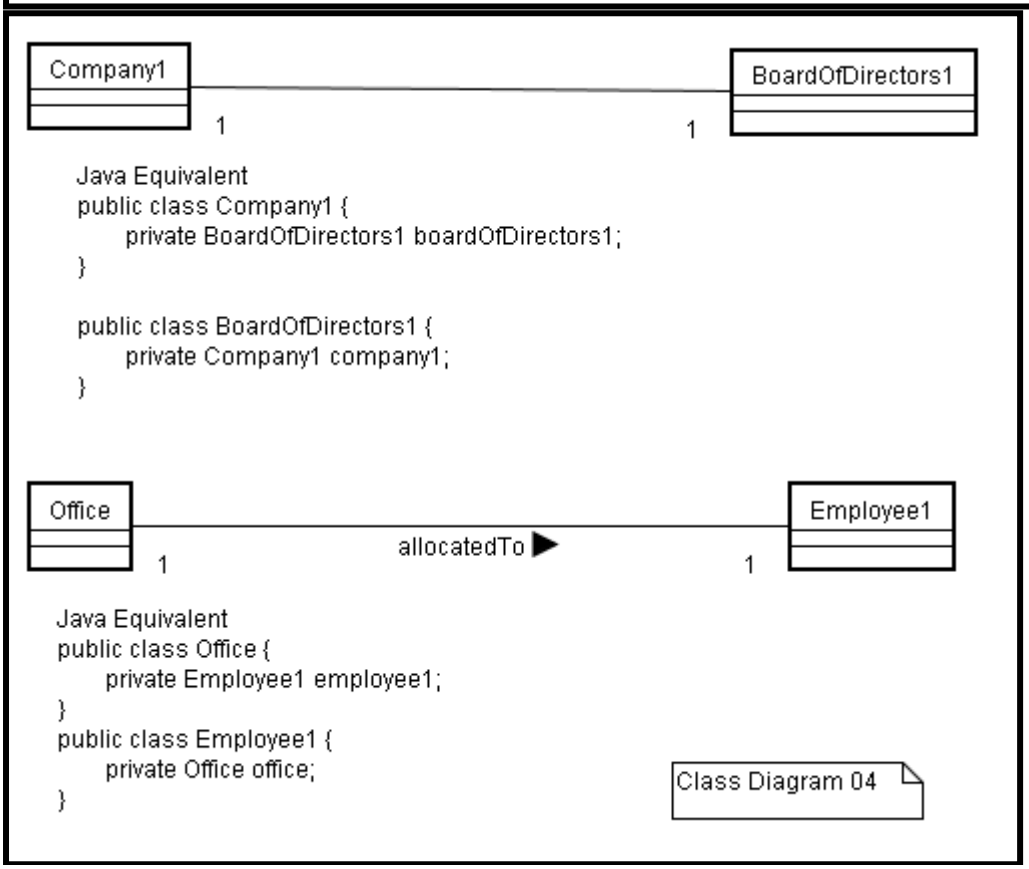
A verbal contract has no value. Written documents (including this one) give the illusion of correctness; it is only an illusion.

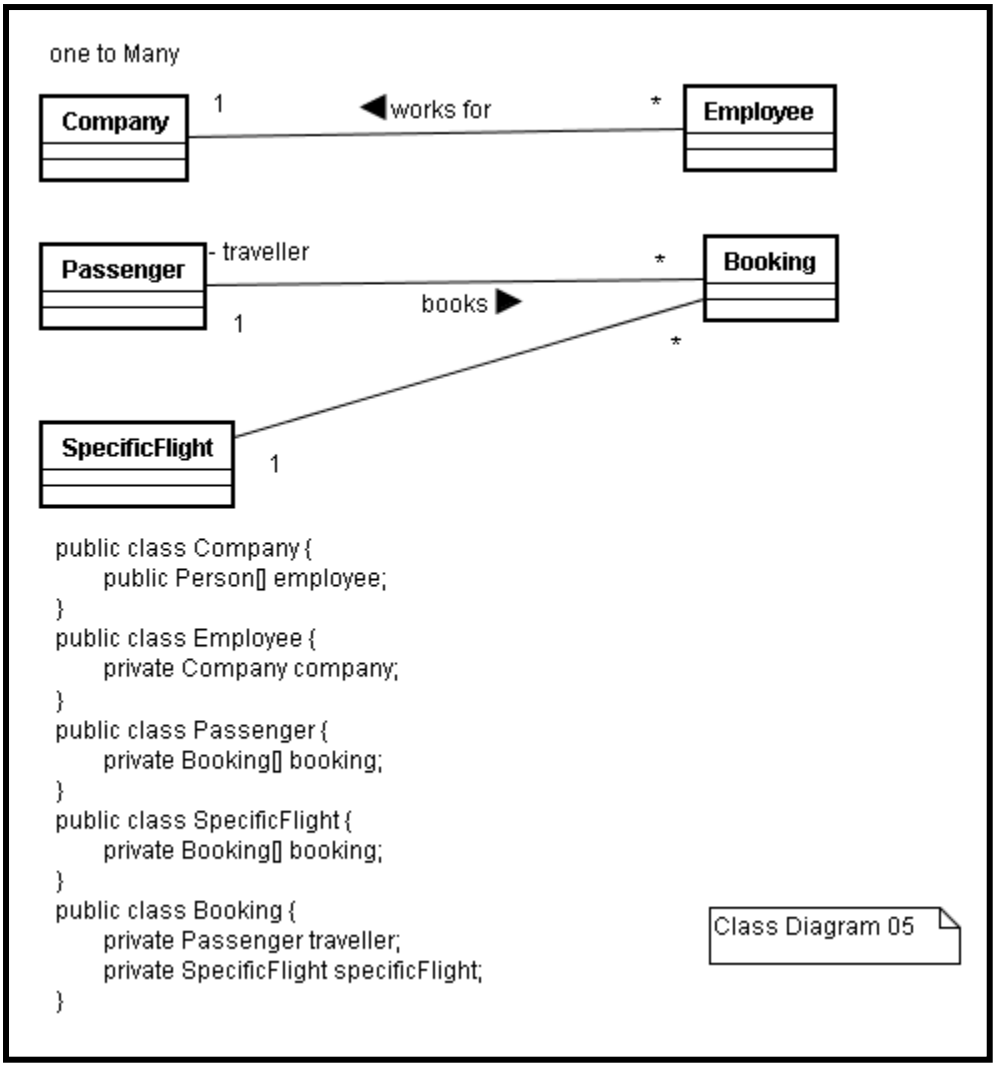
Use cases written will not be 100% correct – guaranteed. The programmer will complain that it lacks critical information and contains wrong statements. Use cases serve as a starting point not the commandments for software development. Ongoing personal communication is needed between the SMEs and developers, even after the use cases are written.

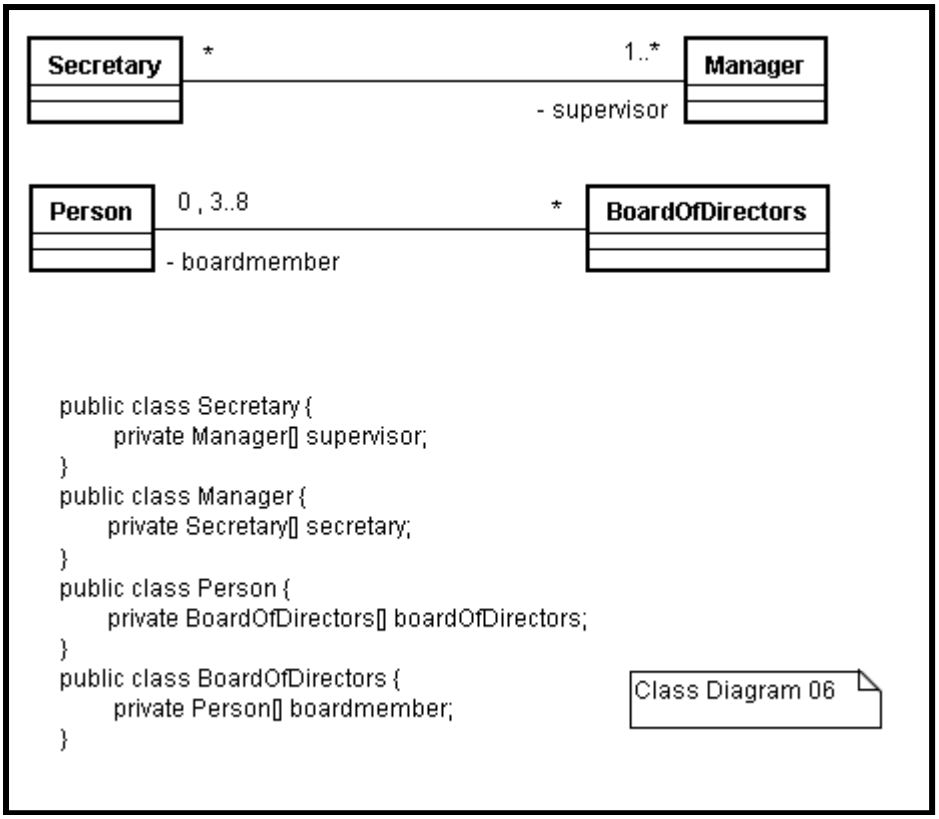
Class Diagrams




Order
+ dateReceived : Date
# isPrepaid : boolean
~ number : String
- price : Number
+ dispatch() : boolean
~ close() : int
*
1
Customer
- name : int
~ address : String
+ creditRating() : String
1
{if Order.customer.
creditRating is "poor",
then Order.isPrepaid}
CorporateCustomer
- contactName : int
- creditRating : int
- creditLimit : int
+ remind() : boolean
+ billForMonth(month : int) : double
PersonalCustomer
- creditCard# : int
{CreditRating()
== "poor"}
*
Order_Line
- quantity : int
~ price : Money
+ isSatisfied : boolean
Product
Class Diagram 03


Company1
1
1
BoardOfDirectors1
Java Equivalent
public class Company1 {
private BoardOfDirectors1 boardOfDirectors1;
}
public class BoardOfDirectors1 {
private Company1 company1;
}
Office
1
allocatedTo
1
Employee1
Java Equivalent
public class Office {
private Employee1 employee1;
}
public class Employee1 {
private Office office;
}
Class Diagram 04


one to Many
Company
1
works for
*
Employee
Passenger
- traveller
1
books
*
Booking
*
SpecificFlight
1
public class Company {
public Person[] employee;
}
public class Employee {
private Company company;
}
public class Passenger {
private Booking[] booking;
}
public class SpecificFlight {
private Booking[] booking;
}
public class Booking {
private Passenger traveller;
private SpecificFlight specificFlight;
}
Class Diagram 05



Class diagrams are for visualizing, specifying and documenting the system from a static perspective.

Class diagrams will have different levels of detail (abstraction) depending on where we are in the software development process. Class diagrams are of two types

1. Conceptual. It is related to the concepts of the domain under study. It does not have a direct mapping with the language code. It is used in analysis. It is language independent and technology independent. It shows real world objects and not the software classes that are to be built. It is built over several iterations during the elaboration phase. Conceptual model are related closely to software interfaces and not classes. A conceptual class diagram is also called domain model because it is essential to have a domain expert while building this diagram. The purpose of this diagram is to enhance comprehension about the system.

2. Software: Here the emphasis is on the interface of the class and not implementation. They are used while working with software. Here we try to answer the questions "What classes are needed? How does the class do its work?" It shows real world classes. The conceptual model is used as a base for the design model.

Multiple class diagrams are required to model large systems.

Associations represent relationships between instances of classes. An association should exist if a class *possesses / controls / is connected to / is related to / is a part of / has as parts /* is *a member of* / has *as members* some other class in your model
Associations between classes have multiplicity.

Associations have navigability indicated by arrows. No arrows can be bi-directional or unspecified navigation.
Association names are verbs or verb phrases.
Association can have roles at both ends. Each role is

- The face that a class on one end of an association presents to the class on the other end of the association.
- A class can participate in many associations and thus have multiple (different) roles.
- Role names are nouns.

Generalization is inheritance in OO languages.
A constraint is a condition that every implementation of the design must satisfy. Constraints are written in curly braces { }.

A well-defined class is loosely coupled (few entry points) and highly cohesive (all members work toward a common functionality).

The biggest danger with class diagrams is that people can get bogged down in implementation details far too soon.

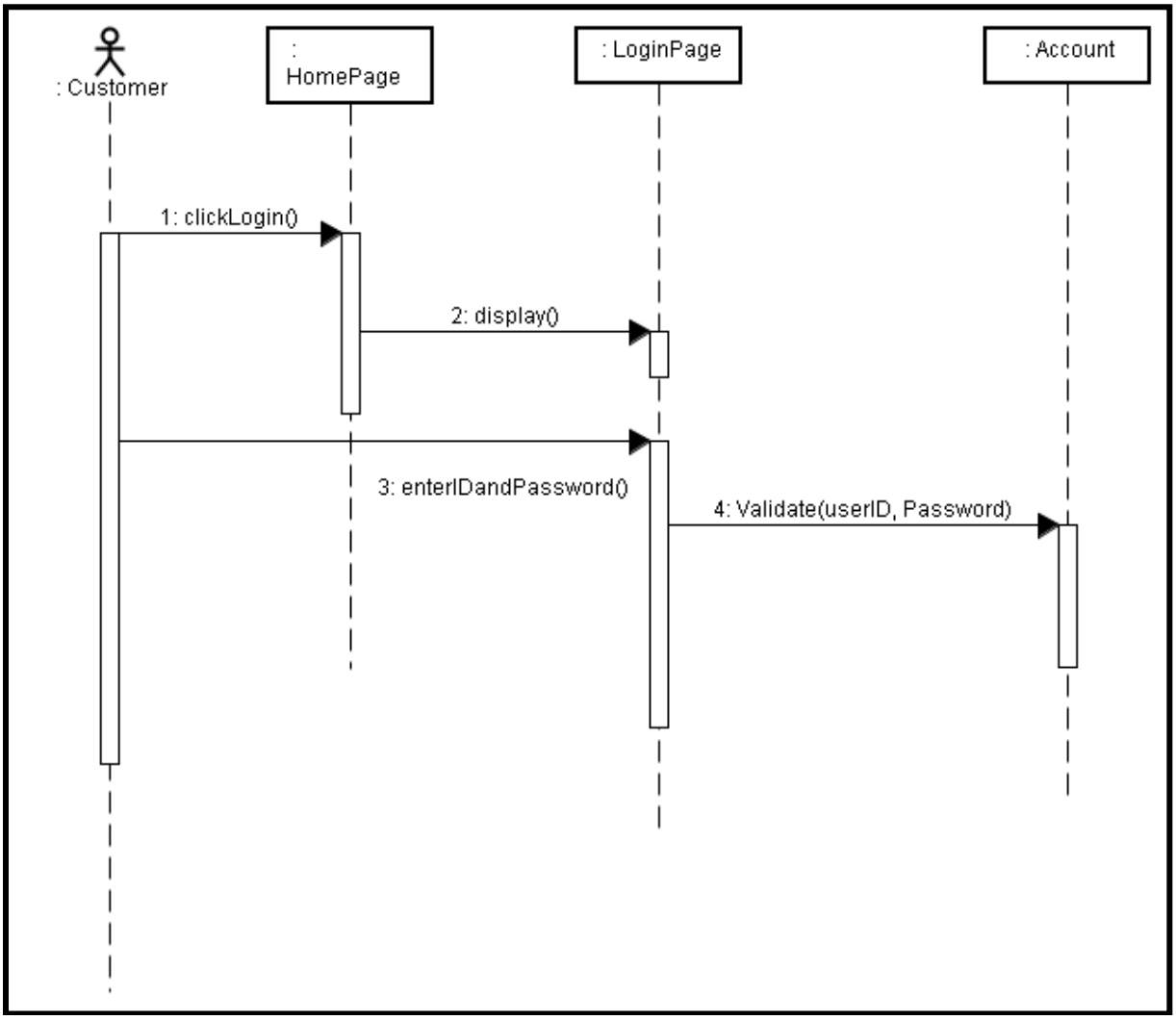
Guidelines

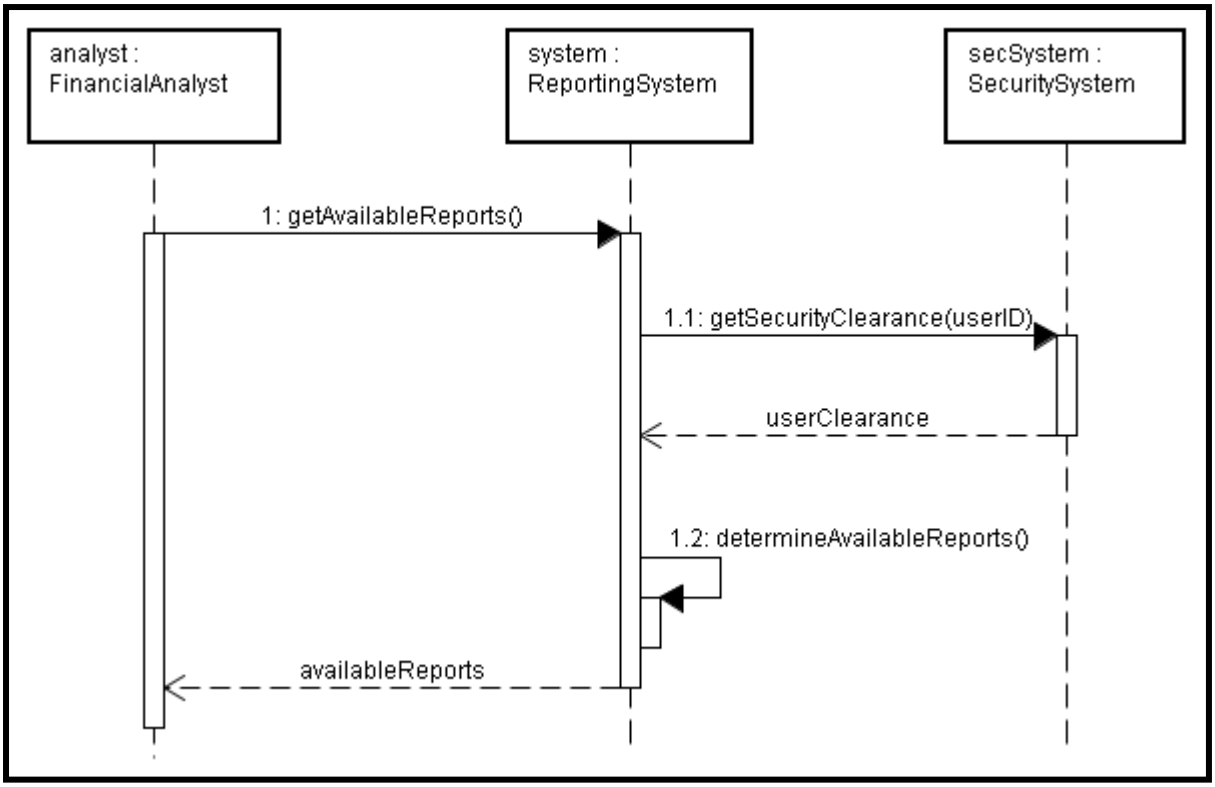
- Use Singular nouns for Class names.
- Name operations with a strong verb. Write and Center names on associations. Name of the association should be in active voice. E.g. “places” instead of “is placed by”
- Name attributes with a domain based noun.
- Associations should preferably be horizontal.
- Indicate direction on association name. Association direction should preferably left to right.
- Use “lollipop” notation to indicate that a class realizes an interface.
- Do no models for scaffolding code.
- Indicate incomplete lists by …
- Depict models simply. Do not attempt to show every single dependency between classes. Show only the important ones. Do not show implied relationships.
- Use Role names when multiple associations exist between two classes. Also, Indicate Role names on recursive associations.
- Use Aggregation and Composition only when we are interested in whole and part separately. Depict the whole to the left of the parts. If in doubt, whether to use aggregation or composition, leave it out. There is minimal difference between association, aggregation and composition at coding level.
- Avoid showing redundant or derivable associations.

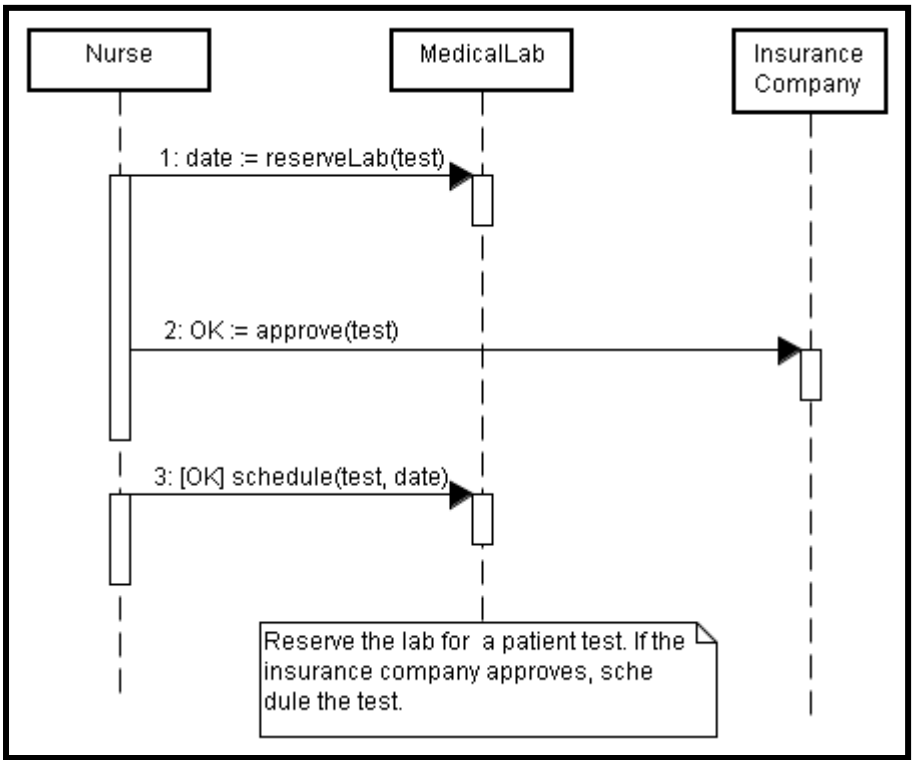
A non-domain class diagram should be kept till we need to communicate the internal structure of your software to others. The value of these diagrams is low in the long run.

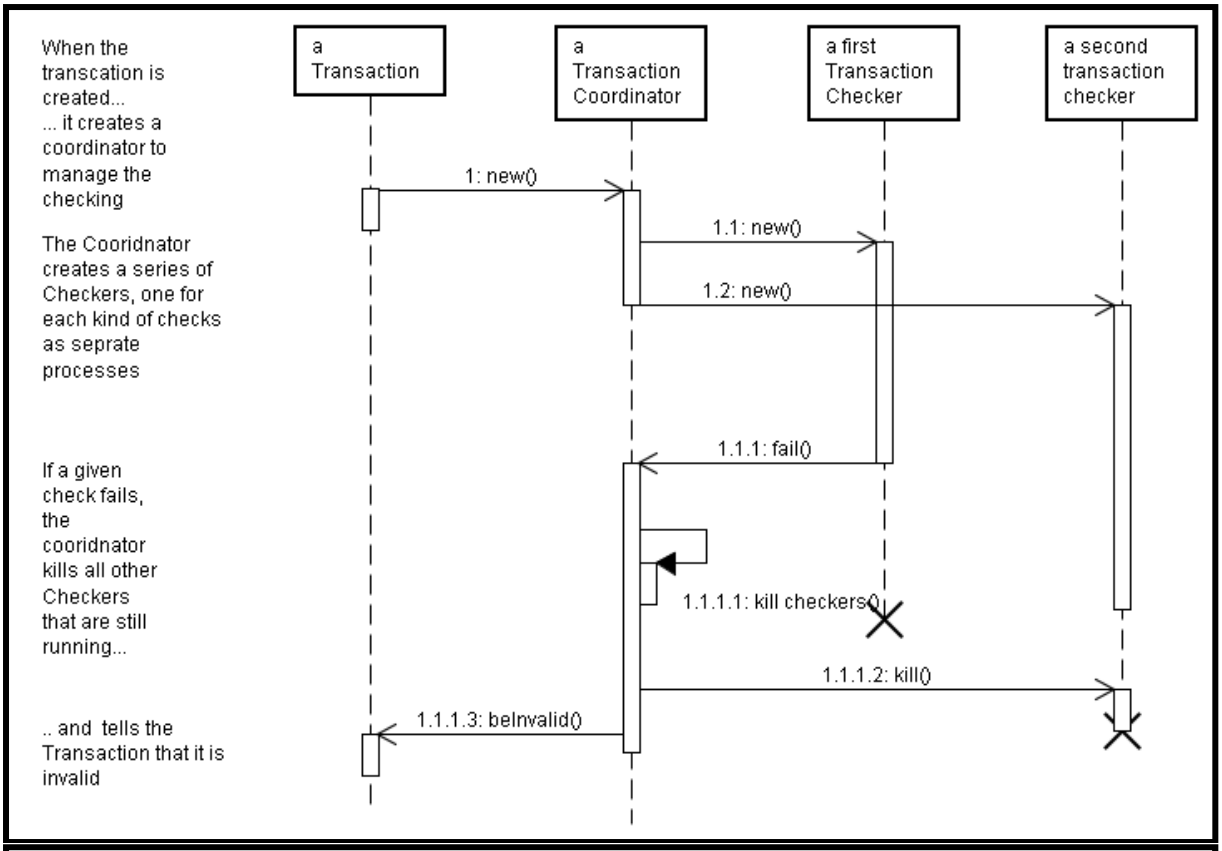
Stereotypes are the core extension mechanism for the UML. If we find that we need a modeling construct that isn't in the UML but is similar to something that is present, then we treat our construct as a stereotype of the existing UML construct.
A stereotype is showing in text between << >>.

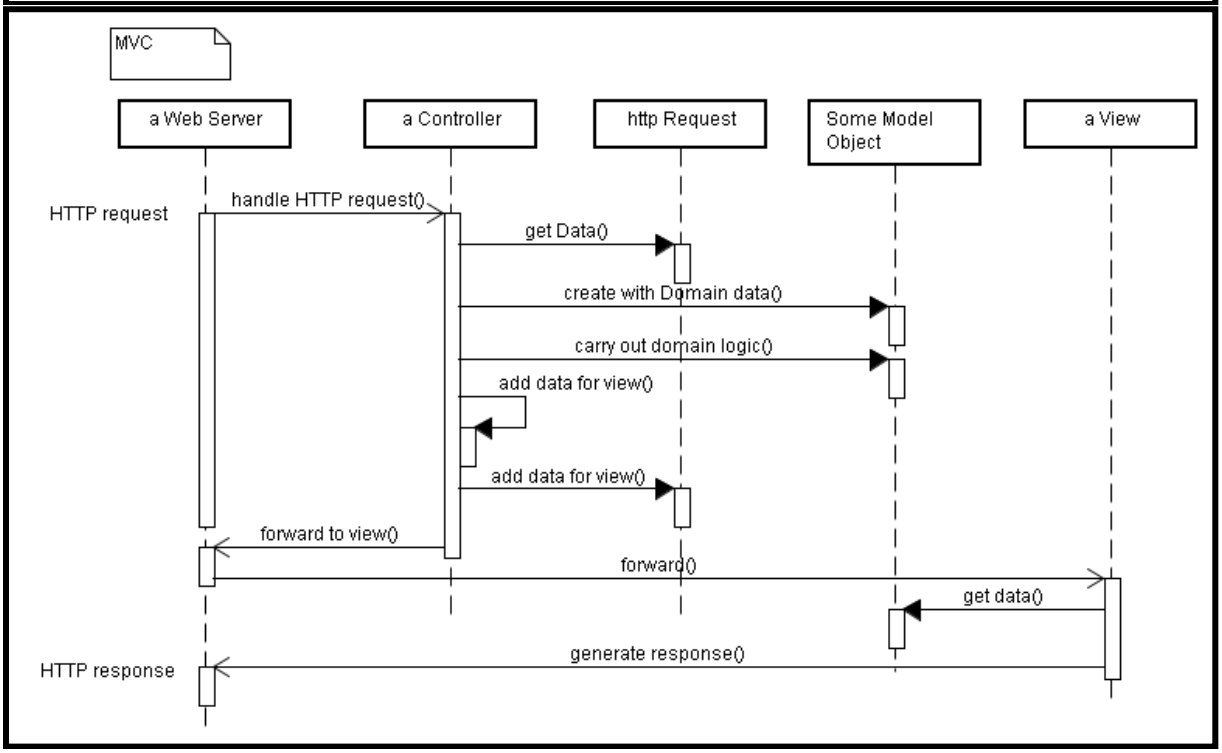
Sequence Diagrams




analyst :
FinancialAnalyst
system :
ReportingSystem
secSystem :
SecuritySystem
1: getAvailableReports()
1.1: getSecurityClearance(userID)
userClearance
1.2: determineAvailableReports()
availableReports


Nurse
MedicalLab
Insurance
Company
1: date := reserveLab(test)
2: OK := approve(test)
3: [OK] schedule(test, date)
Reserve the lab for a patient test. If the
insurance company approves, sche
dule the test.





These diagrams are the storyboards of selected sequences of message traffic between objects. Their popularity is next only to class diagrams.

Sequence diagrams commonly contain objects and messages. The object is shown as a box at the top of the diagram. The vertical line is called object's lifeline. Arrows between lifelines represent messages.

A box on the lifeline means that the object method is active i.e. it has data on the stack.
Time passes from top to bottom on Y-axis.
[Condition] – The message is sent only when the condition is true.
* - Indicates that the same message is sent multiple times.
A dotted line with arrow indicates return. Showing a return from method is optional.
X – Indicates that the object is no more necessary and may be deleted.

Guidelines

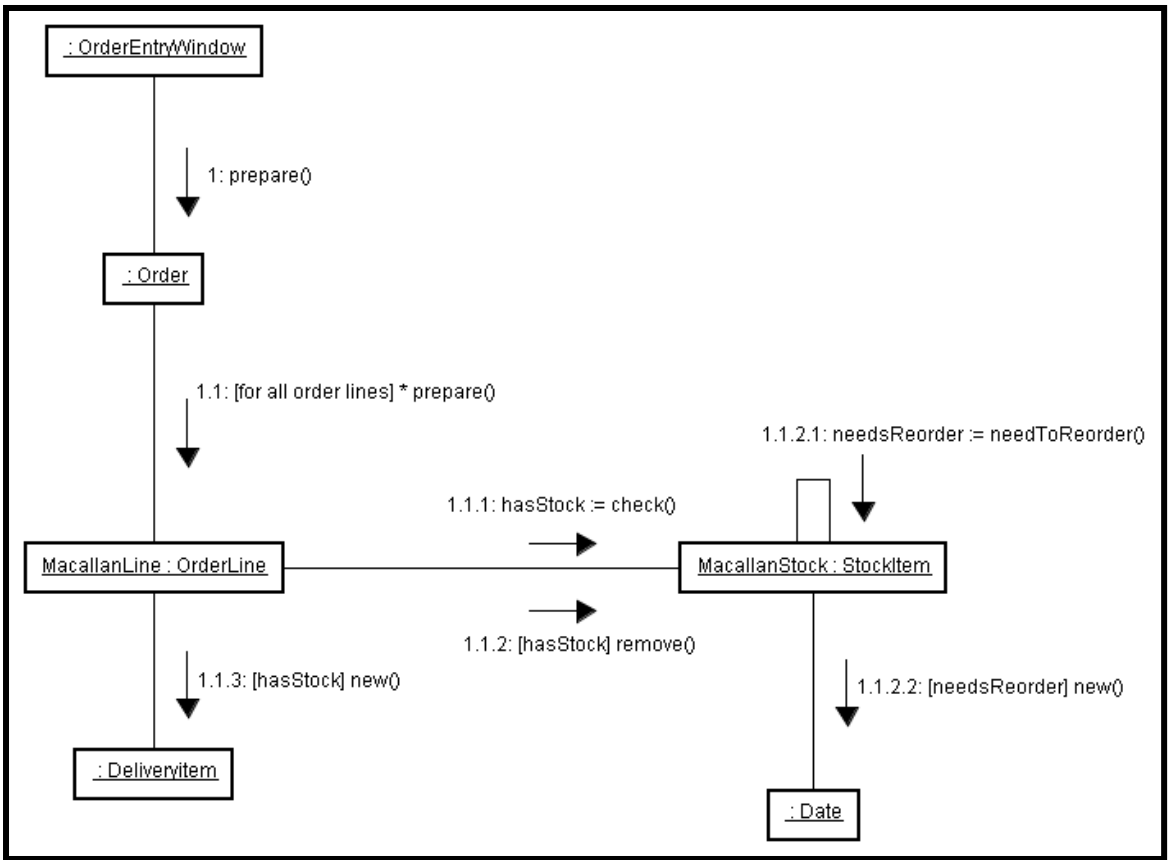
- The important objects must be to the left corner.
- The name of the actors used (if any) should be the same as that in Use Case diagrams.
- The name of classes should be the same as that in Class diagram.
- The left hand margin usually contains a description of the diagram.
- Avoid “X” to denote deletion of an object. Let the programmer / languages decide about deletion of object.
- Object is named as “name:ClassName”. The name is optional. It becomes mandatory, if more than one object of the same class is present in a single diagram.
- Use names in methods and their parameters e.g. addDeposit (amount,target,transactionID) is better than addDeposit (Currency,Account,int).

Sequence diagrams are used only for clarity by the designer. The designer uses these diagrams to communicate with the user and / or programmer. The diagrams are usually discarded after the code for that diagram is written. They have only a transient lifetime.

Sequence Diagrams.

- Iteration or looping is indicated with a "*" and an optional iteration clause. For example: *[i:1..N]:num:=nextInt().
- Iterations are shown within a box that is used to enclose an iteration area. The * [...] is used as an iteration marker.

Communication or Collaboration Diagrams



Communication diagrams are equivalent of Sequence diagrams. They just use a different notation.

There are two types of interaction diagrams …

- *Sequence diagrams* emphasize the time ordering of message traffic between objects.
- *Communication diagrams* emphasize the structural relationships between objects that send and receive messages.

It does not matter whether we use sequence or communication diagrams to model interactions. Both types of diagrams are semantically equivalent and can easily be converted from one format to the other.

Communication diagrams, like Sequence diagrams, are used only for clarity by the designer. The designer uses these diagrams to communicate with the user and / or programmer. The diagrams are usually discarded after the code for that diagram is written. They have only a transient lifetime.

Guidelines

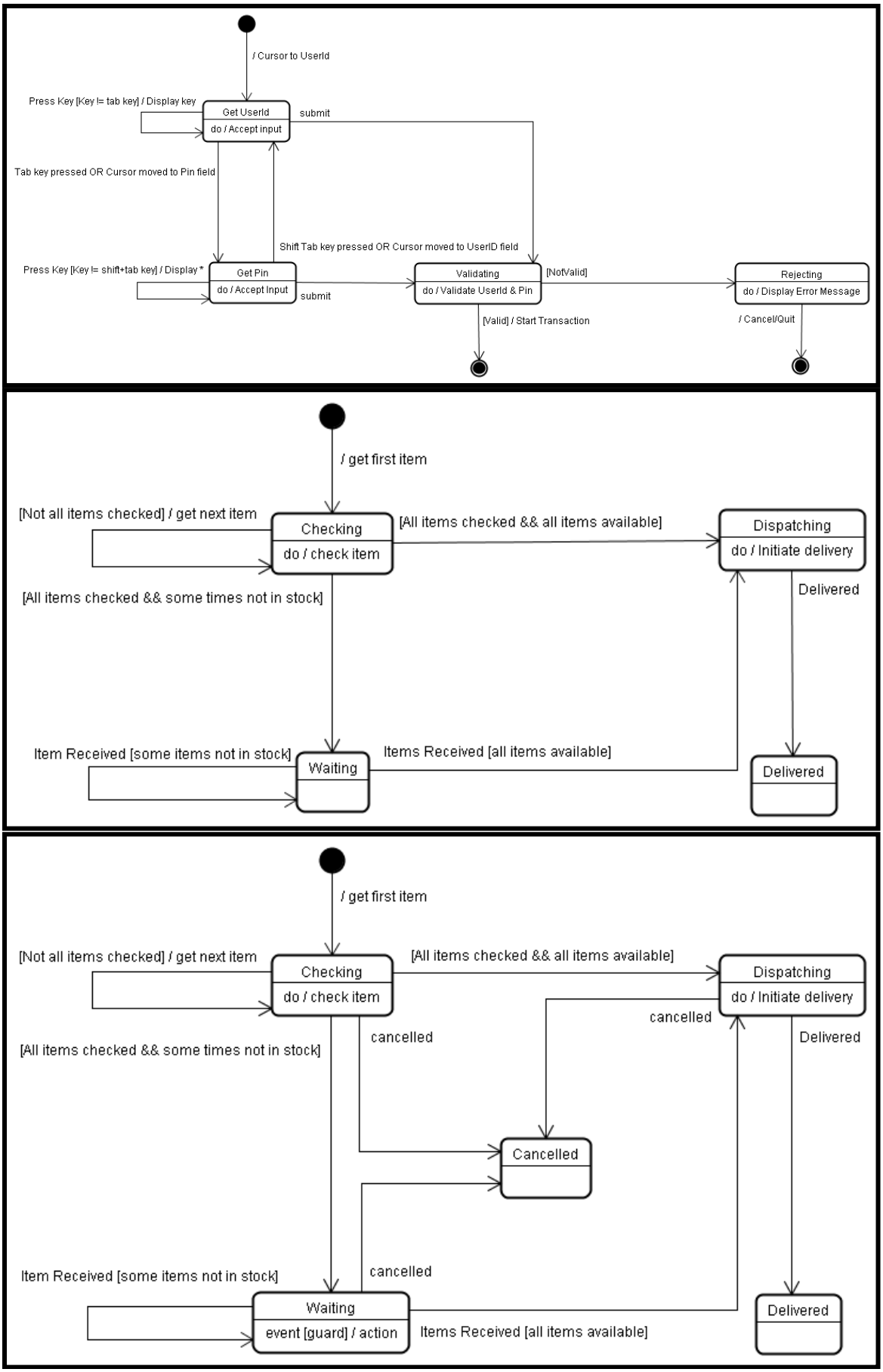
- Interaction diagrams show interaction among objects typically within a single use case only. Activity diagram show behavior across multiple use cases. State diagrams show behavior of a single object across multiple use cases.
- Interaction diagrams are good at designing part or all of one use case's functionality across multiple objects.
- Interaction diagrams allow the analyst to show iteration and conditional execution for messaging between objects.

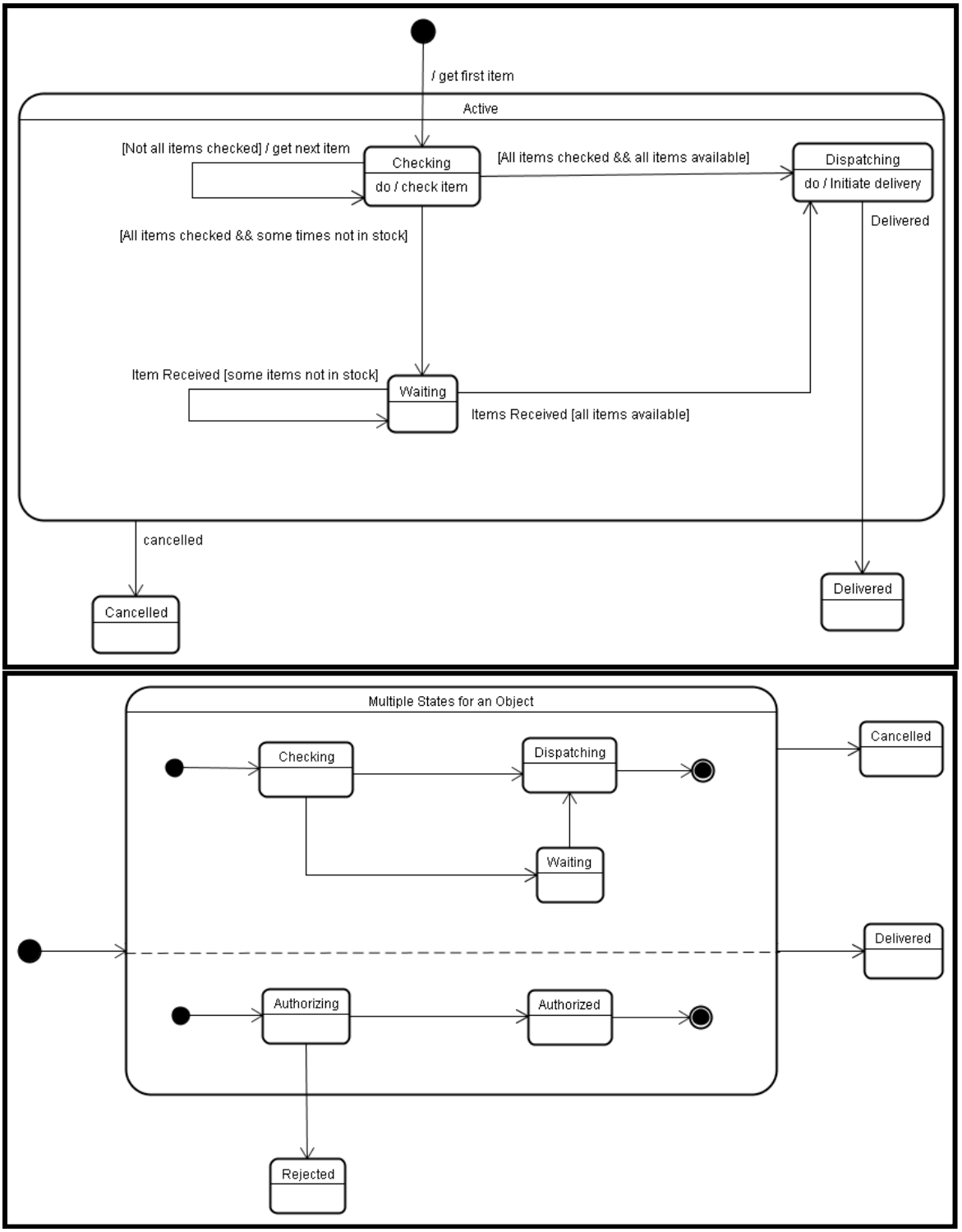
- Communication diagrams model interactions between objects, and objects interact by invoking messages on each other. If we want to model process or data flow, instead of the message flow, then we should consider drawing an Activity diagram.
- The basic notation for a message is
  seqNo loopIndicator : returnValue := methodName (parameter: parameterType) : returnType
- Indicate a return value only when needed.
- In general, if something doesn't add value then don't include it in the diagrams. We must not try to show each and every parameter for every message in the diagram. Show only the parameters that help to make the diagram achieve its objectives.
- Do not show multiplicity, navigability, name of relationship, association details on Communication diagram. These items are reserved for Class diagram.
- Showing asynchronous messages is difficult in Communication diagram. Use Sequence diagram for threads as far as possible.

Communication Diagrams:

- When either one or another message could execute, mutually exclusive messages notation is used. A letter is appended to the sequence number, examples: 1a. The letter "a" is first used by UML convention.
- Nested messages are still consistently pre-pended with their outer message sequence, for example the message 1b.1 is nested message within 1b.
- Iteration or looping is indicated with a "*" and an optional iteration clause following the sequence number. For example: 1*[i:1..N]:num:=nextInt().

State Diagrams





A state chart diagram shows the life cycle of an object; what events it experiences, its transitions and the states it is in between events.
An event is a significant occurrence. It is a trigger that can cause state of the system to change.
A state is the condition of an entity (object) at a moment in time - the time between events.
E.g. a telephone is in the state of being idle after the receiver is placed on the hook and until it is taken off the hook.

A transition is a relationship between two states; it indicates that when an event occurs, the object moves from the prior state to the subsequent state. E.g. when an event off the hook occurs, transition the telephone from the idle state to active state.
At any point of time, the system can be in only one of the states. States are connected by transitions.

The transition label is “Event [Guard] / Action”.
Action are associated with transitions do not consume any noticeable amount of time. They are uninterruptible.
State is labeled “do / activity”. An activity is associated with states and usually takes some amount of time. They may be interrupted. As long as the object is in this state, it performs the activity.
A guard is a Boolean condition. Transition occurs only when the value is true.
A transition with event means that the transition occurs when the event occurs.

This diagram is used to

- Design the behavior of a single complex class. During coding, usually a set of classes will be used for implementing this complex class.
- Analyze a complex business process.

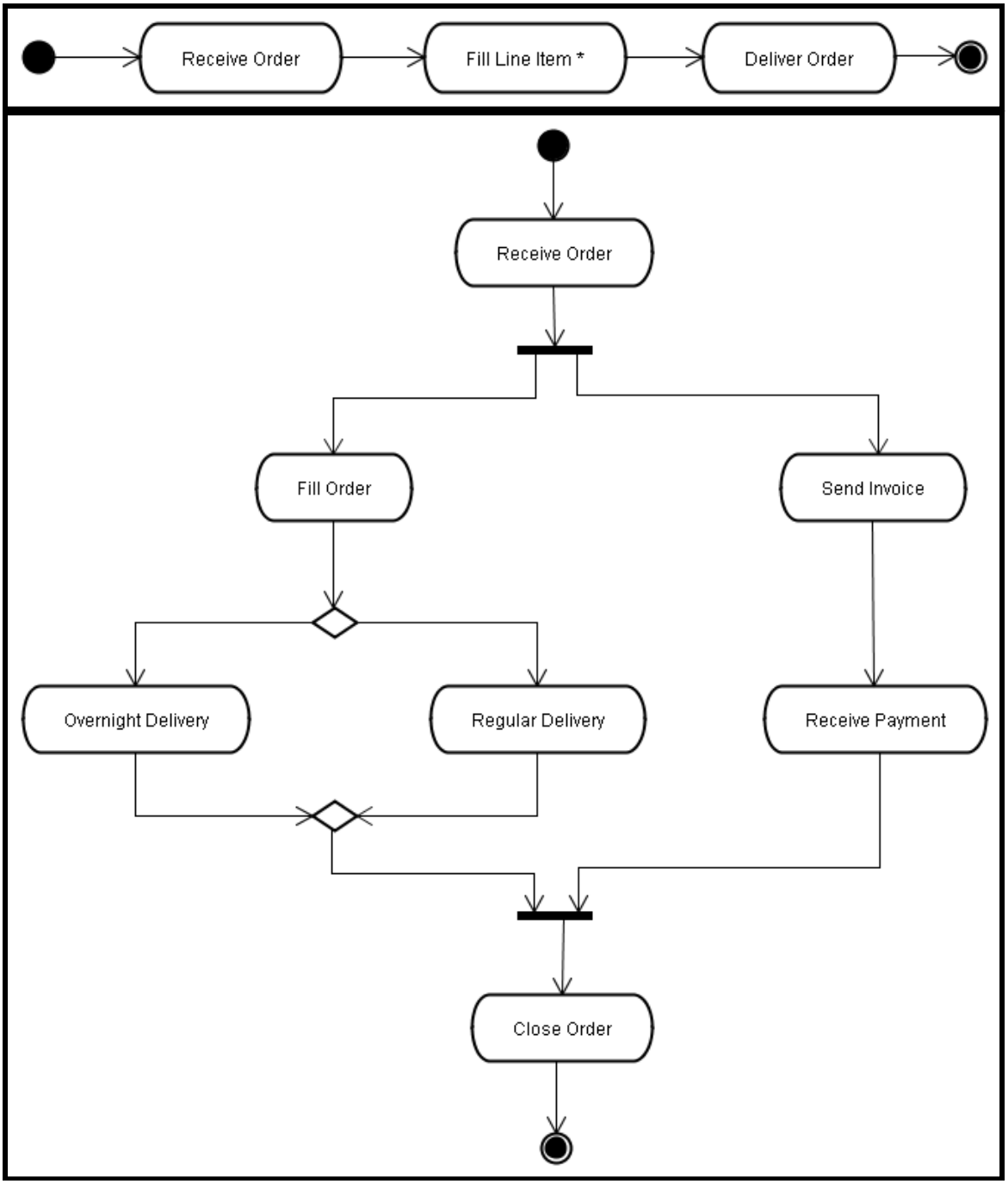
Guidelines

- This diagram is good at describing the behavior of an object across several use cases.
- This diagram should not be used to show the collaboration between objects.
- Do not draw this diagram for every class in the system. It would be a waste of time.
- Real-time software design tends to use this diagram more as compared to others.
- A state diagram need not illustrate every possible event for the object but it should try to illustrate the important events.
- The initial state should preferably be in top left corner.
- The final state should preferably be in bottom right corner.
- State names should preferably be in present tense. They should be simple and descriptive.
- Name transition events in past tense.

This diagram may be retained as a part of documentation for the project. However, its utility is low. We can usually discard this diagram after the code for that class has been written.

This diagram is also known by the names of State machine diagram, State chart, State transition diagram and Petri nets.

Activity Diagrams



This diagram is a sequence of actions. Each action is a state of doing something or an operation or a step in the business process or an entire business process.

- Actions are connected by transitions. A transition is automatically triggered when the action finishes.
- The great strength of activity diagrams lies in the fact that they support and encourage parallel behavior. Parallel behaviors are shown by fork and join symbols.
- Conditional behaviors are shown by branch and merge symbols. These two symbols are optional. Guards can be used on a transition, as in a State diagram.
- A * indicates multiple iterations.

- Partitions are used to convey the object / person / department / etc responsible for that action.
- Objects may be connected to Actions. The action to which the object is connected has a direct effect on the object's state or lifetime. An arrow direction on the dotted line indicates whether the object is input or output.

This diagram should be used for

- Analyzing interactions related to one or more use cases.
- Understanding workflow.
- Describing a complicated sequential algorithm
- Modeling Multi-threaded operations.

This diagram should not be used for

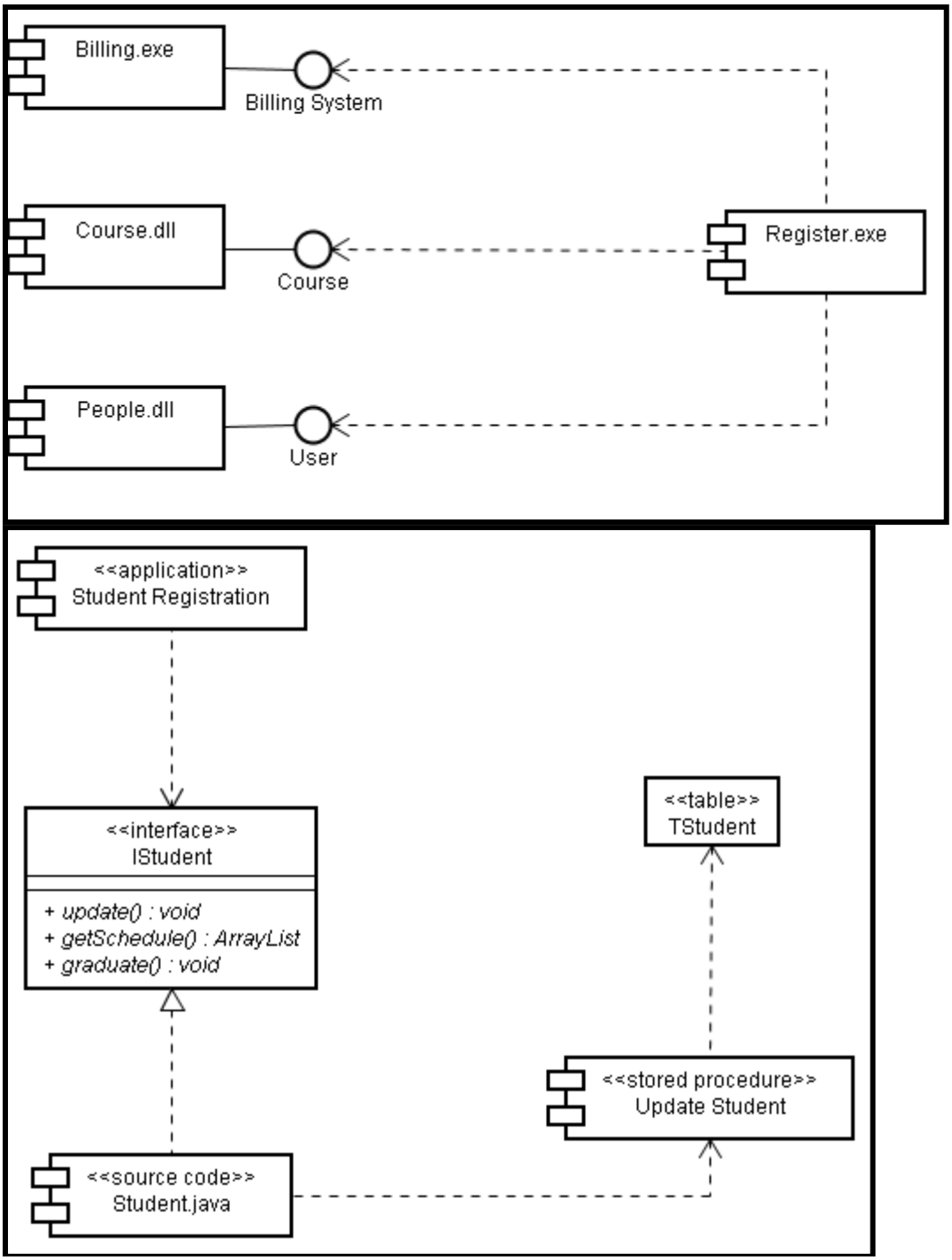
- To see how objects collaborate. For this purpose, we have interaction diagrams.
- To see how an object behaves over its lifetime. For this purpose, we have state diagram.
- Representing complex conditional logic. For this purpose, we can use a simple truth table.

Guidelines

- Avoid Partitions. Use interaction diagram to assign responsibility.
- Keep the start point at the top left corner.
- If the activity diagram does not fit on one page, consider refactoring.
- If an action has transitions into it but no transition out of it, the designer has missed something.
- If an action has transitions out of it but no transition into it, the designer has missed something.
- Avoid superfluous decision points.
- Every transition leaving a decision point must have a guard.
- Guards should not overlap.
- A [otherwise] guard is used to catch all remaining conditions.

This diagram is the UML equivalent of a data flow diagram.
This diagram is usually retained to provide a high-level overview of the logic for a business process, but its utility value seems low in the long term.

Component Diagram



Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy.

This diagram is used for

- Logical business architecture modeling
- Physical architectural modeling of a component-based software system

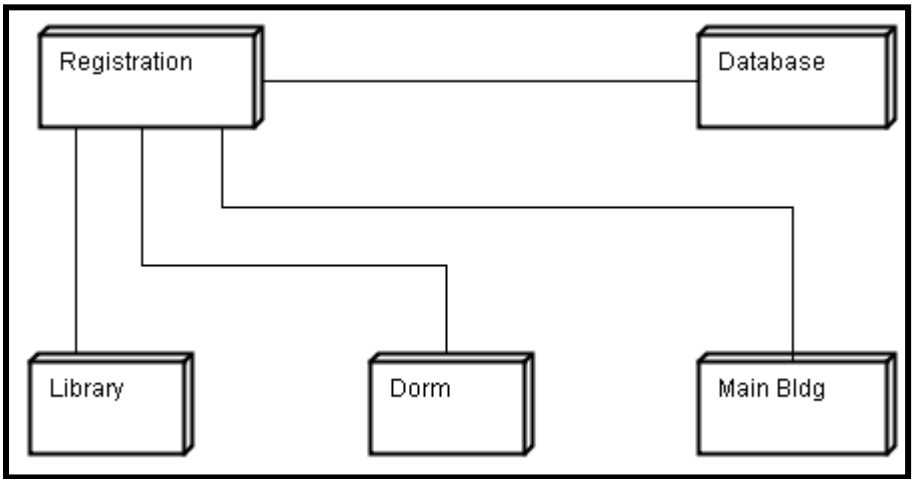
The diagram is often kept and maintained to depict a high level architecture of the system. The importance of this diagram is high in a large system.

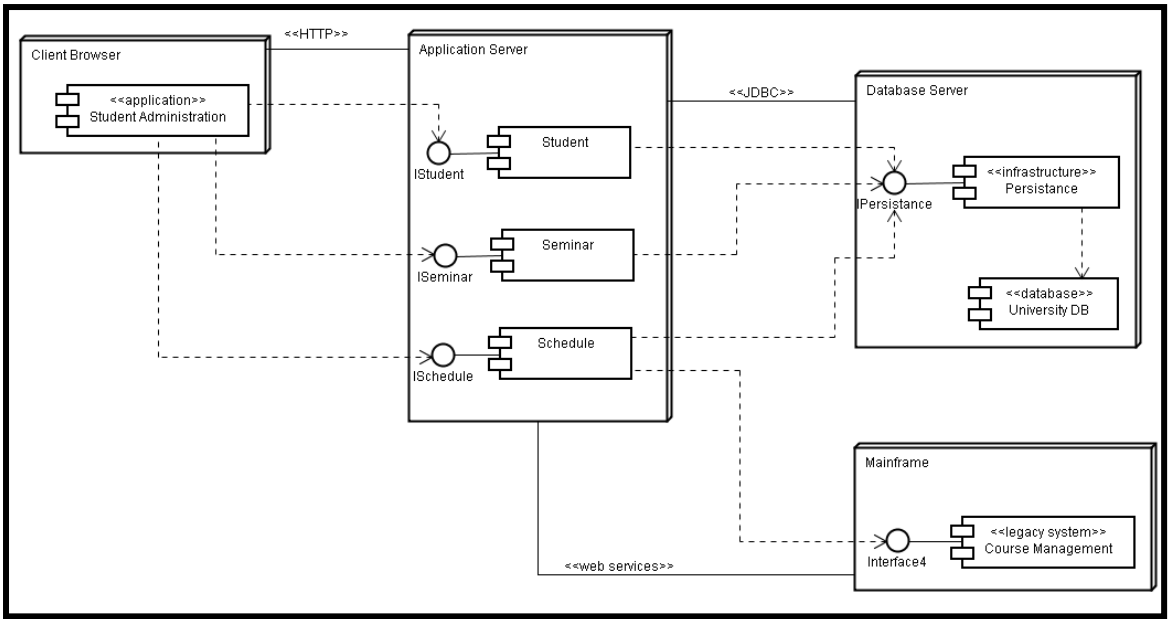
Guidelines:

- Use Descriptive Names for architectural components.
- Use environmental specific names on components e.g. xyz.dll, abc.java, user manual, etc.
- Common stereotypes used in a component diagram are application, database, document, executable, file, infrastructure, library, source code, table, web service, XML DTD.
- Do not use this diagram for database design.
- Do not use this diagram for compilation dependency among source files.
- Interfaces are connected to components by lollipop notation or a dashed line with a closed arrowhead. The lollipop notation is preferable.
- Show only relevant interfaces.
- Components can be dependent on other components, but this is poor design. Components should be dependent upon the interface of other components.
- Draw dependencies from left to right.
- Identify cycles of dependency and try to eliminate them.

A component diagram shows the various components in a system and their dependencies. A component may have more than one interface.

Deployment Diagram





Both Component and Deployment diagrams are used to show physical layout of the system. This diagram depicts a static view of the run-time configuration of hardware nodes and the software components that run on those nodes. Deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

Deployment Diagram.

- A deployment diagram shows the physical relationships among software and hardware components in the system.
- Each node on a deployment diagram represents some kind of computational unit in most cases a piece of hardware.
- Connections among nodes show the communication paths over which the system will interact.
- It probably has the maximum impact on the performance and throughput of the system.

We create a deployment model to:

- Explore the issues involved with installing your system into production.
- Explore the dependencies that your system has with other systems that are currently in, or planned for, your production environment.
- Depict a major deployment configuration of a business application.
- Design the hardware and software configuration of an embedded system.
- Depict the hardware/network infrastructure of an organization.

This diagram shows how components and objects are present in a distributed system.

A node is computational unit e.g. PC, Server, etc. A node hosts a set of components. Connections are communication paths between nodes.

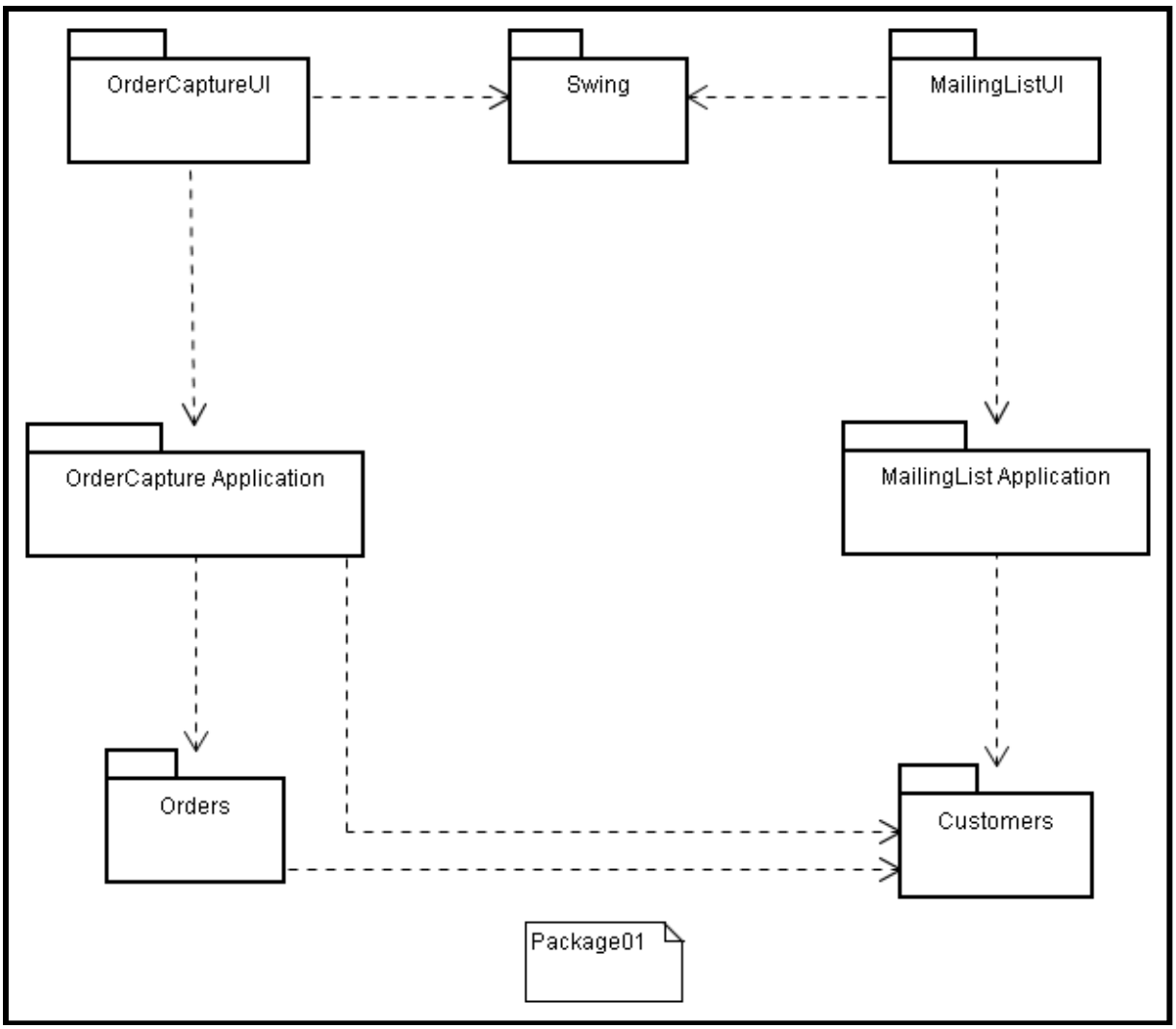
Guidelines:

- Each node should have a simple name
- To avoid clutter in the diagram, we may not show components in this diagram.

- Indicate communication protocols with stereotypes like asynchronous, HTTP, JDBC, ODBC, RMI, RPC, Synchronous, web services.
- Systems that live entirely within one computer and interact with all its hardware devices via the operating system have no use for deployment diagrams.

This diagram serves as documentation of technical architecture of the system. Its utility value in the long term ranges from low to medium.

Package diagram



It is a mechanism to split a large system into smaller ones. The smaller systems are easier to build and understand. Packages are called namespaces in C++.

A Package diagram shows packages of classes and dependencies between packages. If a change to Package A can cause a change to Package B, then Package B is dependent on package A. Dependency is shown by a dotted arrow line.
Ideally other packages should be affected only when the package changes its interface and not otherwise. A good design attempts to minimize dependency between packages.

A package diagram is a UML diagram composed only of packages and the dependencies between them. A package is a UML construct that enables you to organize model elements, such as use cases or classes, into groups. Packages are represented by tabbed boxes and can be applied on any UML diagram.

Create a package diagram to:

- Depict a high-level overview of your requirements (overviewing a collection of UML Use Case diagrams).
- Depict a high-level overview of your design (overviewing a collection of UML Class diagrams).
- To logically modularize a complex diagram.
- To organize the source code.

Guidelines

- Each package should have a simple, descriptive name.
- Divide & Conquer. If a diagram cannot be printed on one page, it is too large. Break it down.
- Packages should be highly cohesion and low coupling.
- Keep all user interfaces in separate classes. Keep all user interface classes in one or more package. Use a stereotype <<user interface>> to indicate this.
- Keep all database access in separate classes. Keep all database access classes in one or more package. Use a stereotype <<database>> to indicate this.
- Keep all the business logic in separate classes. Keep all business logic classes in one or more package. Use a stereotype <<domain>> to indicate this.
- No cycles in a package diagram.
- Classes in the same inheritance hierarchy usually belong to the same package. Classes related to one another via aggregation / composition usually belong to the same package. Classes that communicate a lot among themselves usually also belong to the same package.
- Use Facades. Here, all classes, attributes and methods are given visibility private (preferably) or package (default of Java). Now we add extra public classes, attributes and methods to allow interface with this package. These extra classes, called Facades, delegate public operations to their shyer companions in the same package.
- A package can contain other packages. This hierarchy should typically not be extended beyond 2 levels as it becomes difficult to comprehend.

A package diagram for a system is usually retained as long as the system is in use.

When using OOAD artifacts to organize and assign team responsibilities on a project, it is best to divide teams according to package diagram dependencies and utilize use cases to schedule the work for the individual team members.

Other UML Diagrams

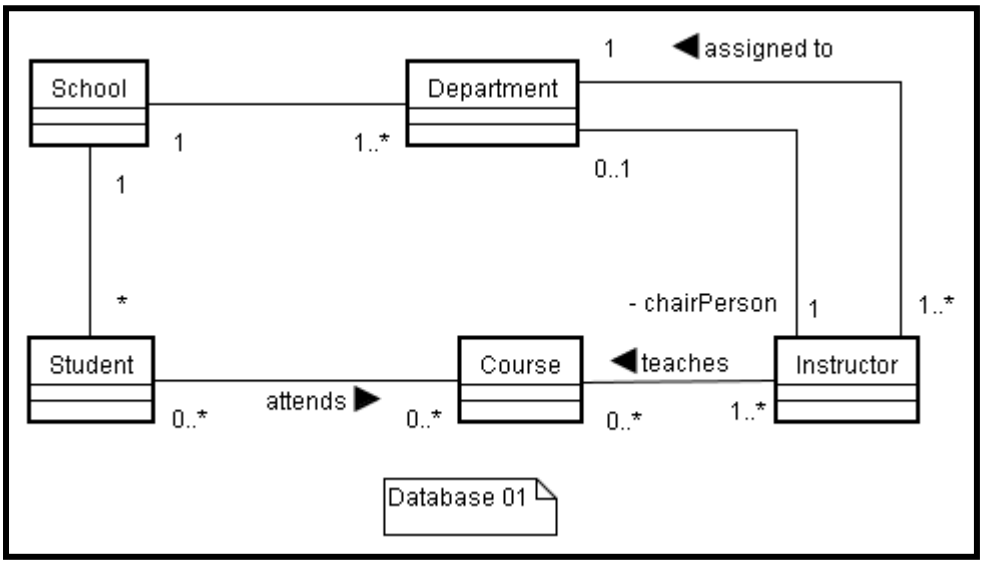
These diagrams were added in UML 2.0. They are yet to become popular.

1. Composite Structure diagram: Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system.

2. Interaction overview diagram: A variant of an activity diagram which overviews the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram.

3. Timing diagram: Depicts the change in state or condition of a classifier instance or role over time. Typically used to show the change in state of an object over time in response to external events.

Database Modeling



Data model is not a part of UML. A class diagram is used for this purpose. Each table is represented like a class.
This diagram is used exploring relationships between a handful of tables or for building a conceptual model of the database.
This diagram should not be used to determine the class structure.
This diagram serves as a document of physical design. It value is very high. It used heavily and is maintained of the project.

One to one relationship and one-to-many relationships can be implemented with a foreign key. Same for aggregation & composition relationships. For Many to many relationship, a new table will be needed to store the primary key of both the tables.

Software Development Process

Rule 1: *People are more important than any process.*

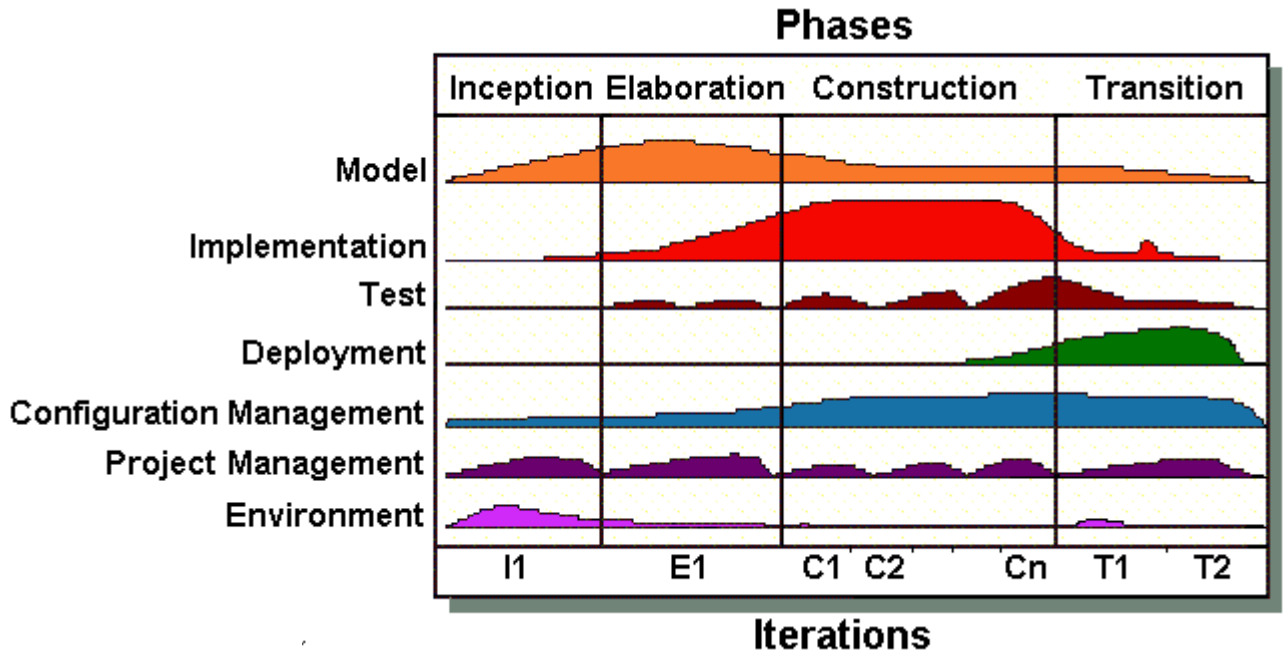
Rule 2: *Good people with a good process will usually outperform good people with no process.*
Rule 3: *Rule 1 should always be applied before Rule 2.*

No model has any value other than to assist in object thinking and to provide a means for interpersonal communication.

> *If you can model your objects and your scenarios in your head while engaged in writing code, and if those mental models are consistent with object thinking, great! No need to write them down.*
>
> *If you and your colleagues use a visual model on a whiteboard as an aid in talking about scenarios and in clarifying your collective thinking about those scenarios, and you erase the board when you're done meeting, also great!*
>
> *If your models are crudely drawn and use only a subset of the syntax defined here (or a completely different syntax that you and your colleagues collectively agree upon), still great!*

Model when you must, what you must, and only what you must.

A project can have 4 phases: inception, elaboration, construction and transition.



Inception
Inception implies approximate vision, business case, scope and vague estimates. Here we envision the product scope, vision and business case. The main problem to be answered here is "Do the stakeholders have a basic agreement on the vision of the project and is it worth

serious investigation?" An approximate estimate is made whether the project will cost $100K or $1 million; whether the project is 10 person years or 100 person years; etc.
Inception is less than 1 week for most projects.

Elaboration

Here

- The majority of requirements are discovered and stabilized.
- The major risks are mitigated or retired.
- The core architectural elements are implemented and proven. This is not a throw-away prototype but a portion of the final system.
- More realistic estimates are done.

A project should have a systematic approach to finding, documenting, organizing and tracking the changing requirements of the system.

Elaboration usually has 2 to 4 iterations with each iteration usually of 2 to 6 weeks. In this phase,

- Build the core architecture, resolve the high-risk elements, define most requirements and estimate the overall schedule and resources.
- Adaptively design, implement and test the core and risky parts of the architecture during early iterations.
- Programming and testing should be started early.
- Adapt based on feedback from test, users and developers.
- Elaboration has heavy use case analysis.
- All use cases should *not* be written in detail. Only basic information about each use case should be available. The information should be just enough to know what this use case is and what its priority is.
- Elaboration is complete when all the significant risks have been identified and the major ones are understood to the extent that we know how we intend to deal with them.

Common *misunderstandings* about elaboration are

- It should be a few months for most projects.
- It should have only one iteration.
- Most requirements are defined before elaboration.
- Risky elements and core architecture are not tackled.
- There is no executable production-code written in this phase.
- It is primarily a requirements phase or design phase of a waterfall model.
- There is an attempt to do a full and careful design before programming.
- The SMEs are not engaged in continuous evaluation and feedback.
- There is no early and realistic testing.
- The architecture is finalized before programming.
- It is considered to be the proof-of-concept programming rather than programming the production core executable architecture.
- Requirements are not iteratively refined based on feedback from prior and current iterations.

Construction
Construction implies Iterative implementation of the remaining lower risk and easier elements and preparation of deployment.

We maintain a stack of use cases. The use cases at the top have the highest priority and the use cases at the bottom have the lowest priority. We always work on the use cases with high priority.

A study conducted in 2005 found that (on an average) 66% of features, in a software application developed with predictive model, are not used. Concentration on high priority functionality automatically avoids this error.

It usually has many Design, Code and Test iterations. Each iteration is a short, time boxed and risk-driven mini project whose outcome is a tested, integrated and executable system. An iteration length is recommended to be between two and eight weeks. Larger the team size developing software, longer is the iteration length.
If it seems that it will be difficult to meet an iteration deadline, the recommended response is to remove tasks or requirements to a future iteration; date slippage is discouraged.
The output of iteration is not an experimental or throw-away prototype. Iterative development is not prototyping. The output of iteration is a production grade subset of the final system. The system may not be eligible for production deployment until after many iterations.
Although each iteration tackles new requirements and incrementally extends the system, it usually revisits existing software and refactors it.

Here,

- Early feedback is more important than speculating on the correct requirements and design.
- Iteration, in general, tackles new requirements and incrementally extends the system. Occasionally, iteration revisits existing software and improves it.
- Change is welcomed in iterative development. This is reverse of waterfall methodology.
- Tackle the high-risk and high-value issues in early iterations.

Transition
Transition implies Beta tests, deployment.

- Acceptance tests, deployment, performance tuning and user training.
- Optimization (performance improvement) should be left always at the end.
- During transition there is no development to add functionality unless is small or absolutely essential.

This phase may overlap with construction phase.

Documentation

When to draw diagrams?

- Several people need to understand the structure of a particular part of the design because they are all going to be working on it simultaneously. Stop when everyone agrees that they understand.
- You want team consensus, but two or more people disagree on how a particular element should be designed. Put the discussion into a time box, and then choose a means for deciding, such as a vote or an impartial judge. Stop at the end of the time box or when the decision can be made. Then erase the diagram.
- You want to play with a design idea, and the diagrams can help you think it through. Stop when you can finish your thinking in code. Discard the diagrams.
- You need to explain the structure of some part of the code to someone else or to yourself. Stop when the explanation would be better done by looking at code.
- It's close to the end of the project, and your customer has requested them as part of a documentation stream for others.

When not to draw diagrams?

- Because the process tells you to.
- Because you feel guilty not drawing them or because you think that's what good designers do. Good designers write code. They draw diagrams only when necessary.
- To create comprehensive documentation of the design phase prior to coding. Such documents are almost never worth anything and consume immense amounts of time.
- For other people to code. True software architects participate in the coding of their designs.

What about Case tools?

UML CASE tools can be beneficial but also expensive dust collectors. Be very careful about making a decision to purchase and deploy a UML CASE tool.

- Don't UML CASE tools make it easier to draw diagrams? No, they make it significantly more difficult. There is a long learning curve to get proficient, and even then the tools are more cumbersome than whiteboards, which are very easy to use.
- Don't UML CASE tools make it easier for large teams to collaborate on diagrams? In some cases. However, the vast majority of developers and development projects do not need to be producing diagrams in such quantities and complexities that they require an automated collaborative system to coordinate their diagramming activities. In any case, the best time to purchase a system to coordinate the preparation of UML diagrams is when a manual system has first been put in place, is starting to show the strain, and the only choice is to automate.
- Don't UML CASE tools make it easier to generate code? The sum total effort involved in creating the diagrams, generating the code, and then using the generated code is not likely to be less than the cost of simply writing the code in the first place. If there is a gain, it is not an order of magnitude or even a factor of 2. Developers know how to edit text files and use IDEs. Generating code from diagrams may sound

like a good idea, but I strongly urge you to measure the productivity increase before you spend a lot of money.
- What about these CASE tools those are also IDEs and show the code and diagrams together? These tools are definitely cool. However, the constant presence of UML is not important. The fact that the diagram changes as I modify the code or that the code changes as I modify the diagram does not really help me much. Frankly, I'd rather buy an IDE that has put its effort into figuring out how to help me manipulate my programs rather than my diagrams. Again, measure productivity improvement before making a huge monetary commitment.

In short, look before you leap, and look very hard. There may be a benefit to outfitting your team with an expensive CASE tool, but verify that benefit with your own experiments before buying something that could very well turn into shelf ware.

Facts about documentation

- Documentation isn't the primary issue—communication is. For example, your primary goal isn't to document requirements; it's to understand them so that you can implement them fully. Your primary goal isn't to document the architecture; it's to formulate a viable one that meets the long-term needs of your stakeholders. This isn't to say that you won't decide to invest in such documentation, but that isn't the reason for your project to exist.
- Comprehensive documentation does not ensure project success. Just because you have a detailed requirements specification that has been reviewed and signed off, that doesn't mean that the development team will read it, or if they do, that they will understand it, or if they do, that they will choose to work to the specification. Even when the "right thing" happens, comprehensive documentation still puts the project at risk because of the perceived need to keep the documentation up-to-date and consistent. Your real goal should be to create just enough documentation for the situation at hand; more on this later.
- Documentation doesn't need to be perfect. People are flexible, they don't need perfect documentation. Furthermore, no matter how hard you work, it's very unlikely that you could create perfect documentation—even if you could, it isn't guaranteed that the readers will understand it anyway.
- Few people actually want comprehensive documentation. When was the last time you met a maintenance programmer who trusted the system documentation that they were provided? Or an end-user who wanted to sit down and pore over a massive user manual? What people typically want is well-written, concise documentation describing a high-quality system, so why not invest your time to deliver that instead?
- Documentation doesn't need to be standardized. Many organizations have adopted the unfortunate philosophy that repeatable processes lead to success, yet in reality what you really want is repeatable results (that is, the delivery of a high-quality system). The quest for repeatability often leads to templates, and because each system has its own unique set of issues, these templates have a tendency to grow over time into all inclusive monstrosities that do little more than justify bureaucracy. Try to achieve a reasonable level of consistency, but don't go overboard doing so.

How much documentation?

Good documentation is essential to any project. Without it, the team will get lost in a sea of code. On the other hand, too much documentation of the wrong kind is worse because you have all this distracting and misleading paper, and you still have the sea of code.

Documentation must be created, but it must be created prudently. The choice of what not to document is just as important as the choice of what to document. A complex communication protocol needs to be documented. A complex relational schema needs to be documented. A complex reusable framework needs to be documented. However, none of these things need a hundred pages of UML. Software documentation should be short and to the point. The value of a software document is inversely proportional to its size.

For a project team of 12 people working on a project of a million lines of code, I would have a total of 25 to 200 pages of persistent documentation, with my preference being for the smaller. These documents would include UML diagrams of the high-level structure of the important modules, ER (Entity-Relationship) diagrams of the relational schema, a page or two about how to build the system, testing instructions, source code control instructions, and so forth. I would put this documentation into a wiki or some collaborative authoring tool so that anyone on the team can access it on the screen and search it and change it as need be.

It takes a lot of work to make a document small, but that work is worth it. People will read small documents. They won't read 1,000-page tomes.

When to Document?

Documentation should be created on a just-in-time (JIT) manner when you need it, and only if you need it. You should document something only when it has stabilized, otherwise you will be updating it endlessly as the situation changes. This is one of several reasons why it proves foolish to write comprehensive requirements documentation early in the system lifecycle—because your stakeholders will change their minds, usually for completely valid reasons, you discover that the investment in detailed documentation was counter-productive. The implication is that detailed documentation should be written after, or at least towards the end of, the development work. Yes, you likely need to do some very high-level modeling to think things through, but investing in detailed documentation too early is questionable at best.

You should update documentation only "when it hurts". People are flexible, if the documentation isn't perfect that's okay, they can figure it out. Millions of software systems are successfully running around the world as you read this paragraph, and how many do you honestly think have perfect documentation that is up to date and fully consistent? Perhaps a handful? How many billions of dollars, if not trillions, do you think has been wasted over the decades striving to produce perfect documentation? Your goal should be to produce documentation that is good enough for the situation at hand—once it's good enough any more investment in it is clearly a waste.

Documentation Alternatives

Developers have rediscovered the concept of literate programming, of writing high-quality, easy-to-understand source code that contains embedded documentation. Furthermore, teams consider their test suites to be executable, detailed documentation. Automated acceptance test suites forms a significant portion of the requirements documentation and developer test suites the design documentation. We follow Single Source Information practice by having our tests do double-duty as specifications and therefore we require significantly less external documentation.

The "documentation is in the source code" philosophy covers the vast majority of technical documentation, but system overviews will still be required as will documentation for non-development staff. Good documentation is concise—it overviews critical information such as design decisions and typically provides visual "roadmaps" showing the high-level relationships between things. Developers should focus on this style of documentation, and only write detailed documentation when it is needed and no better alternative exists.

Good Software developers travel as light as we possibly can, creating just enough documentation for the situation at hand in a just-in-time (JIT) manner. We believe that the benefit of having documentation must be greater than its total cost of ownership (TCO), that our stakeholders should decide whether to invest their money in it, and that we should strive to maximize the return on investment (ROI).

Exercises

Make suitable assumptions on a need basis.

Exercise 1 – Address Book

The software to be designed is a program that can be used to maintain an address book. An address book holds a collection of entries, each recording a person's first and last names, address, city, state, zip, and phone number.

It must be possible to add a new person to an address book, to edit existing information about a person (except the person's name), and to delete a person. It must be possible to sort the entries in the address book alphabetically by last name (with ties broken by first name if necessary), or by ZIP code (with ties broken by name if necessary). It must be possible to print out all the entries in the address book in "mailing label" format.

It must be possible to create a new address book, to open a disk file containing an existing address book to close an address book, and to save an address book to a disk file, using standard New, Open, Close, Save and Save As ... File menu options. The program's File menu will also have a Quit option to allow closing all open address books and terminating the program.

The initial requirements call for the program to only be able to work with a single address book at a time; therefore, if the user chooses the New or Open menu option, any current address book will be closed before creating/opening a new one. A later extension might allow for multiple address books to be open, each with its own window which can be closed separately, with closing the last open window resulting in terminating the program. In this

case, New and Open will result in creating a new window, without affecting the current window.

The program will keep track of whether any changes have been made to an address book since it was last saved, and will offer the user the opportunity to save changes when an address book is closed either explicitly or as a result of choosing to create/open another or to quit the program.

The program will keep track of the file that the current address book was read from or most recently saved to, will display the file's name as the title of the main window, and will use that file when executing the Save option. When a New address book is initially created, its window will be titled "Untitled", and a Save operation will be converted to Save As ... - i.e. the user will be required to specify a file.

Draw the

- Use case diagram
- Conceptual Class Diagram
- Software Class Diagram. Mention the design patterns that were used.

Exercise 2 - The Universal Vending Machine Project

You are to head a team building the software for a “better vending machine.” Vending machine sales are flat, but your CEO notes that in Japan almost anything can be purchased from vending machines, and in Scandinavia people can buy products from machines using their cell phones.
The hardware side of the company is busy designing and building customized vending machines of all types, and your team is to develop a common software base to run every type of vending machine they might come up with.
The Universal Vending Machine (UVM) will be capable of dispensing liquids as well as packages (cans or products in wrappers of various kinds). Payment can be made using any or all of three kinds of currency and coins (U.S. dollars, euros, and yen), debit cards (including a line of prepaid debit cards sold by your company), and credit cards.
Customers can purchase goods from the machine via a Web-based transaction—also using debit and credit cards.
Each vending machine will initiate product reordering via the Web. Restocking will be adjusted based on demand—which products sell the most in the least amount of time.

To save money, your team will create a single program, one capable of supporting the entire line of planned UVMs.
Draw the

- Use case diagram
- Conceptual class diagram
- Software class diagram. Mention the patterns used.

Exercise 3 - Requirements Statement for Example ATM System

The software to be designed will control a simulated automated teller machine (ATM) having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash (in multiples of $20), a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned - except as noted below.

The ATM must be able to provide the following services to the customer:

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card, in multiples of $20.00. Approval must be obtained from the bank before cash is dispensed.
2. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.
3. A customer must be able to make a transfer of money between any two accounts linked to the card.
4. A customer must be able to make a balance inquiry of any account linked to the card.

A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and dollar amounts, but for security will *never* contain a PIN.

Draw the

- Use case diagram. Write the description of each use case in an editor.
- Conceptual class diagram
- Software class diagram. Mention the design patterns that were used.
- State diagram having following states: OFF, IDLE, SERVING_CUSTOMER
- State diagram for a customer Session having following states: ReadingCard, ReadingPin, ChoosingTransaction, PerformingTransaction, EjectingCard.
- State diagram for one typical transaction. Use the following states: GettingSpecifics, SendingToBank, HandlingInvalidPin, CompletingTransaction, PrintingReceipt and a branch condition for multiple transactions.
- Component diagram
- Sequence diagram for system startup with following objects: OperatorPanel, ATM, CashDispensor, NetworkToBank
- Sequence diagram for system shutdown with following objects: OperatorPanel, ATM, NetworkToBank
- Sequence diagram for Session with following objects: CardReader, ATM, Session, CustomerConsole, Transaction.
- Sequence diagram for a Transaction with following objects: Transaction, NetworkToBank, ReceiptPrinter, CustomerConsole, Log.
- Communication diagram for Withdrawal transaction with following objects: Withdrawal, Message, Receipt, CashDispensor, CustomerConsole.
- Communication diagram for Deposit transaction with following objects: Deposit, Message, Receipt, NetworkToBank, EnvelopeAcceptor, CustomerConsole.
- Communication diagram for Transfer transaction with following objects: Transfer, Message, Receipt, CustomerConsole
- Communication diagram for Inquiry transaction with following objects: Inquiry, Message, Receipt, CustomerConsole.
- Communication diagram for Invalid Pin processing with following objects: Transaction, Message, NetworkToBank, Session, CustomerConsole, CardReader.

Exercise 4 – HR Application

Draw the Use case diagram, class diagram, database diagram and other UML diagrams on a need basis for the following problem:

This system consists of a database of the company's employees, and their associated data, such as time cards. The system must pay all employees the correct amount, on time, by the method that they specify. Also, various deductions must be taken from their pay.

- Some employees work by the hour. They are paid an hourly rate that is one of the fields in their employee record. They submit daily time cards that record the date and the number of hours worked. If they work more than 8 hours per day, they are paid 1.5 times their normal rate for those extra hours. They are paid every Friday.
- Some employees are paid a flat salary. They are paid on the last working day of the month. Their monthly salary is one of the fields in their employee record.
- Some of the salaried employees are also paid a commission based on their sales. They submit sales receipts that record the date and the amount of the sale. Their commission rate is a field in their employee record. They are paid every other Friday.
- Employees can select their method of payment. They may have their paychecks mailed to the postal address of their choice, have their paychecks held by the paymaster for pickup, or request that their paychecks be directly deposited into the bank account of their choice.
- Some employees belong to the union. Their employee record has a field for the weekly dues rate. Their dues must be deducted from their pay. Also, the union may access service charges against individual union members from time to time. These service charges are submitted by the union on a weekly basis and must be deducted from the appropriate employee's next pay amount.
- The payroll application will run once each working day and pay the appropriate employees on that day. The system will be told what date the employees are to be paid to, so it will generate payments for records from the last time the employee was paid up to the specified date.

Draw the

- Software class diagram. Mention the design patterns used.
- Draw the important interaction diagrams
- Design the database

Exercise 5 - Home Realty System Problem Statement

As the Vice-President of Development for BC Realtors you are asked to develop a new home realty system. The company would like to create an e-development solution that will replace the home listing catalogs that are printed on a monthly basis, and to some extent, replace the need to generate sales literature for every home that is listed with BC Realtors.

The new system will allow prospective buyers to search the home database for current listings and to initiate the loan process. Buyers will also be able to communicate with the listing agent to request additional information, a home walkthrough, and so on.

Realtors will be able to list their properties on the BC Realtor system for a nominal fee. A prospective buyer will be able to log on to the system and set up a personal profile. This profile will allow the buyer to enter a set of personal preferences and search requirements.

Buyers will also be able to bookmark properties to the personal planner for easy reference the next time they log on. After a customer has logged on to the system they may choose to search for a home, find a Realtor, or apply for a mortgage loan. The customer should be able to search for a home in a geographic area by city, zip code, or the Multiple Listing Service (MLS) number. The buyer should be able to further narrow their search through a series of filter criteria until they find a number of homes they are interested in. The buyer should be able to view a picture of the home and receive a full text description on all the amenities and features that the home has to offer. Finally, if the buyer is interested in receiving more information on the home, the buyer will be able to send an e-mail to the listing broker. BC Realtors has recently invested in a new mail server, and this server should be used to send e-mails.

The prospective buyer has the option to apply for a mortgage loan using the Home Realty System. BC Realtors has an existing Loan System that communicates with a number of partner lenders to gain loan pre-qualification approvals. This system should continue to be used for sending loan requests to potential lenders. The Home Realty System will ask the prospective buyer a series of questions about their current financial standing. After the customer has answered all questions, the system will send the data to the Loan System and receive a list of possible offers for a loan. If the customer chooses to select one of the pre-qualification offers, the system will inform the customer that a credit report must be generated. BC Realtors subscribes to a Credit Reporting service, and the existing interface to this system should be used to provide this service.

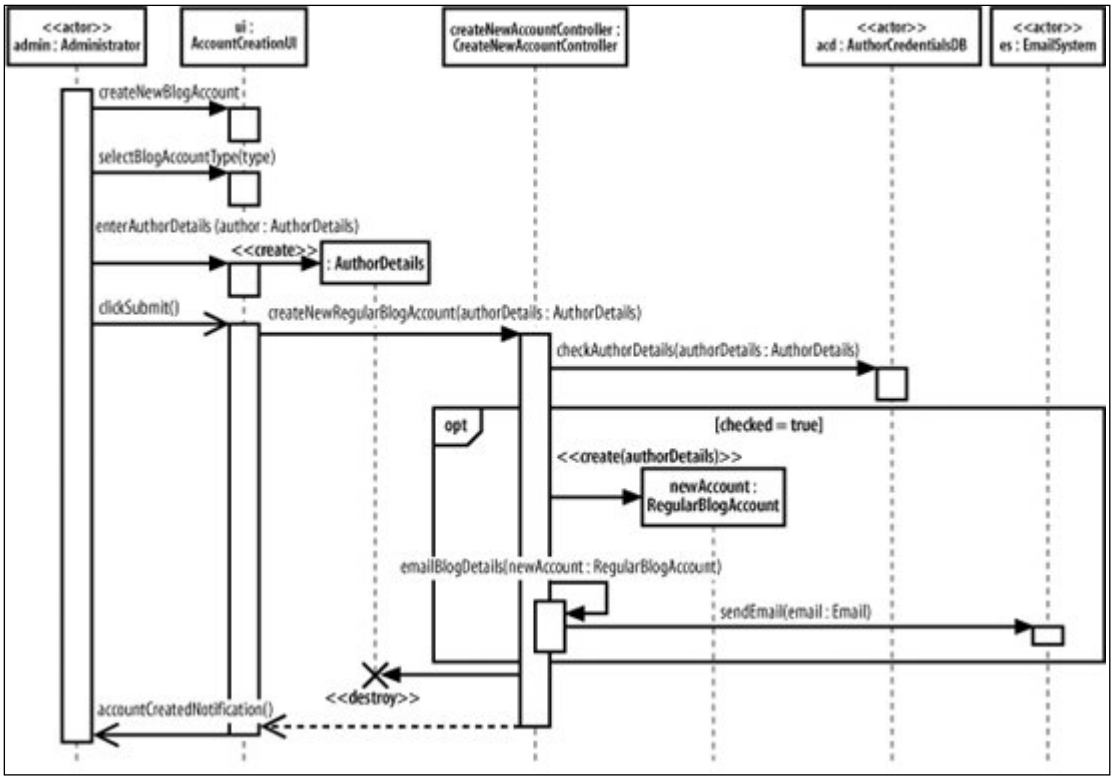
A prospective buyer may request information on any broker that is listing property with the Home Realty System. The buyer should be allowed to view the broker's personal profile that may contain any type of information that the broker enters and also a summary of all the properties that the broker currently has listed. Realtors must be able to access the on-line system to modify their personal profiles that are displayed to buyers. They will also need to create a new on-line listing for properties that they represent.

Draw the

- Use case diagram (A use case diagram is meaningless without description associated with each use case)
- Activity diagram
- Conceptual class diagram

Exercise 6 – Sequence to Communication

This sequence diagram describes the interactions that take place in a Content Management System, when a new regular blog account is created. Convert it to a communication diagram



Description of the sequence of events:

1. The Administrator asks the system to create a new blog account.
2. The Administrator selects the regular blog account type.
3. The Administrator enters the author's details.
4. The author's details are checked using the Author Credentials Database.
5. The new regular blog account is created.
6. A summary of the new blog account's details are emailed to the author.

Exercise 7 – Video Rental

Movie Corner is a video store company consisting of a number of shops, where the customer can rent movies. The company started its operation with only one shop in the year 1995.Today they have 50 shops, 2 warehouses, distributed in five cities. *All the shops are SBU's and maintain their own accounts and profitability.*

Today, 20,000 titles are available and each shop is renting out about 1,000 movies per day.

The company has offered a software development firm to develop the software with the following requirements as per the modules

Rental

> **Rent a movie:** The customer chooses a movie in the store, runs a member card through the card reader and provides the identification code (PIN). The

clerk zaps the bar codes on the movies, writes a receipt with return dates and receives the money from the customer.

Register returned movies: The clerk checks the bar code and the movie is identified. The clerk receives customer data, and is alerted if the rental time is overdue. The clerk accepts return of movie and collects the overdue fee.

Register new member and rent first movie: The customer chooses movies, but is not a member yet (has no card). If this person is OK according to our policy, he/she is registered and receives a card. The customer chooses a PIN code and can rent the chosen movies.

Block a customer: The store manager finds customer and customer data and blocks the customer, giving the reason for blocking.

Analysis

Find rental patterns: The product manager looks for information regarding rental patterns such as, when people rent films, what kind of films, did the ad campaign generate more income…

Produce Top Ten Lists: The store manager produces a list of the most popular movies for the last month. This is distributed to all offices.

Identify the movies with the best revenue: The manager chooses how the selection shall be performed, like date, store, kind of movie etc. The search is performed and the manager can look at the result as well as print it out.

Register incoming new movies: The store manager zaps the bar code of each movie and adds the number of copies of each movie. The inventory for the office is updated and the movies are made available for renting.

Stock Management

Take stock: The store manager checks the inventory by printing a list and checking it against the physical stock. The on-line inventory is updated according to the result.

Order movie from another office: The clerk finds the movie at another store and places an order to get it to the office/store. The clerk receives a receipt with confirmation of order and expected delivery date.

Administration

Manage users: The administrator can add, update and delete users and assign appropriate privileges to the users.

Backup: The administrator can take back of the database either manually or schedule the run either on weekly or monthly basis.

Users can access the respective functionalities with appropriate login.

References for UML

1. UML Distilled – Martin Fowler
2. Learning UML - Kim Hamilton, Russell Miles
3. UML for Java Programmers – Robert Martin
4. http://www.agilemodeling.com

An abundance of words often conceals a lack of ideas. I have tried to keep this document short. Hope you enjoy it. If possible, please send feedback to vijay_nathani@yahoo.com.