Objectives:

1. To understand the place of analysis in the overall software development process
2. To understand the distinction between domain and application analysis.
3. To become familiar with selected analysis tools
   a. Use cases / use case diagrams / use case flows of events
   b. Analysis class diagrams

Materials:

1. Transparency: Lethbridge and Laganiere Figure 4.2
2. Video projection: ATM example on the web
3. Handout: ATM requirements
4. Handout: ATM use case diagram
5. Handout: ATM use case flows of events - partial (session, transaction, withdrawal, invalid PIN extension)
6. Transparency of Examples 7.2, 7.4 in Lethbridge
7. Transparencies of Larman pp. 50-53 (excerpts) + 53 two-column variant
8. Transparency of Example 7.5 in Lethbridge
9. Handout: ATM analysis class diagram and discussion

I. Introduction

A. As we pointed out at the start of the course, there are many different lifecycle models that can be used in software development.

B. Regardless of what development model is followed, of course, certain tasks will need to be done as part of the development process per se - whether all at once, iteratively, or incrementally.

1. Analysis. The goal of this task is to understand the problem.

2. Design. The goal of this task is to develop the overall structure of a solution to the problem in terms of individual, buildable components and their relationships to one another.

3. Implementation. The goal of this task is to actually build the system as designed.

4. Quality Assurance. The goal of this task is to ensure that the individual components and the system as a whole do what they are supposed to do (which involves identifying their shortcomings and fixing them.)
5. Deployment and Maintenance - using the software and making changes as necessary.

C. Today’s focus is the first of these tasks: analysis. The goal of the analysis phase of software development is a thorough understanding of the problem.

1. Often a software project begins with a general statement of the requirements, which may be more or less detailed. However, the wise developer does not begin designing a solution until he/she thoroughly understands both the general domain of the problem and the specific application that needs to be developed. We call the process of developing this understanding analysis.

2. Obviously, this must take place in the mind of the analyst(s); but it also must be documented in some form, especially if - as is often the case - people other than the analyst(s) are involved in later phases of the project.

3. Typically, this documentation takes the form of one or more models of the system being analyzed.

   a) A model is simply a representation of a system.

   b) For complex problems, more than one model is often needed - each focusing on a different aspect of the system.

   c) There are a large number of modeling tools that can be used for doing analysis - some more appropriate to a particular problem than others. We will learn a few modeling tools, and try to give you some experience using them. One thing that you will need to develop over time is the ability to make appropriate decisions as to which tools to use for a particular problem.

   Example: A carpenter learns to use a wide variety of tools, though he may need only a few for any particular project.

D. It turns out that there are two distinct ways to approach analysis. For a complex problem, each of these approaches sheds light on a different aspect of the problem. Together, they give a complete picture.

To illustrate these different types of analysis, we will use the example of developing software for a controller for traffic lights that are to be activated by sensors embedded in the road.
1. **Domain analysis** is concerned with understanding the overall domain of which our specific problem is an instance.

   a) *Example:* In the domain of traffic lights, there are things we need to know before we ever undertake to construct a traffic light system - e.g.

   (1) Traffic lights can give several different indications:

       go (typically a green light)
       stop (typically a red light)
       prepare to stop (typically a yellow light)
       go - but only in a specific direction (typically, a green arrow)
       do not go in a specific direction (typically a red arrow)
       pedestrians can walk (a walk light, or (rare now) red + yellow)
       pedestrians may not walk

       Typically, only a subset of these will be used at any particular intersection - but a person who designs traffic light systems needs to be aware of all of the options that can be used if needed

   (2) Vehicle drivers are normally expected to behave in certain ways when confronted with certain indications - e.g. a vehicle driver who sees a “stop” indication must stop. (At least in the 49 states outside of Massachusetts :-) ) However, emergency vehicles sounding their sirens are allowed to ignore a stop indication.

   (3) Traffic light indications must satisfy certain consistency constraints - e.g.

       Normally, it is permissible to give a go indication for opposite directions of travel on the same street.

       However, when two streets cross, if one has a go indication the other must have a stop indication.

   (4) Of course, the domain of traffic lights is familiar to us - which is why I’ve used it for this example. Even so, there are crucial issues that need to be understood before further work is done - e.g. the relationship between light placement, anticipated vehicle speed, and timing of the lights. In the case where a software system is being developed for an unfamiliar domain, a great deal of effort may need to be devoted to domain analysis before undertaking the specific problem.
b) Work done in performing domain analysis is not specific to any one problem - and hence can be reused for any system that belongs to that particular domain. Work done in performing domain analysis also helps ensure that each application fits in a broader context of other applications dealing with the same domain.

c) One of the goals in performing domain analysis is to discover the “business rules” the rules that any properly-functioning system in that domain must conform to.

d) The text gives an example of how a “Domain Analysis Document” might be structured. Whether or not this actual outline is used, the points in it indicate some of the kinds of issues that need to be explored in domain analysis:

(1) Glossary - terminology that is unique to the domain, or that has a special meaning in the domain. (E.g. the term “strike” means something very different in the domain of baseball than it does in the domain of labor relations - or maybe not!)

(2) General knowledge about the domain, including relevant statistics.

(3) Customers and users.

(4) The environment.

(5) Tasks and procedures currently performed.

(6) Competing software.

(7) Similarities across domains and organizations.

e) Note that, in doing domain analysis, it is often critical to interact with people who actually work in the domain, not just with “software people”.

2. **Application analysis** is concerned with the specific functionality needed by the system being developed.

   a) *Example*: For a traffic light system:

   What is the required relationship between the state of the sensors and the sequencing of the lights at a particular intersection?
Are pushbuttons needed to activate the walk lights?

What mechanisms are needed to allow manual control of the system under special circumstances?

b) An important part of application analysis is determining the scope of a particular project - i.e. what does it include? what does it not include?

TRANSPARENCY: Lethbridge and Langanier figure 4.2

c) Application analysis is concerned with the specific requirements for a particular software application within a broader domain. There are two aspects to these requirements.

(1) Functional requirements - what specific things must the software do?

(2) Nonfunctional requirements - other requirements which, while not functional in nature, are also important.

(3) Example:

(a) For a traffic light system, the requirement that the lights display a “walk” indication some time after a pedestrian pushes the walk button is a functional requirement.

(b) The requirement that, in the case of system failure, all lights display a “safe” indication (e.g. blinking yellow) is a nonfunctional requirement. A light system that would allow green to be displayed simultaneously in two conflicting directions would not be acceptable!

3. A key reason for distinguishing these two forms of analysis is to facilitate reuse: to the extent that our design is based on a general understanding of a particular domain, as opposed to the specifics of a particular application, components of our system are likely to be reusable in subsequent applications dealing with the same domain.

a) Example: If, as a result of doing domain analysis, we were to develop a general component that models a traffic light, we might be able to use it in many different applications. Thus, even if our particular application only called for the standard three round colors, it would be beneficial to allow for these other possibilities in
the component of the system that model traffic lights to facilitate possible later reuse in another application.

b) There is an emerging domain specific software component industry, in which companies are creating and marketing reusable components that deliver functionality typically required by applications in a particular domain. (“Enterprise Java Beans” are an example of this.)

4. We will not discuss domain analysis in any detail in this course.

E. Over the years, software engineers have developed a large array of analysis tools that can be used for modeling a system.

1. Each tool is useful for modeling certain aspects of a system; and different sets of tools will be appropriate for different problems. Frequently, in fact, several different tools will be utilized to capture different aspects of a given problem.

2. Some of these tools are discussed in the software engineering course or other courses

3. There are also numerous publications and workshops that deal with various tools.

4. In this lecture we will consider two such tools:

   a) Use cases

   b) Analysis class diagrams.

5. As we shall see, these tools are not only used for analysis, but will also serve to drive part of the design process. There is a certain “seamlessness” to OO methodologies in this regard.

F. For a number of our examples in the next few lectures, we will use the continuing example of developing software to simulate an automated teller machine (ATM).

1. HANDOUT ATM REQUIREMENTS

   Note: The requirements in this example are somewhat more detailed and specific than would typically be available at the outset of a project. Normally, much of the knowledge we need at this point comes a combination of studying the domain and conversations with the client.
This example document embodies some of the fruit of what would typically come from such study.

2. **DEMONSTRATION**: Example system on the web

   a) Run it

   b) Show structure of pages

II. **Use Cases**

A. An approach to analysis that is particularly useful for understanding functional requirements: the *use case* approach developed by Ivar Jacobson. We begin by identifying the use cases for the system.

1. The basic idea here is to look at the system requirements in order to identify two things:

   a) Actors - people (or sometimes other systems) that must use the system that is to be built. Note that the same person may need to be regarded as two or more actors if he relates to the system in two or more roles.

   (1) Actors may be classified as primary and secondary. The primary actor(s) are the actors for whom the system is built. Secondary actors are other actors who must also make use of the system.

   (2) Sometimes, we will also discover passive actors - external systems (or, more rarely, individuals) that the system being built makes use of to do its job, rather than vice versa.

   b) Use cases - ways in which the actors use the system

2. Example: ATM System

   a) Actors:

   **ASK**

   (1) Primary: customer

   (2) Secondary: operator. (We consider the operator to be a secondary actor because we wouldn’t build the system simply to give the operator a job to do!)
(Note: the same person may be a customer at one time and an operator at another - but the roles are distinct, so they are considered two different actors)

(3) There is also a passive actor - the Bank with to whom the ATM is connected. (But since it is passive, it is not responsible for any use cases.)

b) Use cases:

**ASK**

(1) Customer: overall session, depositing money, withdrawing money, transferring money, balance inquiry

(2) Operator: starting and stopping the system, plus various tasks performed “off line” (e.g. filling the cash dispenser and printer). (Note: we don’t need to analyze these off-line tasks further, because they’re outside the scope of the software we are developing.)

B. We now represent the use cases involved in a system by drawing one or more *use case diagrams*. (Multiple diagrams are useful for large systems where it is useful to group the use cases into categories).

1. Use case diagrams use two basic symbols.

   a) The symbol for an actor

      ![Actor Symbol]

   b) The symbol for a use case

      ![Use Case Symbol]

2. Actors and use cases are connected by lines. Each use case serves a particular type of actor.

   **HANDOUT: USE CASE DIAGRAM FOR ATM SYSTEM**

3. Use case diagrams can also depict relationships between use cases. There are three general ways in which use cases can be related to each other
a) The generalization relationship is used when one use case generalizes several similar use cases. (For example, in the ATM system the transaction use case is a generalization of the various specific types of transaction: cash withdrawal, deposit, etc.)

Generalization is denoted by the use of the “isa” triangle.

b) The « include » relationship is used when one use case is contained within another use case. (For example, in the ATM system the use cases representing individual transactions are included within the use case for an overall customer session - a customer can perform any number of transactions during the course of one session.)

c) The « extend » relationship is used when one use case may be extended (under some circumstances) by another. (For example, in the ATM system the use case for handling the reentry of an invalid PIN extends the basic transaction use case.)

C. For each use case, we can write a description of the flow of events that constitute that case, incorporating the user's stated requirements.

1. Example: ATM session

   **HANDOUT - Session Use Case Flow of Events**

2. As mentioned earlier, a session includes some number of individual transactions

   **HANDOUT- Transaction Use Case Flow of Events**

3. The above illustrates an important point about use cases: they typically follow a normal pattern, but have to allow for variations in individual cases.

   a) The customer may enter an incorrect PIN.

   b) The bank may disallow the transaction for other reasons (e.g. insufficient funds for a withdrawal, or the customer does not have an account of the specific type chosen).

   c) The customer may cancel the transaction in the early stages by pushing the "Cancel" button.

   etc.
These variations can be accounted for either directly in the use case (if this happens, then ...) or by creating additional sub cases. (In particular, the case of an improper PIN might well be implemented as a sub case, since a similar pattern will occur for all types of transactions.)

NOTE INVALID PIN USE CASE EXTENSION ON HANDOUT AND REFERENCE TO IT IN TRANSACTION

Often, it is useful to develop several scenarios for a given use case - a typical scenario and various special case scenarios.

4. Sometimes, a general use case has various specialization's.

EXAMPLE: “Transaction” generalizes four different specific kinds of transaction: withdrawal, deposit, transfer, and inquiry.

HANDOUT - go over withdrawal

D. Use case flows of events can be developed with greater or less formality.

1. The examples in chapter 7 of the book lean more in the direction of formality.

   a) TRANSPARENCY - Example 7.2

      Note explicit statement of Actor, Goals, Preconditions; use of “two column” format

   b) TRANSPARENCY - Example 7.4

      Note explicit mention of related use cases

2. It is possible to use an even more formal style - the so-called “usecases.org” format

   a) TRANSPARENCY - Larman pp. 50-53 (excerpts)

      Note explicit statement of Stakeholders, Success guarantee (postcondition), etc.

   b) This can also be done in a two column format

      TRANSPARENCY - Larman p. 53 two column alternative
3. Sometimes, before creating a fully-dressed use case, it is helpful to develop one or more scenarios for typical cases. Note that these are quite informal, and focus on what normally happens, not on exceptions.

TRANSPARENCY - Example 7.5

E. Class Exercise:

a) Develop deposit flow of events

b) Show flows of events for deposit, transfer, inquiry on line, as well as operator cases.

III. Analysis Class Diagrams

A. A tool that is often useful at this point is called an analysis class diagram. In an analysis class diagram, we model the domain of the system, and key classes that arise in the context of the use cases.

1. These classes fall into three broad categories.

a) Boundary classes whose objects serve as interfaces between the system and the rest of the world.

EXAMPLE: ATM System - ASK

Card reader, customer console etc.
Also: network connection to bank

b) Entity classes whose objects represent information that is stored and manipulated by the system. These mostly come from the domain.

EXAMPLE: ATM System - ASK

Representation for customer’s ATM card
Log of transactions

(Note: in this case, a bank account would certainly seem to be such an entity, too; however - as it turns out - the task of dealing with bank accounts really ends up belonging to the Bank system we interface with, not with the ATM itself. In the context of the domain of banking, an account is a very important kind of object - it just doesn’t fall directly into the scope of this system)
c) Controller classes whose objects control and coordinate the activities of other objects. In particular, we will often associate an object with each use case that exists while that case is active, and whose internal state keeps track of information we need as we progress through the case. (But not every use case needs its own object - if a use case is simple enough, it can be made the responsibility of some other object).

2. Another thing we can discover by reading the use cases is something of how these objects relate statically to one another - i.e. which objects have to interact directly with which other objects.

Example: We find, in reading the Session use case, that the first thing that happens during a session is that the customer’s ATM card is read. This implies a relationship between the Session object, the Card Reader object, and the Card object.

Example: We find, in reading the Session use case, that the next thing that happens during a session is that the customer is asked to enter his/her PIN. This implies the need for a Session object to have to interact with the Customer Console object. (The same is true of the various Transaction objects.)

3. We can take an initial cut at representing the key objects of the system and their relationships to one another by creating an analysis class diagram at this point.

a) We call this an Analysis class diagram because it focuses on the classes that become evident during analysis - i.e. the ones that are evident as we focus on what the system must do.

b) When we get to the design phase, we will discover more classes that are needed when we actually think about how we are going to build the system.

(Example: when thinking about the design of a house, one thinks about things like floors, doors, windows, walls, plumbing fixtures etc. But when the house is actually built, the builders have to also deal with joists, studs, rafters, sub flooring, etc. - things that are not evident to an occupant of a house, but which are certainly necessary to build it.)

c) Recall the comment earlier that Booch, et al., suggest that there is likely to be something like a 5:1 ratio between the number of
classes ultimately needed for implementing the system and the
number of analysis classes discovered initially.

B. In creating an analysis class diagram, we may make use of specialized
symbols for the three types of analysis classes

1. Boundary classes

2. Entity classes

3. Controller classes

C. HANDOUT: Analysis classes for ATM System
IV. An Analysis Example To Work as a Class

A. Suppose we were developing a new registration system for the Gordon College registrar, which allows students to interact with it over the web.

As a starting point for considering this specific application, let’s see if we can formulate a statement of requirements for such a system.

ASK

B. Use Case Analysis

1. What actors would use this system? (Note: actor is based on role, so we can have several kinds of student actors depending on the reason for the student's interaction with the system.)

ASK

2. What use cases can we identify for these actors?

ASK

3. Develop use case diagram

4. Develop use case flow of events for a student signing up for courses for the next semester.

C. Develop an analysis class diagram for this system.

V. Summary

A. We have looked at two general approaches to doing analysis:

ASK

1. Domain analysis

2. Application analysis

B. We have also looked at two tools we can use when doing analysis:

ASK

1. Use cases (diagram and flows of events)

2. Analysis class diagrams