Systems Analysis and Design

Tak Auyeung, Ph.D.

May 16, 2005

- 20050309: add table example to 11.2.3
- 20050307: add pictures to 10
- 20031026: add section 13.2
- 20031001: add chapter 11
- 20030916: add chapter 10
- 20030913: add chapter 9
- 20030910: add homework assignment 8
- 20030827: add chapter 7, up to but not including 7.2
- 20030926: add chapter 6
- 20030828: add section 5.3
- 20030827: add material from 5.2.1
- 20030824: chapter 5 in progress
- 20030824: add chapter 4
- 20030820: added more to chapter 2
- 20030819: chapter 2 is created
- 20030817: first chapter is being written, not finished yet.
- 20030816: creation

Contents

0.1 Copyright Notice

Part I

Background

Chapter 1

Information Systems

1.1 Computers: the Unneccessary Component

Not all information systems use computers.

It is important to remember that an "information system" is simply a system that deals with the collection, storage, organization, retrieval and analysis of information. This means information systems have been in place long before computers were invented.

It is true that most information systems that people mention today use computers. But this does not mean that we should make that assumption immediately. There are still plenty of businesses and organizations that do not utilize computers to handle information.

1.2 What is an information system?

As mentioned in the previous section, an information system is one that handles information. That is a very vague description. Perhaps some examples help to illustrate.

An example of a large information system is the system that is used to track drivers' records and vehicle information. The DMV of California needs to keep track of a tremendous amount of information. In this case, using computers as a tool makes it possible.

Another example of a small information system is the system that a family-owned small grocery store uses to track inventory, orders and sales. This type of information system is small scaled and the use of computers is certainly not necessary.

If you think about it, you, too, use information systems in your daily life. Some religiously track income and expenses, others track phone numbers, addresses and birthdates systematically. Teachers, including professors, track the performance and grades of students throughout a semester. At the end of a day, some of us sit in front of a computer to enter and maintain information in

s Quicken-type application, while others balance the book literally using paper-based ledgers.

As you can see, information systems come in all sizes and complexity.

1.3 Who is a computer systems analyst?

What is in an information system to analyze?

As the name implies, a computer systems analyst is a person who analyzes a computer-based information system. But this hardly describes what a systems analyst do. Afterall, what is there to analyze? What can go wrong when a system is not analyzed properly?

An excellent resource is http://www.umsl.edu/ sauter/analysis/analysis_links.html, you should read through some of the stories (fables). Not only are they easy to read, but they are also so true in the realm of systems analysis.

A systems analyst is not a programmer. A programmer writes programs to meet pre-determined specifications. A systems analyst is responsible for the specifications.

A systems analyst is not a tester. A tester executes tests to determine whether a computer system does what it claims to do. A systems analyst specifies what a computer system is supposed to do.

A systems analyst is not a system administrator. A system administrator maintains the operation of a computer system. A systems analyst specifies the components that make up the computer system.

I am sure you are painting a picture of what a systems analyst does not do. Does a systems analyst do *anything*?

The simplest functional description of what a systems analyst do is "to design, modify or enhance an information system that meet the requirements and objectives of a client and maintains its specifications." Let's pick this sentence apart and explain the components.

"Client" in this context simply means an organization (possibly of one person) that is in need of a new or better information system. If you are a systems analyst working for the Board of Equalization, your client is most likely internal, the Board of Equalization itself. If you are a consultant or work for a consulting group, your client is most likely external, a company that pays for your services.

"Requirements" are a set of quantified or qualified statements that the information must meet. For example, a growth related requirement may state that "the system must be scaleable to handle a data size growth of 30% annually." An efficiency related requirement may state that "the system must reduce the current transaction processing time by 40%." A requirement can also defer much details. For example, a security requirement may simply state "the system must be rated level 3 or above by computer security consulting firm XYZ." In short, "requirements" is "what the system does."

"Specifications" are a collection of documents (text, charts, diagrams and etc.) that describes *how* the proposed system operates. A common question is "how detailed?". There is no clear answer to this question. It suffices, however, to say that the specifications must be sufficiently detailed for other parties in the

development, deployment and maintenance of the computer systems. In other words, the "specifications" are documents that programmers, testers, system administrators, purchasers and etc. use to do their jobs.

Some believe a systems analyst is also responsible for the management and execution of the development of an information system. In other words, a systems analyst makes decisions and tells others what to do. This is not the case in general. A systems analyst is responsible for projecting a proposed timeline/plan, but this is a specification that results from the analysis of task dependency, available resources and task resource requirements (more on this later). A systems analyst usually does *not* determine task dependency, available resources or task resource requirements.

You can also see a systems analyst as the glue that holds all other participants together so everyone is "on the same page". Please note that the systems analyst does not necessarily *determine* the content of that "same page". A systems analyst is, however, responsible to communicate with other participants so that whatever is determined is written precisely, completely and distributed efficiently.

1.4 Homework Assignment (100 points, due one week after it is assigned)

Collect three systems analyst job advertisements or job descriptions and comment on what you think they mean. Remember to quote the source of the advertisement or description. Try to find ones that differ as much as possible from each other. What is different, besides the technical requirements, among the advertisements or descriptions? What is still shared by these advertisements or descriptions? What is *your* description of the minimal duties of a systems analyst?

The break down of points is as follows:

- each advertisement is worth 10 points for a total of 30 points
- legibility (including quality of handwriting, formatting and etc.) is worth 20 points. 0 point for stuff that I cannot read, 10 points for stuff that I can read but with difficulty, 20 points for stuff that I can read easily
- the differences are worth 15 points.
- the common points are worth 15 points.
- your own description is worth 20 points.

Because this course has no prerequisite, I will not require that you turn this assignment in by electronic means. If you turn it in in-person, make sure that I mark the date of submission.

Chapter 2

The Participants

This chapter introduces the various participants who are involved in the development of an information system.

2.1 The Client

An mentioned in the previous chapter, a "client" is simple a person or an organization that needs to have an information reviewed, revised, improved or created.

2.1.1 The Client's Expertise

Although most clients may not be computer savvy, they are all familiar with their respective "business". In this context, "business" simply means "whatever activities that need to be handled by an information system." The knowledge to conduct business is sometimes called "domain specific knowledge", and the client acts as a "domain specific expert".

A systems analyst cannot assume a client's ability to put everything about conducting his/her/their business in writing. Often enough, especially with larger organizations, there are operation manuals and other document to help a client explain how business is conducted. Other times, what is written may not be what is practiced. It is the responsibility of a systems analyst to inquire a client to clarify points.

2.1.2 The Client's Motives

A client does not initiate the development of an information system simply because there is money to do so. The need for a new/revised information system can be a combination of the following:

- the current system (manual or otherwise) is inefficient

- there is no system in place for a new business
- the current system cannot handle anticipated growth
- the current system is at a dead end (no future maintenance and support)
- the current system does not handle a required (new) feature

A client's motive(s) to initiate the development or revision of an information system is a starting point for more questions. For example, if a client states that "our competitor processes significantly more transactions then us", a systems analyst should ask "could you quantify the competitor's ability?"

2.1.3 The Client's Requirements

Believe it or not, although all clients have their reasons to initiate a project, few actually knows the requirements of the new information system. This is analogous to a hungry child knows he/she is hungry and wants to eat, but he/she does not necessarily know how much of what type of food is best for his/her health.

As a result, a systems analyst must discuss with a client and specify the exact requirements. For instance, a client may initially say "our information system must be faster at transaction processing than our competitors." The natural response is "how much faster?" If a client says "the new information system must keep up with our growth", then an analyst should ask "what is the projected growth rate?"

2.2 The Developer/Programmer

Although there are CASE (Computer-Aided Software Engineering) tools that generate code, most systems are still hand coded by developers (programmers). A developer/programmer writes code that eventually will be compiled or interpreted for execution.

2.2.1 The Programmer's Expertise

The lead of a programming team should be familiar with programming languages, development tools and other technical topics that relate to programming. It is probable that the lead of a programming team knows more about programming languages and other topics than systems analysts.

2.2.2 The Programmer's Input

In most organizations, programmers are told how finished programs should behave. One can argue that if a programmer can be told how finished programs should behave, whoever specifies the behavior can write the programs in the first place.

This is true only if the behavior of the finished product is specified in low-level detail. Generally speaking, the specification of program behavior given to developers is no more detailed than the specification reviewed and approved by the client. Yet, this specifications must leave no room for ambiguity.

For example, let us consider "let a user attempt to log in three times, then lock the user out of the system after three failed attempts." This specification is ambiguous. The client may understand it as "three *consecutive* failed attempts", while a programmer may understand it as "a *total* of three failed attempts, not necessarily consecutive." It is the responsibility of a systems analyst to ensure that a specification only has one interpretation (no ambiguity).

Another type of documents distributed to developers is bug report. However, a systems analyst should ensure every bug report distributed to developers relates to the original specifications of the programs. In other words, instead of simply stating "the log in screen times out too quickly", the bug report should indicate "the log in screen times out in less than 10 seconds, as specified in section 12.3.5 of the specification, version 2.6.1." What if there is no specification of the time out delay?

2.2.3 The Programmer's Output

Obviously, a programmer write code, and that's the output. In the process of systems development, the output of a development team is not only code, but "code that meet specification document version 2.6.1." The version number is important because it clearly identifies *which* specification. It is not uncommon that a development team may be working toward version 2.6.1, while the client proposes changes and creates version 2.6.2.

In response to a bug report, the developer's output is the modified (fixed) code with a reference to the bug report. The reference to the original bug report is important because it associates a code change (and the version) to a specific bug report.

2.3 The Quality Assurance (QA) Team

The necessity of a QA team composed of members *other than* the developers is not always understood. Why can't the developers be their own QA? Why do we need a QA team in the first place, can't developers write code that works the first time?

The answer is it depends. For tiny systems, a developer can effectively test his/her code because there is not much else to interact with. For large systems, however, a developer can only practically test that his/her module works independently according to the distributed specifications. After individually tested modules are integrated into a complex system, their inter-module side effects can easily lead to "bugs".

As a result, it is often necessary to have a QA team that assures the *integrated* system behaves correctly (consistent with the specifications).

2.3.1 The QA's Input

The QA team should be given specifications of how the system is supposed to behave. This is sometimes the exact same document given to the development team.

2.3.2 The QA's Output

The QA team has at least two output documents. The first one is a test plan for a particular specification. A test plan describes the tests used to verify a system behaves as specified. The second one is a validation report which includes the *outcome* of the tests described in the test plan. Parts of the validation report serve as bug reports to the development team.

2.4 Project Manager

In a small project or a small organization, a project manager often wears other hats. However, for large projects, a project manager only has one responsibility: to manage a project. This includes the coordination of time, budget and other resources allocated to the project, as well as to track the progress and alarm various parties if the project is not proceeding as planned.

The overall project plan is the result of colaboration of the lead people of various departments. Neither a project manager nor a systems analyst should dictate "how much time it will take to do XYZ". Once a project plan is agreed on, a project manager coordinates with other participants to ensure the plan is followed as closely as possible.

We will discuss project management later in the semester. The specific diagrams, techniques and tools will be discussed then. For now, we are only interested in an overview of what a project manager does and how that relates to a systems analyst.

Because only systems analysts communicate directly with the client, systems analysts communicate with project managers in an early stage to set the basic parameters of a project. These parameters include target dates of deliverables (intermediate and final), overall schedule (as far as the client is concerned) and allocated budget. The detail of a project plan is worked out when the specifications of the proposed system is made available. Without the specifications, other participants (programmers, QA and etc.) cannot accurately estimate resource requirements.

After a project is initiated, system analysts do not typically track the daily progress of subtasks. With the help of software tools like MS Project, a project manager keeps track of progress of the tasks in the project. When a task consumes more than its allocated resources (or when that appears inevitable), a project manager alarms others to find a solution.

2.5 The Systems Administrator

A systems administrator is responsible for the installation, daily operation and maintenance of a computer information system. Some projects need to involve a systems administrator early because of the necessity to install new computer hardware and software components.

In the analysis process, systems administrators are sometimes involved because they understand the cost of keeping a system operational and are experts in systems level components. For example, given a limited budget but also a lower performance requirement, systems administrators can participate early in the analysis phase and suggest suitable components. As such, systems administrators participate after a preliminary systems specification is available but before the presentation of a proposal to the client.

In general, systems administrators do not participate in the analysis and development of an information system as actively as programmers, QA personnel and systems analysts. Nonetheless, they possess specialized knowledge and expertise that is necessary to design and implement a successful information system.

2.6 The Technical Writer/Trainer

It is seldom that a systems analyst doubles as a technical writer or a trainer to lead training sessions. A technical writer is a person who is specialized at writing technical documents such as user's guide and operation manuals. Although a systems analyst *must* have good verbal and written communication skills, a systems analyst focuses on more technical and concise documents.

A technical trainer leads training sessions for the client's operators and end users. The trainer is responsible for the authoring of the materials used in such sessions.

Technical writers and trainers must understand the systems specifications prepared by the systems analyst. Although they may not need to understand how things are done *internal* to the system, they need to understand the specifications that relate to the operators and end users.

Technical writers and trainers are involved usually after a systems specification is approved by the client. Technical documents and training materials can, sometimes, be developed in parallel with coding and test plans (because all are based on the same specifications).

Part II

Systems Requirements

Chapter 3

Initial Contact

This chapter describes what needs to be done in the initial contact with a client.

3.1 Objectives

One of the most important objectives of the first meeting is to understand enough about the client and the needs to start an information systems project. It is helpful for an analyst to review what outcome is expected from the initial meeting before attending one.

3.1.1 Motives

Understand *why* the client is initiating the project. Some motives were discussed in 2.1.2. A good understanding of the motives of a client provide guidance to technical steps in the analysis and design of an information system.

3.1.2 (Vague) Requirements

It is unlikely that an analyst can get sufficiently specific requirements in the first meeting to start the analysis phase. However, it helps to at least acquire "ball-park numbers". The knowledge of ball-park numbers helps to eliminate useless questions in subsequent meetings.

3.1.3 Additional Contacts

Client participants in the initial meeting may not attend future meetings. Executive managers go to the first meeting to express their executive objectives and requirements, but they may not attend later meetings that focus on details. As a result, it is important to know whom else in the client organization should be involved in following meetings.

3.2 Preparation

Always prepare before a meeting. There is much to prepare, especially before the initial meeting with a client.

3.2.1 Attendee Information

Be sure to know who is attending the meeting. An analyst cannot control whom from the client organization is going to attend a meeting. However, an analyst should always know ahead of time the names of attendees and their responsibilities (as far as the information system and management is concerned).

It is helpful to prepare name plates prior to a meeting. This allows other attendees to quickly get acquainted with each other.

3.2.2 Meeting Agenda

Sometimes, a client decides the meeting agenda of an initial meeting. This makes sense because, afterall, the client is *initiating* the project. Other times, however, a client leaves it up to the systems analyst(s) to decide the meeting agenda.

Be sure to confirm who is deciding the agenda. The worst case is both the client and analyst think they are not responsible for the agenda. A meeting without an agenda is a disaster waiting to happen. This is particularly the case when the number of attendees is large.

If the systems analyst is responsible for the agenda, the agenda should be distributed to the client as early as possible. This way, not only can the client prepare for the meeting based on the topics on the agenda, but the client can also suggest necessary changes.

3.3 The Meeting

A systems analyst needs to let the client express the needs first, then responds by asking clarification questions and set up for subsequent meetings.

This chapter discusses the various issues an analyst may encounter in the initial meeting with a client.

3.3.1 Listen

An analyst must listen carefully, especially in the initial meeting. Sometimes a client expresses the motivations and needs precisely and with much rationale explained, other times a client may only have bits and pieces of why a new system or improvement is needed.

One of the worst thing that can occur is an analyst immediately drilling into details and starts to make suggestions to the client. Not only does this behavior distract the client from his/her train of thought, it is also too premature to make any suggestions.

Recording each and every meeting is often a good idea with the consent of the client. Such audio records can be helpful later on when an information systems development team need to review details of a meeting. It makes sense to make the recording available to the client so the client can also review meetings afterwards.

3.3.2 Document

If meetings are recorded, the documentation of meetings can be done later. However, if there is no audio records, it is best to document important points of a meeting as the meeting progresses.

Although an analyst can jot down notes and points in a meeting, such an activity can be distracting and interferes with the listening and analysis responsibilities. It is best, whenever possible, to rely on another person to trascribe a meeting.

3.3.3 Ask

At appropriate moments, ask a client to clarify or explain a point. However, the questions should not interfere with the client's flow. If a question is likely to have a long answer, write it down and ask later. Questions that have definite short answers are usually not a problem.

3.3.4 Summarize and Confirm

Instead of just nodding and confirming with "yes", "a-ha", or "understood", it helps to confirm with reexplaining or rephrasing to the client. Such responses not only let the client double-check that the analyst's understanding is correct, but it also improves the interactive quality of a meeting. Just by rephrasing or reexplaining a point, an analyst often gains better understanding of a point made by the client.

3.3.5 Defer

Defer any question that has a long answer to either the end of the meeting or another meeting. Make sure any question asked during the meeting does not interfere with the flow.

3.4 What you can expect

From the first meeting, an analyst can expect to know the business of a client organization. For example, in the case of an inventory auditing client organization, an analyst should get a basic understanding of the business of inventory auditing. The keyword here is "basic". Who are the clients of an inventory auditing company? What services are offered?

An analyst can also expect to understand why a client wants to initiate an information systems project. Is the current system insufficient? Are the competitors all doing better? Is this a preventive project to ensure future operation?

Who else in the client organization will attend following meetings? Whom should an analyst contact when there are technical questions? Who makes the decisions?

3.5 Post-processing

After the initial meeting, there is much to process. It is difficult to generalize exactly what needs to be done because the outcome of the initial meeting depends on the client.

Based on the review of the transcript or audio recording of a meeting, analysts prepare for a follow-up meeting. It is unlikely that the first meeting will provide all the information an analyst needs to get a project started.

Identify action items and who should perform those actions. This is the number one item in the post-processing of any meeting. Who is going to provide the manuals of operation? Who is going to review the transcript and send a copy to the client?

Chapter 4

Non-Functional Requirements

Non-functional requirements refer to requirements that are not related to the functionality of an information system. In reality, any requirements that cannot be captured by a systems analysis tool suite is "non-functional".

This chapter discusses what type of non-functional requirements an analyst needs to capture, document and maintain.

4.1 Financial Requirements

Most systems analysis tools/languages do not capture the financial requirements of an information system. Such requirements include the overall budget, worst-case cash-flow schedule, amortized operation cost and etc.

A good understanding of the financial requirements of a client organization is *critical* to the success of an information systems project. Even if an IT team can design the most cost effective and highest quality information system, the project can still fail because a client bankrupts in the middle of the project.

Most clients, especially external ones, are only willing to provide an initial budget for an information systems project. Such a budget is usually a schedule of how much funding is available at what time. In this case, an analyst should remind the client to make sure the client has analyzed the information systems budget with respect to the worst case cash flow schedule.

Internal clients, however, are more willing to share the worst case cash flow schedule with systems analysts. Although there are other factors that determine the worst case cash flow schedule, at least the systems analysts understand how the information system fits in the overall financial plan of the client organization. Such an understanding is used to create budget schedules and check points. These feedback mechanisms provide early warning when the development of an information system becomes too costly. Early warnings, in return,

allows the client organization and the development team to react sooner to avoid irreversible failures.

4.2 Security Requirements

While some security issues that relate to the operation of an information can be captured by diagrams, most systems analysis languages/tools do not capture overall security requirements. This is, however, not to say overall security requirements are not important in the analysis and design of an information system.

For example, an internet-exposed information system by means that are independent of the business dependent operations. A DOS (denial of service) attack can be launched against an information system regardless of the business that the system serves.

As information systems evolved from specialized and closed hard wired system to generalized and internet connected distributed systems, security has become one of the most important factors to keep a system operational. Even savvy systems analysts often do not know all the aspects and issues with computer security. As a result, it is best to contract to computer security specialists to analyze the security requirements of a computer information system.

4.3 Reliability Requirements

The initial requirement is always deceptively simple: "we want the system operational 24/7/365 despite acts of God." In reality, reliability comes in all shades of gray.

On the hand, using a single hard disk with no redundancy makes a system vulnerable to a single hard drive or controller crash. On the other hand, an analyst can specify storage subsystems with astronomical cost to make the subsystem "the last component to fail". How can a client choose a requirement from this continuous gradient of gray?

Remember, the storage subsystem is only one component in an information system that affects the overall reliability.

The ultimate question that a client needs to answer is "How much does it cost per incident of loss of service? What is the cost of each down hour?" A client does not care about the latest and greatest hard drive redundant array, nor the availability of completely redundant systems. The bottom line is how much it costs per incident and how infrequent the incidents occur.

As a result, instead of asking a client "how much reliability is required", it makes more sense to ask "what is the cost of losing operation, how much does it cost per down-time hour?" Of course, this number changes as the business grows. This brings us to the next section.

4.4 Growth Requirement

Most information systems have lifespans of multiple years. Over the expected lifespan of an information system, the business of a client may grow. The growth rate is a key factor in the analysis and design of an information system that guarantees to remain operational during the expected lifespan of the information system.

With larger and older organizations, growth can usually be predicted based on historical trends. In other words, a client can use the average annual growth over the past 10 years as an estimate of the future growth rate for the next 5 years. A safety margin may be useful based on the amount of growth fluctuation in the past.

Smaller and young organizations, however, have a much more difficult time predicting growth. Such a client will probably say "only time will tell", which is useless from the perspective of systems analysis. Instead of asking "what is the expected average annual growth rate over the lifespan of the system", it makes more sense to ask "what is the business size before the system is deemed obsolete?" This way, the systems analyst only needs to make sure the system has a certain capacity, it is up to the client to know when to look into a new system.

4.5 Delivery Date

This is a touchy subject, especially in the area of systems design and implementation. Nonetheless, it is an important requirement that should be stated as early as possible. This requirement is important because it affects the financial and strategic planning of a client organization.

4.6 Other Non-functional Requirements

There are other non-functional requirements. Such requirements are just as important as other requirements because they collectively state the "bottomline" from the client's perspectives. Some of these requirements may be vague and fuzzy initially, but that is fine for the time being.

4.7 Tracking Non-functional Requirements

Because most CASE (computer-aided software engineering) tools only track functional requirements, an analyst often needs to use more "traditional" means to document and track non-functional requirements.

Just like all other documents, it is best to choose a format that supports "version control". In other words, a feature that allows the convenient and visual comparison of different versions of a document. Microsoft Word (and

most compatible word processors) supports flexible and visual version control for documents.

While it is convenient to computers to create, track, distribute and view documents, it is equally important to "back up" documents with printouts. All major versions, especially ones released to the client, should be filed and organized by a document control person.

Chapter 5

Functional Requirements

The functional requirements relate to the functionality of an information system. As such, most mature CASE tools provide means to capture such requirements. This chapter introduces the basic functional requirements capturing diagram used in the UML (unified modeling language).

5.1 Actors

In the UML, an actor is a *classification* of agents that are *external* to an information system that interact with it.

For example, in an information system for curriculum process, the classification of "professor" is an actor. The classification of "committee member" is another actor. Note that Tak is an *instance of* the "professor" actor, but Tak is *not* an actor.

5.1.1 Getting started with actors

Actors are a natural starting point for capturing the functional requirements of an information system. A client can typically list most, if not all, of the various types of participants who will interact with an information system.

Initially, try to capture actors without worrying about how actors interact with each other and how each actor interacts with the system. Just ensure that each actor is external, a classification and somehow interacts with the system.

5.1.2 Organizing actors

The UML allows fairly complex relationships among actors. Of all these relationships, the most useful type is "inheritance" (also known as generalization or specialization). This type of relationships help to make the document concise.

Let us consider an information system that handles enrollment. In this system, any student can add a class, drop a class and check his/her current

enrollment. For CIS students, however, the system also allows the initialization of a roaming computer account that is available at the computer labs.

In this example, the actor "CIS student" is said to be a specialization of the actor "student". One can also say that the actor "student" generalizes the actor "CIS student". Someone familiar with object-oriented concepts can also say that the actor "CIS student" inherits from the actor "student".

All of the above alternatives has one meaning: an instance of "CIS student" can do everything that an instance of "student" can.

This relationship is very convenient because an analyst does not need to unnecessarily replicate functional requirements.

Note that generalization can be "many-to-many". In other words, in our example, the actor "art student" is also a specialization of the actor "student". At the same time, the actor "CIS student" is also a specialization of the actor "CIS lab user".

5.2 Use cases

While actors provide a starting point for capturing the functional requirements of an information system, use cases deals with the bulk of the details of the functional requirements of an information system.

A use case is a *complete* interaction of *at least one* actor and the information system that provides *value* to at least one participating actor. Let us try to explain this rather long and vague description.

"Complete" means the interaction cannot only serve as a component of another interaction. This rules "log in" out as a possible use case because logging in to a system can only serve as a step of some other uses of an information system. By comparison, "check grade" is a complete interaction because an actor can check grade without any further interaction.

5.2.1 Inclusion

A use case can include another use case. If use case A includes use case B, that means use case B is a component of use case A. Please note that it is okay for a use case to be a component in another use case, as long as the included use case is complete by itself and it provides value to at least one actor.

For example, in an airline information system, "check flight information" is a valid use case. An end user can simply use the system to check the arrival time of a flight. At the same time, this "check flight information" use case can be a component of the use case "book flight ticket".

5.2.2 Extension

A use case can extend another use case. "Extension" does not imply "inclusion". In other words, when we say use case A extends use case B, A does not include B.

The "extension" relationship in use cases is rather complex and it requires a good understanding of exception handling. This class will not discuss use case extensions.

5.2.3 Inheritance

A use case can inherit from another use case. However, use case A inheritting from use case B does not mean that use case A includes use case B. This is because use case B is not a component of use case A.

When use case A includes use case B, use case A can only specify exactly the same behavior as use case B. Use case A can, however, add behavior before o after use case B. In our earlier example, "book flight ticket" includes "check flight information". This means "check flight information" may include user log in before "check flight information" and include "order ticket" after "check flight information". Nonetheless, whatever behavior is specified in "check flight information" cannot be changed by "book flight ticket".

When use case A inherits from use case B, however, use case A can modify certain attributes of use case B, even to add new attributes and behavior. For example, "check flight information" may inherit from a use case called "query flight data". The "query flight data" use case is intended for experienced users who wish to use a more command-line oriented interface for efficiency. Airport code is manually entered (SFO, LAX, etc.) rather than selected from a drop-down box. The "check flight information" use case, on the other hand, is intended for end users who do not memorize airport codes. As a result, use case "check flight information" inherits everything from "query flight data", but it redefines some attributes and methods so it is more end user friendly.

An analogy may help to illustrate the differences between inheritance and inclusion. A fast-food meal *includes* a drink, an order of frys and a burger sandwitch. However, a cheeze-burger inherits from a burger sandwitch. In this case, a fast-food meal cannot change the attributes of the included burger sandwitch. However, a cheeze-burger can add a new attribute "cheeze" to a burger sandwitch.

5.3 Use case to/from actor association

Association relationship are the only type of relationships that is allowed between actors and use cases.

Each use case must be associated with one or more actors. Likewise, each actor must be associated with at least one use case. There is no limitation of how many use cases can be associated with an actor, or how many actors can be associated with a use case.

Note that at this stage, it is not necessary to know *how* an actor is involved in a use case. Instead, a systems analyst should concentrate on enumerating all actors who can be involved in a use case.

5.4 Use case diagrams

Actors, use cases and their associations are captured by use case diagrams in the UML.

5.4.1 What is in a use case diagram?

A use case diagram pictorially illustrate how actors and use cases are related. Use cases are represented by elipses, while actors are represented by match-stick figures.

A "specialize" relationship is represented by a solid arrow going from the specific to the general. In other words, the arrow goes from the actor "CIS Student" to "Student".

An "association" relationship is represented by a solid line (no arrow head) between a use case and an actor.

An "include" relationship is represented by an arrow of dotted line from the including to the included. There is usually a letter "i" on the arrow to indicate this is an inclusion relationship.

An "extend" relationship is represented by an arrow of dotted line from the extending to the extended. There is usually a letter "e" on the arrow to indicate this is an extension relationship.

5.5 Free UML Tool

While you can use "traditional" drawing tools to draw use case diagrams, it is often better to use a tool that is designed to capture UML diagrams. Most such tools are expensive, but you can download and use a "community edition" of Poseidon for this purpose.

5.5.1 Getting Java Installed

To make Poseidon work, you need to download the Java virtual machine. Go to http://java.sun.com, and click on "Downloads". Then look under "Java 2 Platform, Standard Edition (J2SE)", click on the drop down box and select "All Platforms" under "J2SE 1.4.x". Click the "Go" button.

Look up the heading "Download J2SE v 1.4.x_xx". When presented the various download options, choose you operating system. For most, choose "Windows Installation" or "Windows Offline Installation" and click the "DOWNLOAD" link under the "JRE" (Java Runtime Environment) option. Choose "Windows Installation" if you have broadband connection and want to install the software over the web. If you want to keep the installation file (so you can install it on another machine), click the "Windows Offline Installation" link.

Go through the J2SE JRE installation. Be sure to let the installer install "Java Web Start". It leaves an icon on your desktop.

5.5.2 Downloading and Running Poseidon

Once you have Java installed, go to http://www.gentleware.com. Click on the picture labeled "Launch! Now! with Java WebStart", this should automatically start the Web Start installation. The first time this is done, it may take quite a while to download all the files. However, future invocations should be faster (unless a newer version is available for download).

Once you have run Poseidon once, Java Web Start has an entry for it. You can invoke Java Web Start, then select Poseidon to start it.

5.6 For the Analyst/Programmer

A use case is modeled by two types of classes (in the object oriented sense). The first class is a "control" class that captures the logic, methods and operations of a use case. There is exactly one control class for each use class. The second, call a "boundary" class, models how a use case is presented to an actor.

If a use case has multiple actors, actors may share a common boundary class or each has its own boundary class. Note that boundary classes apply to actors representing human users as well as external information systems.

For now, there is no need to decide how boundary classes are assigned to actors. We can wait until more is determined.

5.7 Summary

Use a use case diagram as the first step to functional requirement capture. A use case diagram consists of actors and use cases. An actor is a classification of agents external to the information system that interact with the system. A use case is a function of the information that must be participated by at least one actor. A use case must be complete and provide value to at least one actor.

The most important point to remember is that we are trying to capture the functional *requirements*, but not how things are done in details. Focus on "what" and not "how" for now.

Chapter 6

Capturing Use Case Scenario

In the previous chapter, we discussed actors and use cases. Although actors and use cases are good starting points to capture the functional requirements of an information system, they are insufficient as far as an estimate (of necessary time, money and other resources) is concerned.

This chapter discusses "sequence diagrams" and "scenarios". Some additional concepts are also introduced in order to complete the discussions of sequence diagrams and scenarios.

6.1 A Use Case Instance

A sequence diagram serves as the log of a use case instance. The emphasis here is "instance". In other words, we are *no longer* working with classifications. Rather, in each sequence diagram, we use actual "objects", or instances of classifications.

Sequence diagrams are very general and can be used to capture the log of any sequence of message interchange. However, in this context, we will only use sequence diagrams to capture how an instance of an actor interacts with the information system being modeled.

It is important to remember that a sequence diagram is like a video tape. It captures the happening (or imagined happening) among two or more actual participants (not classifications of participants).

6.2 Sequence Diagrams

In a sequence diagram, columns represent timelines. A sequence diagram is read from top to bottom (top being the earliest time). The heading of each

column indicates a participant. Messages are allowed to flow between columns to indicate how information is passed from a participant to another.

Refer to figure 6.1 for an example. This diagram is created by Poseidon for UML. In this diagram, "student" is a particular student, and "grading system" represents an implementation of the information system.

Note the solid arrows aligned horizontally. These arrows indicate a "call" message. A call message is initiated by the object opposite to the direction of the arrow. The dotted arrows indicate "return/reply" messages that only serve as responses to call messages.

Each "link" (or arrow) is captioned. The caption has the format of "*stimulus*:*dispatch action*". The first part, the stimulus, describes the origin, rationale or objective of the message (determined by the originator). The second part, the dispatch action, indicates an action taken by the receiver of the message as the message is received. Sometimes, especially in reply messages, the "dispatch action" can also indicate a decision made or an answer (result, returned value) to the originating call message.

Let us take a closer look at the first call message labeled "request check grade service : check session status". The first part of the caption indicates that the reason of the student to originate this message is to "request check grade service". The message, upon reception, triggers the action called "check session status" in the grading system.

Now, let us examine the second message, which is a reply message to the first one. This one is captioned "request authentication : web login form". This means the reason for this message is to "request authentication" (because there is no active session). The payload of the message is "web login form". Note that "web login form" is not a dispatch action. Instead, it simply describes what is contained in this message.

Note that we do not specify logic in sequence diagrams. In other words, we do not explain how the system knows that there is no active session, nor do we explain how the system authenticates the end user. This sequence diagram merely logs the messages that are passed between the student and the system to check the grade of a class.

6.3 How Much Details? What is the Scope

At this stage, we are only interested in the interaction between instances of actors and the information system. We do not, for example, use sequence diagrams to model the interaction among objects *inside* the information system. Furthermore, we do not even care about the feel-and-look of user interface elements.

At the same time, there is no need to enumerate all possible scenarios for use cases. It is usually impossible to do so, anyway. Instead, focus on fewer cases that are "representative" of possible scenarios.

It is tempting, especially for those with a strong background in programming, to think about the logic when completing a sequence diagram. While there is

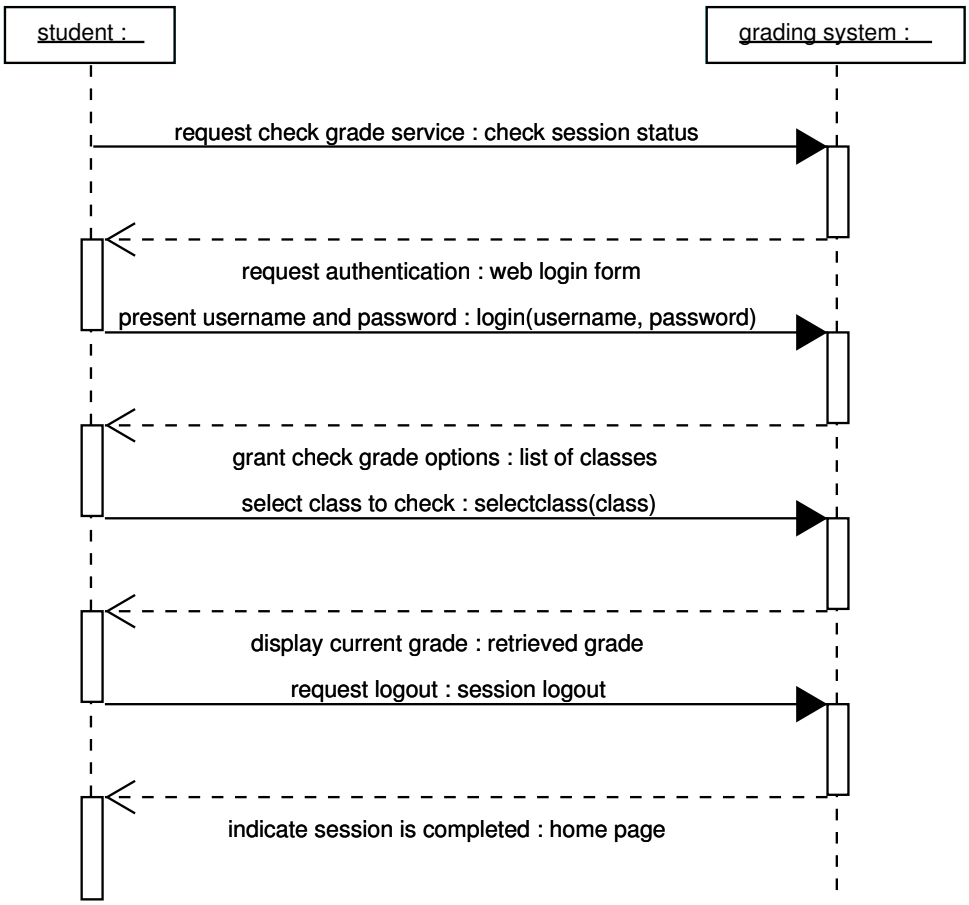


Figure 6.1: Example to illustrate a sequence diagram capturing a sequence of messages for a use case.

nothing wrong with this curiosity, the "distraction" can easily interfere with the capturing of the message exchange between an external agent (the actor instance) and the information system.

As a result, it is best to think about "*how* the system responds" and not "*why* the system responds this way". In our example illustrated in figure 6.1, it is best to just know that the system checks for an active session, but not to investigate exactly how the system tracks active session. Also, it is best to document that system *somehow* authenticates a student, but not to analyze at this point how the authentication is done.

6.4 Events

"Events" are not native to the UML. However, it is a useful concept for capturing the functional requirements of an information system. Now that we have discussed sequence diagrams, we can use them to identify events.

An event (from the perspective of an information system) is an external and asynchronous occurring that is not under the control of the information system being modeled. This means that the occurring is due to a cause that is external to the information system, and that the timing of the occurring is independent of the timing of the information system.

The information system must respond to an event with a response as soon as possible. In other words, if the system does not react to an external and asynchronous occurring, the occurring is not considered an event.

Last but not least, the response to an event must be directed to one of the actors. Note that the originator of an event needs not be the recipient of the corresponding response. Nonetheless, the response must be directed to an recipient actor.

After all this abstract discussion, we can illustrate events with some examples.

"Student checks grade" is a valid event because of the following:

- it originates from outside of the system: unless a student lives inside the information system.
- it is asynchronous: the information system has no way to predict when a student wants to check his/her grade.
- there is a response: the information system responses with the retrieved grade.
- the response is directed: the response, in this case, is directed to the originating actor instance.

A begging question is whether the user authentication is included in the event called "student checks grade". Although the authentication messages are included in the sequence diagram, they are not considered a part of the event.

Chapter 7

Interviews (Systems Requirements)

This chapter discusses common topics for interviews for the systems requirements capture phase. To be more specific, this chapter mainly discusses techniques and issues related to the capturing of functional systems requirements.

7.1 General Techniques

This section discusses techniques (and issues) that are quite general and should be applicable to most interviews.

7.1.1 Identify Objectives of Each Interview

Before an interview is performed, try to identify and document the objective(s). During this phase, an object can be "enumerate all actors", "enumerate use cases for the actor *student*" or "work out sequence diagrams for use case *check grade*".

Note that you should be able to determine if an objective is accomplished. For an objective titled "enumerate all actors", it is not accomplished until *all* actors are listed.

7.1.2 Identify Products of Each Interview

While an objective indicates what needs to be accomplished, a "product" indicates the exact form of outcome from an interview. For example, "a complete list of actors, along with descriptions and examples" is a possible product from the objective "enumerate all actors".

Objectives help an interviewer maintain focus when an interview is being planned, while a list of products help an interviewer accomplish the objectives throughout and after an interview.

For some interviews, it is advisable to identify *intermediate* products and *final* products. For example, if an objective is "capture sequence diagrams for the use case *check grade*", an interviewer may not have enough time to draw the diagrams during the interview. Instead, the intermediate product, produced during the interview, can simply be a multicolumn text document (or spreadsheet) that describes the sequences of messages. The final product, on the other hand, is a set of sequence diagrams drawn in UML CASE tools.

7.1.3 Prepare and Distribute an Agenda

An interviewer should always have an agenda for each interview. The agenda needs not be very specific both in terms of topics and time allocation. Just *having* an agenda often keeps an interview or meeting on track and saves hours of time wasted on digression and irrelevant topics. Furthermore, an agenda also reminds an interviewer of what needs to be done so that the objectives of an interview can be accomplished.

In most cases, an interviewer should share the agenda with the interviewee(s) as soon as possible. This prepares the interviewees so they know what topics will be covered (and prepare for them). Occasionally, interviewees may provide feedback based on a proposed agenda to include additional topics, include additional interviewees and etc.

It is best to allocate time for each segment of an interview so that all participants know how much time an interview requires. However, it is often the case that more time is needed. Use the allocated time in an agenda only as a guideline. If more time is needed to discuss a topic, *use* more time. Schedule for another interview if the allocated amount of time is insufficient.

7.1.4 Think from the Perpectives of Interviewee(s)

When an interviewer prepares for an interview, he/she should consider the interview from the perspectives of the interviewee(s). How much technical background can you assume? How much details does each interviewee possess? How is each interviewee involved in the information system?

7.2 Functional Requirements Specific Topics

This section discusses topics that are specific to the capturing of functional systems requirements.

7.2.1 Actor Centric versus Use Case Centric

You can capture functional requirements in an actor centric manner or a use case centric manner. The former means you first list the actors (classifications of external agents interacting with the information system), then you explore use cases for each actor. The latter means you start with use cases, then you list the actors who participate in the use cases.

In reality, it is usually a little bit of both. The process may start with one actor and some use cases participated by this one actor. As the use cases are introduced, additional actors who participate are introduced. During the process of listing actors and use cases, there is no need to strictly follow the order of actors first or use cases first.

After the actors are use cases are listed, however, it is important to double check to make sure no actor nor use case is left out.

7.2.2 Key Points of Actors and Use Cases

An actor is a classification, not an entity or an object. An actor describes/define a group of one or more individuals. As a result, the name of an actor should be general. For example, "student" is general, "Joe the straight A student" is not.

To test whether X is a good name for an actor, ask (yourself and possibly the client) "can X describe more than one individual entities?" This applies not only to actors representing real users, but also actors representing external systems. For example, "enrollment information system" is general and it describes any system that is used to maintain enrollment information. "Los Rios District PeopleSoft System", however, is specific and it describes exactly one information system.

A use case must be invoked (triggered, started) by an actor, it must be complete, and it must provide value to at least one actor. The name of a use case is always an action. "Check grade" is a use case, "grade report" is not. Recall that "complete" means the use case must be able to standalone and not only used as a component of other use cases.

When you think of a use case, think and ask your client in terms of "what does the actor (or do the actors) want to *accomplish*? What purpose is being fulfilled by the system?" Also, ask "can an actor *just* invoke this action/use case without doing anything else (in a single session)?"

Recall our infamous example of "log in". Although "log in" seems to accomplish some purposes (such as "gain access" and "get support utilities installed), such purposes cannot standalone. The access gained is used to retrieve some classified information. The installed support utilities are used to view/print documents. Ask the question: is it a means, or is it an end?

Your client may insist that "log in" be captured as a use case because "otherwise how is an user going to check account information?" In other words, a client may worry that if a process or procedure is not listed as a use case, it will never by captured. You need to explain that "log in will be captured by other diagrams as one of the steps to check account information."

7.2.3 Identifying Inheritance/Generalization

Although the words inheritance and generalization sound abstract and "object-oriented", you can explain it to your client fairly easily. To say that "student generalizes CIS student", it is the same as saying "every CIS student is also a student". It is natural that "a CIS student can do everything a student can".

The advantage of having your client understand inheritance is that your client can help you identify such relationships among actors.

With use cases, it is a little more difficult to identify inheritance. You may need to perform the analysis without the help of your client. On the other hand, sometimes your client may say "to process a store credit refund is *a special case of* a refund, with the exception of ...". This is a good clue that "process refund" is the generalization of "process store credit refund".

Since your client is unlikely to be an programmer/analyst familiar with object-oriented programming or the UML, you need to look for key phrases other than "generalization", "specialization", "subclass" and etc. Here is a short list of phrases that suggest inheritance/generalization:

- "A is a special case of B": it suggests that A inherits from B.
- "A is similar to B, except that ...": this is a tricky one. It usually suggests that A inherits from B. However, it can also indicate that both A and B are special cases of a common source C. If the exception is that A has an additional operation or property, A inherits from B. If the exception is simply a difference, then A does not inherit from B, nor does B inherit from A. It is the responsibility of the analyst to create a common ancestor, C, that describes everything that is common to A and B.
- "A can do everything B can": this suggests that A inherits from B.

7.2.4 Identifying Inclusion

Inclusion is *not* inheritance. We discussed this in the previous chapter, but it is worthwhile to mention again.

The inclusion relationship only applies to use cases. When use caes A includes use case B, use case B is a component of use case A. Whenever an actor invokes use case A, it automatically invokes use case B. Use case A, however, is not allowed to modify the behavior of use case B.

For example, an online ordering system may have a registry to keep track of active customers. There may be a use case called "verify account information" that lets the actor "customer" verify the information of an registered account. Another use case, called "order", may include "verify account information". This means that when an actor verifies account information while placing an order, the process to verify account information must be identical to the verification of account information as an independent use case.

The following is short list of phrases that suggest inclusion:

- "A involves B": if use case A involves use case B, it is likely that use case A includes use case B.
- "B is a part (component, ingredient, subtask, subprocess, etc.) of A": this is obvious.
- "when A is invoked, B is automatically invoked": this is also obvious (that A includes B).

7.2.5 "What" versus "How" (no, not Sacramento street names)

Focus on what the system is supposed to do, not how it is going to do it.

Not only must you focus on "what" and not "how", you also need to keep the interviewee focused during the interview. You can ask the interviewee to simply treat the information system as a "black box" that "magically knows how to do something". What does an actor want the genie black box to do (in the context of being an information system, that is).

One question you can ask (yourself or the client) is "does an actor care that/about X". If the answer is "yes", X may be a use case. If the answer is "no", X is probably logic internal to the information system that should not be captured as a use case. For example, does student (the actor) cares about check grade? The answer is obviously yes. One the other hand, does the manager actor (Dilbert's pointy hair boss being an instance) care about retrieving data from an SQL server? The answer is "no, the information system is a mysterious black box as far as the manager actor is concerned."

Chapter 8

Requirements Capture Assignment

In this assignment, you are going to capture the functional requirements of an information system. Since this class has no prerequisite, I cannot assume everyone is familiar with a real information system. As a result, your assignment has to do with an information system that you can easily understand and experiment with.

8.1 Choosing an Information System to Model

Choose an information system that is actually useful to you. For example, the online system of a bank, or an online auction system, or an online bookstore, are all perfect examples of information systems useful to everyday people.

8.2 Choosing Actors

The problem with choosing an online banking system is that you can only see it from the perspectives of a registered customer or a browsing user. Nonetheless, that's enough. The same applies to an online bookstore or an online auction site.

To be more specific, you need to include at least two actors: the "casual browser" and "registered user". I will leave it up to you to define the "generalization" relationship.

8.3 Use Cases

This is the part that you really need to work on. The use cases depend on which system you choose. Some use cases of an online auction site are different

from some of the ones of an online bookstore. However, certain use cases can be common.

Try to include as many meaningful use cases as possible. "Browse" is okay as a use case, but it is not particularly meaningful. In the case of an online shop, "Look for product information" is a more meaningful use case because it is more purposeful.

Choose at least four use cases to document in a use case diagram.

8.4 Scenarios

Choose two use cases (out of the four from the previous section), and for each use case, document at least two different scenarios. The two scenarios must be significantly different. "Significantly different" means the two scenarios illustrate different ways that the system interacts with an actor (or actors) for the same use case.

8.5 What to Turn in?

Turn in the following:

- A use case diagram with at least two actors and four use cases. use at least one generalization relationship for the actors. If you want to, you can also use inclusion and generalization for use cases.
- Four sequence diagrams. Choose two more purposeful use cases, for each use case, construct two sequence diagrams. Ensure the sequence diagrams for a use case are different to illustrate two possible but different scenarios.

I prefer email submissions. However, if you decide to submit by email, you have to choose one among the following forms:

- Poseidon `.zargo` format.
- PDF format.
- Postscript format.

I cannot guarantee to be able to open any other formats.

Submit your homework assignments to mailto:auyeunt@arc.losrios.edu with the subject line "CISP457 Assignment2 by *your name*". If you do not use this subject line exactly it is described, it can be "lost" in my mailbox when I sort messages by subject.

If you decide to submit by paper, make sure I time stamp your assignment with the turn in date.

Chapter 9

Cost Benefit Analysis

The bottom line to the client is how much to pay and what the return is. The analysis of cost versus benefit of an information system is usually considered a part of the responsibilities of a systems analyst.

9.1 The Cost

The cost analysis of an information is not a static and single number. For example, 11 million dollars mean nothing as the cost of an information system. This section discusses the various aspects of cost analysis.

9.1.1 Cost Components

There are many components to the cost of an information system. There are mainly two categories: recurring and non-recurring. A recurring cost is one that repeats. For example, the maintenance cost of the hardware and software of an information system is recurring. A non-recurring cost is one that does not repeat, or one that does not have a *scheduled* time table. For example, the cost of the initial design and development of an information system in non-recurring.

Within recurring cost components, there are several common ones:

- Salaries of human components. System administrator, system technicians, operators, IT supervisors.
- Maintenance contracts for software and hardware components.
- Cost of services: ISP, data backup.
- Other bills: electricity, rent.
- Cost of *scheduled* upgrades: workstation upgrades, server upgrades, OS upgrades.

- Cost of training: based on average turn-over rate of operators and other human components.

Within non-recurring cost components, the following are common:

- Initial cost to design, develop and deploy.
- Initial cost to convert: include training cost of users.
- Initial cost to purchase the system and its components.

9.1.2 Every cost is a function of time

Even non-recurring cost is a function of time. For example, the cumulative cost of the development of a system depends on the phase or milestone in the timeline of the project.

It is important to remember that cost is a function of time because a static cost does not provide much value for planning purposes. If a project costs 11 million dollars, does the client need to come up with 11 million dollars up front? Or, can the client pay the 11 million dollars over a long period of time? If so, how can the client pay the 11 million dollars over time?

The time factor of cost has a big impact on cash-flow analysis and cumulative payback analysis. Both topics will be discussed later in this chapter.

9.1.3 Dollars of which year?

For any information system that is designed to work for many years (5 or more), it is important to understand what a dollar means. Because of inflation, a dollar in 2003 does not have the same value as a dollar in 2008.

It is common for analysts to use dollar value based on the year when the project is initiated. In other words, if a project is initiated in 2003, all dollar amount refer to the dollar value in 2003. This makes it easy for everyone involved to compare the cost of the system to other expenses and profit.

It is not uncommon for an analyst to assume an average inflation based on historical data. For example, an analyst can choose a inflation rate of 2% per year. If i stands for the average inflation rate, a dollar n years later is worth $\frac{1}{(1+i)^n}$ of the value of a dollar today.

Non-recurring costs are typically insensitive to inflation because most such costs are spent in the early years of an information system. Recurring costs, however, are highly sensitive to inflation. As a result, it makes perfect sense to run recurring cost analyses with different inflation assumptions.

Find out, from historical data, what is the worst case average inflation over a contiguous five-year period. Use this worst case inflation for the analysis of an information system that has a lifespan of 10 years. The resulting cost will be very conservative as far as sensitivity to inflation is concerned.

Provide your client with an average inflation cost and a worst-case (conservative) cost. This way, your client understands how the cost *may* differ. This

analysis can be the deciding factor of two systems that have the same average cumulative cost, but one may have a much higher cost based on the worst-case inflation assumption.

9.1.4 Cumulative cost versus cost rate

Cumulative cost is the summation (integration) of cost over time. For example, if the cost of the first year is 3 million dollars, and the cost of the second year is 2 million dollars (inflation adjusted), the cumulative cost at the end of the first year is 3 million dollars, whereas the cumulative cost at the end of the second year is 5 million dollars.

Cumulative cost analysis is helpful for the purposes of cash flow analysis as well as break even point analysis.

Most people are more accustomed to the concept of cost rate. For example, "the system costs $12,000 (inflation adjusted) per month to operate." This is a rate at which funds is spent. Cost rate is useful when a client wants to get an idea of the cost of the system when it is in operation phase.

Note that it is easy to compute cost rate from a cumulative cost curve. For example, if the cumulative cost 4 years into the system's lifespan is 5 million (inflation adjusted) dollars, and the cumulative cost 5 years into the system's lifespan is 6.4 million (inflation adjusted) dollars, the average cost rate is $\frac{6.4-5}{12} = 0.1166$ (116.6 thousand inflation adjusted dollars) per month (between year 4 and year 5 of the system's life).

You can also provide an amortized system cost rate. In our previous example, the amortized system cost rate in year 4 is $\frac{5}{4*12} = 0.0833$ (83.3 thousand inflation adjusted dollars) per month. This figure, however, is hardly of any practical use.

9.1.5 Intangible Costs

I'd like to call this "prioritized don't-wants". Besides the monetary costs of an information system, there are costs that do not directly have an associated dollar amount. For example, "some operators may feel intimidated by the new workstations." Unless you can directly associate this intimidation to a quantified loss of work efficiency, it is an intangible cost.

Instead of playing number games with intangible costs, it is easier (and more sensible) to simply prioritize intangible costs. For example, the client can simply indicate that "I'd rather have some operators being intimidated than to have the systems limited to run an OS supplied by a software monopoly." Let us assume a proposed system calls for workstations that may intimidate some operators but run a free OS. Let us assume an alternative system uses non-intimidating workstations that run an OS supplied by a software monopoly. According to the prioritization of intangible costs, the first system is chosen (if all other more important factors are identical).

9.2 The Benefit

This section discusses various aspects of the benefits of an information system. Be warned, however, that the analysis of benefit is not nearly as easy as that of the cost.

9.2.1 What profit?

A common mistake is to think that an information system is going to make enough money, at some, point, to "start to make money". An information system in a client organization is a tool that is used for its business. As a result, it is *seldom* that an information system can general positive cash flow.

This (positive cash flow) is possible only when the services provided by the system directly generates revenue. For example, each completed transaction of eBay charges a percentage of the sold price, and this revenue is directly generated from the use of the information system. On the other hand, amazon.com only uses the information system to help sell books. The revenue generated from book sales is a result of the application of an information system.

For *most* clients, the use of an information system helps to streamline processes and improve efficiency. This means an information system does not directly relate to the generation of revenue for most clients.

So, what exactly is counted as benefits of an information system?

9.2.2 Everything is relative

The benefit of an information system is often *relative* to another information system. For example, the current system may have an inflation adjusted cumulative cost of 20 million dollars for the next 10 years. A new information system, on the other hand, may only cost an inflation adjusted cumulative cost of 12 million dollars for the next 10 years. As a result, the benefit of the newer system is *saving* an inflation adjusted 8 million dollars *after* 10 years.

This "relative" perspective helps to isolate the benefit analysis from the business of the client completely. In other words, you do not need to know the percentage charge of each auction, you do not need to know the book selling business and etc. All you need to know is the current system (or other alternative systems) and the one being proposed.

9.2.3 Every tangible cost is a tangible benefit

Each tangible cost of an information is a basis for comparison against another information system. As a result, for each line item of tangible costs, you can analyze the relative benefit of an information when compared to another information system.

9.2.4 Intangible Benefits

Instead of assigning weighs to intangible benefits and play number games, it is better to prioritize "nice-to-haves" along with the "don't-wants". Note that a "nice-to-have" is the negation of a "don't-want". For example, "prefer to have a free OS environment" is the same saying "prefer not to have a non-free (proprietary) OS environment".

9.3 Cash Flow Analysis (?)

It does not make sense to perform a cash flow analysis with respect to an information system all by itself. This is because, as discussed before, that an information system normally does not directly generate revenue. As a result, for most organizations, either a cash flow analysis is performed with the entire business, or it is not done at all.

Note that you can take the *relative* tangible inflation adjusted cumulative benefit and use that to perform some form of cash flow analysis. The meaning of such an analysis is, however, questionable.

9.4 Cross Over Point

Although a cash flow analysis is meaningless without considering the rest of the business of a client, it does make sense to analyze a cross over point with respect to another information system.

Using the inflation adjusted cumulative cost charts, it is quite easy to visualize the break even points. Simply overlay the inflation adjusted cumulative cost charts of two systems (scaled identically with respect to the same dollar value). The intersection of the two systems is the cross over (usually) point.

The meaning of this cross over point is when a system start to be less expensive (cumulatively speaking) than another system.

Although logically, only the final cumulative cost over the expected lifespan of an information system should matter, the cross-over point has its significance. For example, let us assume system A has the same final cumulative cost over the expected lifespan as system B. Compared to the current system, C, however, A has a cross-over point that is 4 years from now, whereas B has a cross-over point of 6 years from now. Given everything else being equal, A is better than B because it is "more likely" to save money, even if the system prematurely get obsoleted in 5 years.

Chapter 10

Cost Estimate

Let's get one thing straight: if I have a method by which the cost of a project can be estimated accurately, I would have been rich, *very* rich. Many people have different models to estimate the cost of information system development projects, but all boils down to experience, common sense and few techniques.

This chapter attempts to discuss various factors that affect the cost of the development of a project. It does not, however, even pretend to give you the crystal ball that predicts information system project costs.

10.1 Life Cycle

The "life cycle" of an information system is the period starting with the conception and ending with the obsolescence. Of the entire life cycle, the initial cost to design, develop and deploy the system may not be significant (when compared to the cost of operation). However, because most clients pay much attention to the initial cost because of the following reasons:

- For start up organizations, this is the most critical time for cash-flow management. The operation cost normally affects the profitability, but the initial development cost affects the survival chances.
- The cost, in terms of time, of the development phase affects time-to-market and competitiveness.
- Most failures occur in the initial phase of development.
- The initial cost is the most unpredictable, whereas the operation cost is usually more predictable.

As a result, even though the operation cost is also important, more emphasis is directed to the initial development stage of an information system.

10.2 the Initial Phase (Pre-Operation)

10.2.1 Processes

There are three distinct processes that take place in the pre-operation phase of a project:

- Requirements Capture
- Analysis
- Design (Synthesis)
- Build (Code)
- Verify (Test)
- Repair (Debug)
- Demonstrate

Requirements Capture

This is the process to figure out *what* the system is supposed to do. An alternative view is *what* external agents (actors) expect from the system. As discussed in earlier section, requirements are separated into non-functional ones and functional ones. Functional requirements can be captured by use-case diagrams.

The key result of this process is a document that describes, from the perspective of the client, the minimum non-functional requirements of the information system. This process also produces a document that captures the functional requirements of an information system. The results of this process should be approved by the client before other processes can begin.

Analysis

Based on the requirements captured in the previous process, the analysis process breaks each requirement into detailed, technical and verifiable specifications.

For non-functional requirements, the resulting specifications describe the minimum specifications of systems components. For example, if a non-functional requirement state that "the maximum total number of transactions to track is 2,000,000 per year", the technical specifications will be the minimum mass storage requirement (such as 5GB for the storage of transaction records). The derivation from 2,000,000 transactions to 5GB must be sound and included in the document.

For functional requirements, the resulting specifications describe the behavior of the information system such that it is verifiable. This means the behavior specifications completely define the "correct" behavior of the information system. For example, a sequence diagram produced by the requirements capture process may simply indicate that a user is locked out of the system after several

failed attempts to log in. The analysis process needs to specify how the system responds to failed log in attempts and how it determines an account should be locked out.

Note that even at this stage, the information system is still being seen as a black box. The only difference from the requirements is that the specifications are precise and there is no room for ambiguation. Most specifications produced in this process are too technical for a client to understand. However, client interaction may still occur to clarify the logic of systems behavior.

Design (Synthesis)

The design process takes the specifications from the analysis process and synthesize a solution that meets the specifications. For example, if the total mass storage requirement is 50GB (as a specification from the analysis process), the design process determines what kind, brand and model of storage subsystems possess the minimum specifications. Note that it is not good enough to only find one possible solution. The design process should research alternative solutions whenever it is possible. The design document then compares and analyzes the alternatives in terms of a set of criteria.

For behavioral specifications, the design process determines the organization and structure of the system. The design process also translates behavioral specifications in many forms to the few formal forms used by CASE tools. For example, the behavioral specification for failed login can be a truth table for simplicity and clarity. The design process takes the truth table and convert it either to a state diagram or an activity diagram in the UML. The design process also formalize the internal structure of an information system in terms of classes, methods, events and other object-oriented concepts.

Build (Code)

For hardware and system level components, the build process assembles and integrates the components. For example, for hardware components, the build process includes the acquisition and assembly of components (such as servers, mass storage subsystems, networking infrastructure and etc.). For system components, the build process includes the acqusition, installation and integration of components (such as the OS, database server, http server, middleware interpreter and etc.).

For the software design, the build process means coding. This is, in most cases, a mechanical translation from the diagrams produced by the design process to actual code that can be interpreted or compiled.

Verify (Test)

The verify process ensures the resulting system (from the build process) meets the specifications produced by the analysis process. Because all the specifications are supposed to be "verifiable", the verify process does not sound very difficult, at least for hardware and system components.

To verify behavioral specifications, however, is not as easy as reading the actual specifications of a component and comparing that to the specifications produced by the analysis process. A single behavioral specification can yield an infinite number of possible scenarios to verify. For example, consider the following pseudocode:

```
repeat
  ask user for a non-negative number
until a non-negative number is entered or the user cancels
```

The user may enter a negative value once, twice, three times and etc. before entering a non-negative number or click "cancel". As a result, it is important to produce a test plan for each behavioral specification. A test plan should specify a finite number of actual tests that, for all practical purposes, verifies the corresponding behavioral specification.

In our simple example, the test plan may have the following individual tests:

1. Action: enter a non-negative number. Question: does the system advance to the next step? Expected Answer: Yes.
2. Action: enter a negative number. Question: does the system prompt again? Expected Answer: Yes.
   (a) Alternative Action 1: enter a non-negative number. Question: does the system advance to the next step? Expected Answer: Yes.
   (b) Alternative Action 2: enter a negative number. Question: does the system advance to the next step? Expected Answer: No.
   (c) Altenrative Action 3: press "cancel". Question: does the system abort the operation? Expected Answer: Yes.

It is possible to write a program that passes these tests that does not, in general, behave as specified. However, the likelihood is very slim because the program has to be specifically designed to defeat the tests!

Software verification and testing is a very interesting and deep subject matter all by itself. What we have described in this section is merely scratching the surface.

The result of the verify process is a list of violated specifications (if any). If no specification is missed or violated, we can potentially move to the demonstrate process. Otherwise, we need to continue with the debug process.

Repair (Debug)

This process applies only to the code of an information system. If the verify process yields failed tests, the implementation does not match the specifications. These failed tests should be distributed to the coders so the coders can debug the code so it passes the tests. Note that the test *plan* should not be given to the coders, only the violated specification should be indicated. This is because exposing the test plan may lead to the coders fixing the programs only to pass the tests.

Demonstrate

The demonstrate process shows the client the (partially completed) system. This process should occur after the verify and repair processes for obvious reasons (remember Bill Gates standing in front of the blue screen of death?).

Many models do not include "demonstrate" as one of the processes in the pre-operation phase. *If* the requirements capture and analysis processes are done perfectly, it is true that there should be no need to demonstrate a partially completed system. We can simply begin with beta-testing (on site client testing) when the system is fully completed.

In reality, however, the demonstrate process serves many valuable purposes:

- The client gets periodic updates so he/she does not feel as insecure as would have been.
- The client gets a chance to point out misunderstandings and clarify ambiguity in demonstrations.
- The client can provide feedback that can improve the information system (at the expense of potentially restarting the requirements capture and analysis process).

10.2.2 Process Flow Models

Now that we have identified the various processes, this section introduces the different models that guide a team through the pre-operation phase.

Chaotic

The chaotic model menas no particular model, or the lack of discipline to use a model.

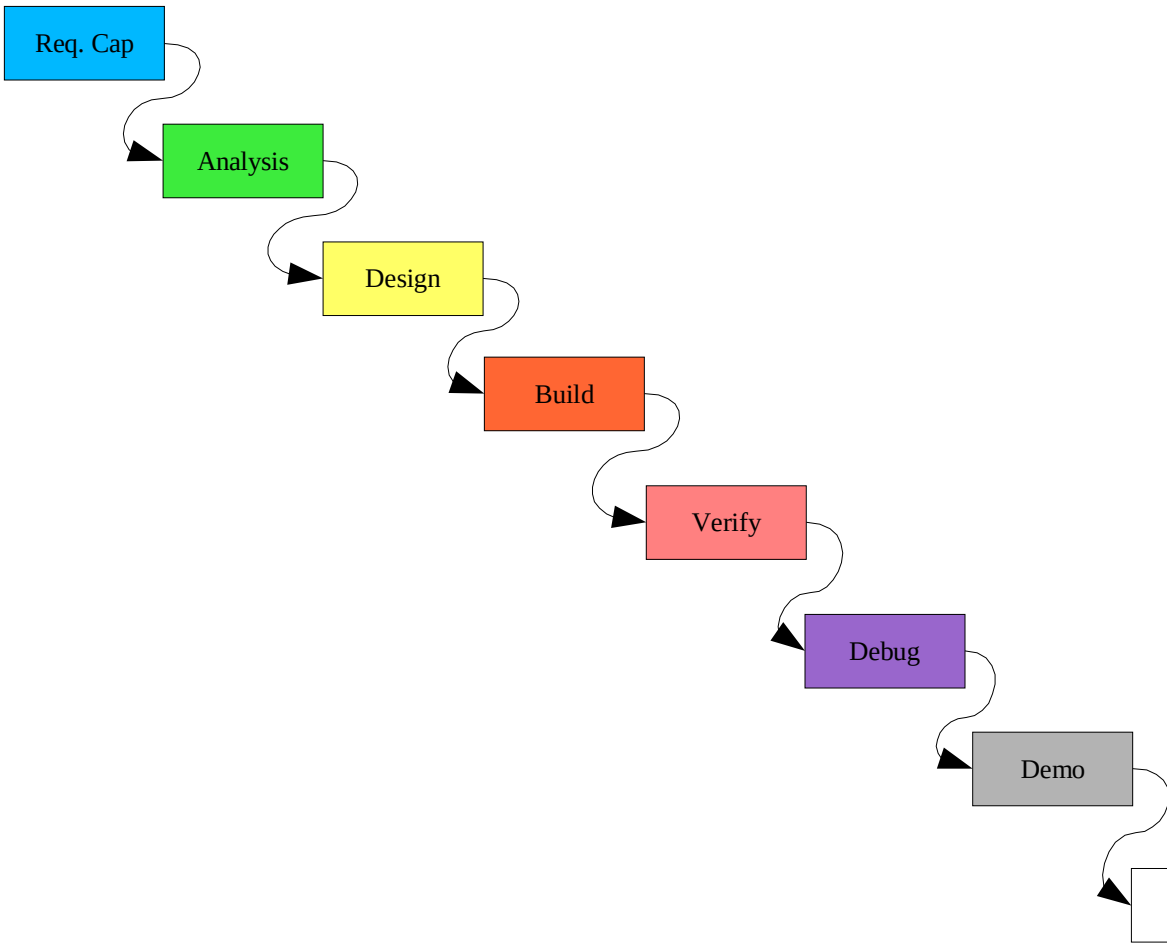
Without a process flow model, work is driven by incomplete and/or incorrect understanding of the requirements. This translates to frequent changes and/or over budget situations.

Strictly Sequential

In a strictly sequential model, processes must follow a strict order. The order of one development cycle is listed as follows:

1. Requirements Capture
2. Analysis
3. Design (Synthesis)
4. Build (Code)
5. Verify (Test): skip repair is there is nothing to fix

6. Repair (Debug): goes back to verify
7. Demonstrate: goes back to requirements capture or analysis is it is not satisfactory

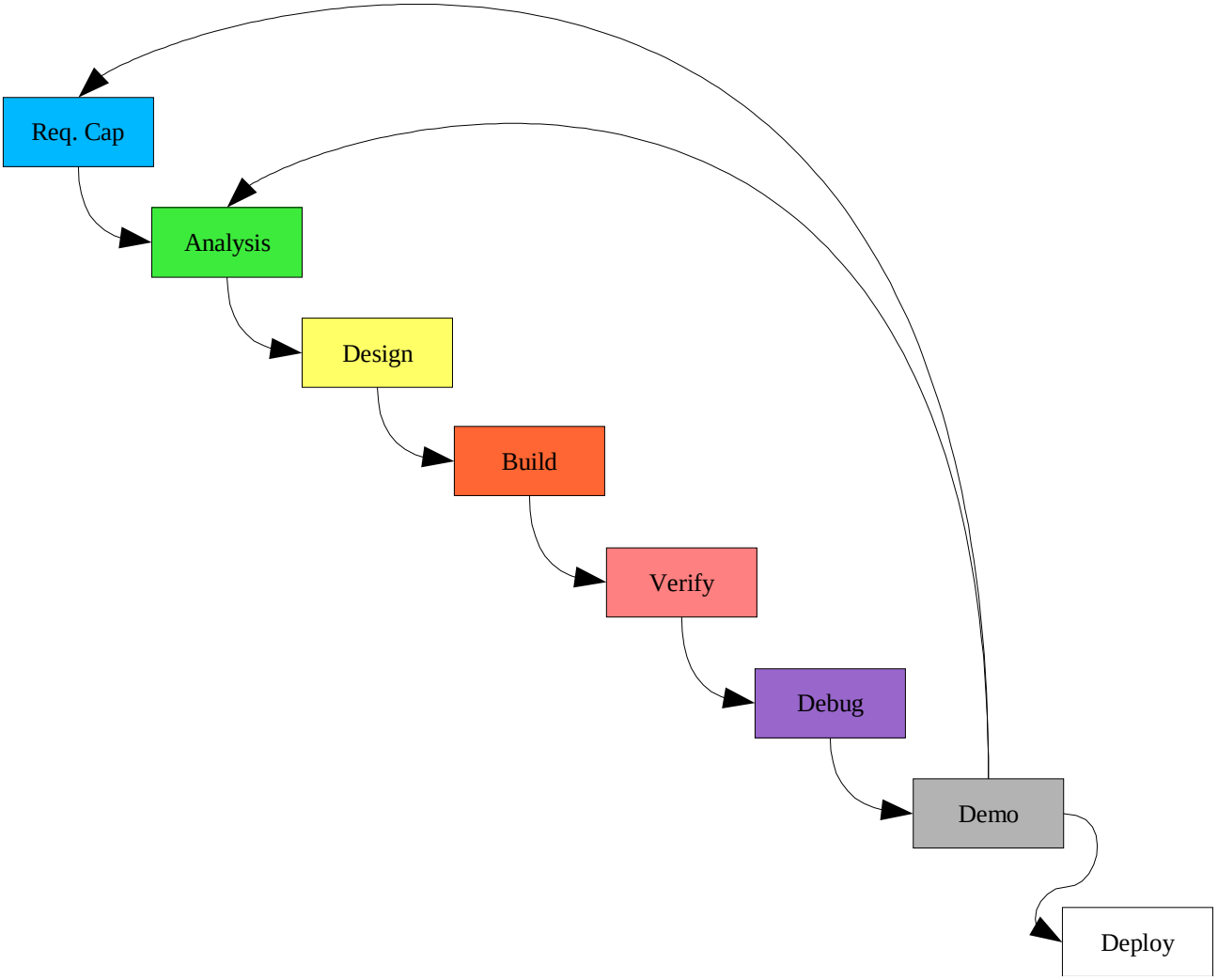


This model is also called "waterfall" because in the presense of gravity and no other energy source, water only flows down.

While this model sounds good on paper, it often has real-life difficulties. This model relies on the ability to capture all the requirements correctly and perform the analysis and design processes *completely and correctly* the first time. If anything is done wrong, it is a very long cycle to get to the "demonstrate" process. Much resources are wasted this way.

For smaller projects in which the whole development cycle is not too long, this model is not bad. However, for huge projects, the lack of client feedback until everything is done can lead to many problems. A slight change or addition to the requirements can easily impact every other process, which voids much effort from a previous cycle.

Overlapped



The overlapped model is more commonly known as RAD (rapid application development), prototyping or rapid prototyping. This model is characterized by the overlapping of processes. For example, when the analysis process is still in progress, the partial specifications are used to start the design process, and before the design process is completed, the resulting documents are used build partial programs. The partial programs are tested (partially) and then demonstrated to the client.

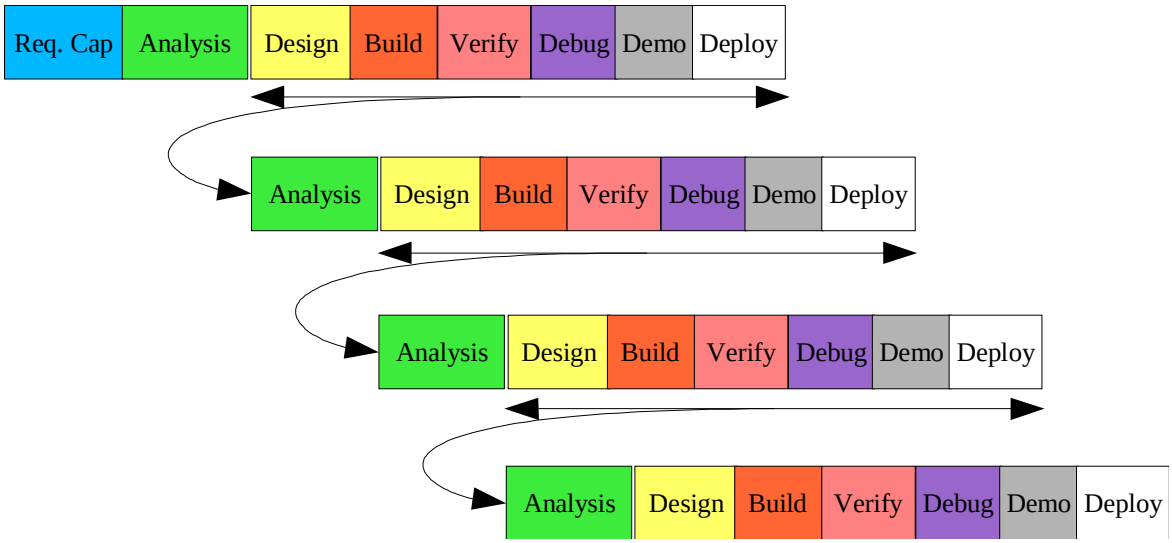
The strength of this approach is the relatively short period between demonstrations. It allows the client to provide feedback frequently so that waste (due to incomplete or incorrect requirements) is limited. This model also provides many opportunities to experiment with the system so the client stays more in

touch with the project.

The weakness of this approach is that it is difficult to track versions of documents, and the client can now easily ping-pong the requirements.

This model is best suited for cases in which the client does not quite know what to require in the first place, and that the need to get the system fully operational is minimal. This model should not be used when delivery time is critical.

Layered



The layered model is also called "incremental". In this model, the system is divided into layers. Except for the innermost layer, the core, all other other layers may have some dependencies upon an inner layer.

The innermost layer, the core, must be fully analyzed first. Once the core layer has a specification, the specification of the second layer is analyzed *while the core is being designed and built.*

The same applies to the second layer, once it is specified, the team starts to analyze the third layer *while the second layer is being designed and built.*

As each layer is built and tested, the client can actually *take delivery* as the out layers are still in progress. This means the functionality of the entire system is now delivered in sections.

The main advantage of this model is the relatively short time to get the core delivered. Instead of delivering everything, including the very infrequently used functions, the system is divided into sections. The sections are determined by mainly two criteria: dependency and priority.

Dependency is determined solely by the interdependency of modules and features. Priority, on the other hand, is determined by the client's needs.

Part III

Analysis

Chapter 11

Behavioral Analysis

The term behavioral analysis refers to the analysis process that produce the system behavioral specification. This specification completely and unambiguously describe the logic that dictates how an information system behaves.

The behavioral specification of an information system can be written using a mixture of diagrams, tables and forms. This chapter discusses the process itself, as well as some of the more common diagrams, tables and other forms for representing logic.

11.1 The Process (for Procedural Analysis)

11.1.1 Where to Start?

At this point, the functional requirements are already determined, and some representative sequence diagrams should have been constructed to illustrate use cases. The leap from captured scenarios to the logic that describes the system's behavior is not a small one.

With object-oriented concepts, the analysis process itself is "object-oriented". In other words, it begins with questions like "what objects exist in the system", "what can this object do?", "what other objects interact with this object?" and etc.

Without using object-oriented concepts, the analysis process is more procedure/process oriented. In other words, it begins with questions like "the system needs to authenticate a user, how is it going to do this?"

This step is one of the most difficult part of systems analysis. However, there are some commonly used techniques that can be helpful.

Start with use cases. Each use case is "a use of the information system that is complete and provides value to at least one actor." For example, consider the use case "check grade" for the student actor. This use case provides value because it retrieves and displays (or downloads) the grades of a student.

One the objective(s) of a use case is identified, an analyst can then try to figure out how it can be accomplished. To retrieve the grade of a student, the

system needs to know the identity of the student, as well as an optional list of classes or a time frame. However, because a person can easily acquire the student ID of someone else, the process must also authenticate the student so that privacy can be maintained.

Do not get bogged down by details at the beginning. Simply identify the *main* steps of a process. Try to keep the list of main steps down to 10 or fewer so that the list is easy to view.

Even at this stage, try to identify key results and requirements of each step in a process. For example, the "check grade" may have a process identified as "retrieve grades from database". The requirements of this step include the student ID and a list of classes or a time frame. The results of this step is a list of classes with the associated grades, probably in a simple comma-delimited format.

The identification of requirements and products of each step helps to define the *interface* of each step. This helps to isolate each step so it remains independent to other steps.

Note that the requirements and results can be qualified by conditions. For example, instead of just indicating that "student ID" is required, the step can specify that the "student ID" be authenticated.

11.1.2 Top-Down Design

Top-down design is commonly known as "divide-and-conquer". This is a powerful technique because it ensures that at any particular time, the amount of details that an analyst needs to maintain (in short-term memory) is manangeable.

Note that top-down can be combined with the "start with the ends" approach, but these two approaches are not the same. To "start with the ends" means you start with the *final* product/result of a process, and you work your way back to the beginning, identifying intermediate results along the way.

Top-down design, however, means that you only keep track of up to 7 to 10 (some suggest 5 to 7) steps at any moment. Whether you come with with the 7 to 10 steps using a forward reasoning technique or the "start with the ends" technique does not impact top-down design.

For example, the "check grade" use case may be broken into the following *big* steps:

1. authenticate student
2. present selection criteria types
3. let student choose selection criteria
4. let student choose display/sorting method
5. inquire database
6. format data and display

Note that the steps are *ordered*. In other words, "present selection criteria types" must occur before "let student choose selection criteria".

After each iteration of the application of top-down design, you can then take one step, and reapply the divide-and-conquer technique and break a step into finer steps. This process is repeated until each step is clear enough so that it cannot have more than one interpretations.

11.1.3 Add Logic

Although there is logic in steps in a sequence, it is solely determined by dependencies. Other types of logic, including conditional (decision) and iterative (repetitive) logic, adds complexity to the process.

Conditional logic is used when there are *alternatives*. In other words, whenever there are multiple ways to proceed, it is implied that conditional logic may be needed. For example, a student may authenticate using a username-password pair, a USB crypto-key or a finger print scanner. As a result, the "authenticate student" step must involve conditional logic to let the student choose a method of authentication. The key question to ask for conditional logic include the following:

- What are the alternative actions to take? Only one can be selected.
- What is the condition that must be satisfied before taking each alternative action?

Iterative logic is used whenever a step may be repeated. This is often done to guarantee one or more conditions be true after the repetition. For example, a student may need to perform authentication again if an earlier attempt fails. Another way to look at this logic is that the student may only proceed to the next step if the authentication is successful. The key questions to ask for repetitive logic are listed as follows:

- What action is being repeated?
- Does the action needs to be performed at least once?
- When does the repetition stop?

11.2 Logic Representation

There are many ways to represent logic. This section discusses many alternatives to represent logic.

11.2.1 Flow Chart

Flow charts are perhaps the oldest of all representations of logic. In a flow chart, control flow is indicated by directed arrows. Processes are presented as rectangles, decisions are represented as diamonds.

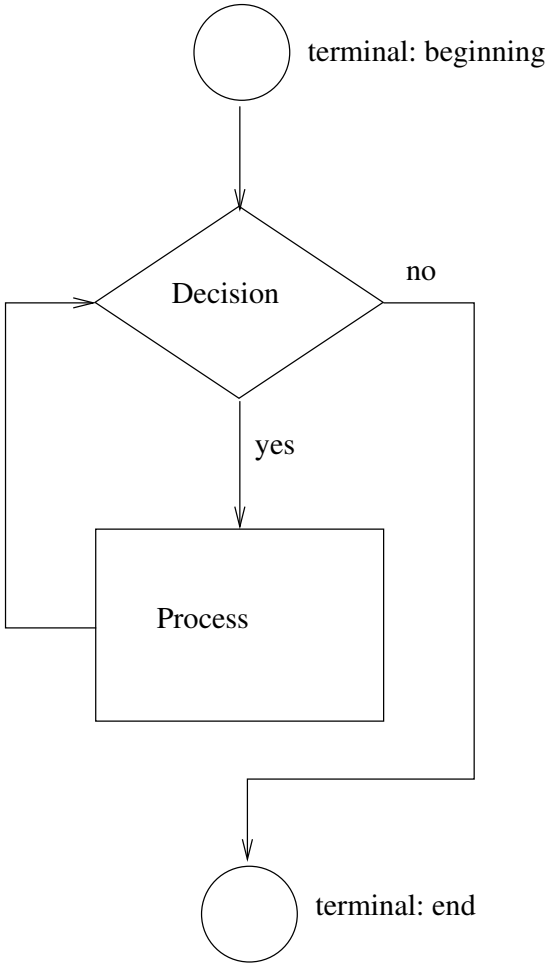


Figure 11.1: An example of a flowchart.

Figure 11.1 is a sample flowchart.

Besides the symbols used in figure 11.1, there are many additional symbols. However, we are not going to describe the other symbols in this class.

Flowcharts are easy to read because logic flow is explicitly indicated by arrows. However, flow charts do not encourage structured approaches, and they are relatively tedious to create.

11.2.2 Pseudocode

Pseudocode is basically structured natural language. "Structured" means there are only a few basic constructs. However, a construct can be completely embedded in another construct.

The basic constructs are listed as follows:

- Sequence: a sequcne is a list of actions that must be performed sequentially as specified.
- Conditional Statement: a conditional statement allows the system to choose from two alternatives based on the outcome of a condition. A condition must evaluate to either true or false.
- Prechecking Loop: a prechecking loop is a construct for repetitions. However, before an iteration begins, a condition must be checked and confirmed to be true.
- Postchecking Loop: a postchecking loop is also a construct for repetitions. In this case, an iteration is performed, then an exit condition is evaluated. If the exit condition is false, another iteration is performed, and so on.

The ability to embed constructs makes pseudocode "structured". This is one advantage of pseudocoding over flowcharting. Furthermore, it is possible to write pseudocode without a special editor. Best of all, programmers can insert complete sections of pseudocode as comments in most programming languages. The last point helps programmers build programs that are consistent with the system specifications.

Conditional Statements

A general conditional statement looks like this:

if *condition* **then**
  true action
else
  false action
end if

In this general form, *condition* is to be replaced by a condition that is either true or false. For example, "account has non negative balance" can either be true or false. *True action* is "action" to perform if the condition is true, whereas *false action* is the alternative action to perform if the condition is false.

Note that *true action* and *false action* can be complex actions. In other words, each can be a sequence, a conditional statement or a loop.

For a simple case, when there is no *false action*, a condition statement can be simplified to the following form:

if *condition* **then**
  true action
end if

For complex cases, when there are multiple alternatives, each for a condition being true, you can use the following form:

if *condition1* **then**
  true action 1
else if *condition2* **then**
  true action 2
else if *condition3* **then**
  true action 3
else
  false action
end if

In this form, conditions are evaluated in the specified order. An action is taken *as soon as* a condition evaluates to true. *False action* is only taken when all conditions evaluate to false.

Prechecking Loop

Prechecking loop is also called while loop because in most programming languages, the reserved word "while" is used to begin such a statement.

The general form of this statement is as follows:

while *precondition* **do**
  iterative action
end while

Precondition is a condition that must evaluate to true or false, an iteration is performed only if this condition is true. If this condition is false, the loop is exited and execution continues with whatever is after the prechecking loop.

Iterative action is an "action" to be performed for each iteration. Note that this "action" can be any statement by itself, including sequences, conditional statements and etc.

Postchecking Loop

Postchecking loop is also called "do-while" loop or "repeat" because these are the reserved words used by many programming languages for postchecking loops.

The general form of a postchecking loop is as follows:

repeat
  iterative action
until *exit condition*

In this statement, `iterative action` is the action that may be repeated. It can be a sequence, a conditional statement and etc. *Iterative action* is performed first, then `exit condition` is evalauted. If *exit condition* is false, another iteration is performed. Otherwise, if *exit condition* is true, the loop exits.

Embedded Constructs

A construct can be used by itself with simple actions. For example, we can have something like the following:

repeat
  ask user for amount
until amount is less than or equal to balance

On the other hand, we can embed a construct completely within another construct like the following example:

repeat
  ask user for amount
  if amount is greater than balance **then**
    give user error message
  end if
until amount is less than or equal to balance

Note how indentation is used to indicate which statement is contained in which other one.

11.2.3 Decision Table

A decision table is a modified truth table. A decision table is useful for enumerating all possible cases, and assigning an action to each possible case. Note that decision tables cannot represent iterations.

A decision table has two parts. The first part is a group of columns, in which each column represents a condition. This first part specifies all the possible conditions to be considered. The second part is also a group of columns, in which each column represents an action to take. Each row is a "rule" that maps a combination of conditions to a combination of actions.

Note that given n unrelated and independent conditions, you can end up with 2^n rows. This means with 6 independent conditions, you can end up with 64 rows.

For complicated cases, a huge decision table can be partitioned to smaller ones. A "parent" table contains a (small) number of conditions. Instead of specifying actions, each row of a parent table refers to a "child" table. In return a child table can resolve a few conditions and refer to grandchild tables. The last descendents are tables that actually refer to actions.

Tables can be simplified. Two rows containing exactly the same action combination and condition combinations that indicate a condition is not used for decision can be merged. This process can be repeated until no two row can be merged.

Here is an example. Let's say we need to represent the following rules:

- if username exists, but number of consecutive failed login is at least 3, tell the user to call in
- if username exists, number of consecutive failed login is less than 3, and password matches, reset the number of consecutive failed login and proceed to the following step
- if username does not exist, tell user that "username/password does not match" and let the user try again
- if username matches, but password does not match, and number of consecutive failed login is less than 3, increment the number of consecutive failed login, tell user that "username/password does not match" and let the user try again

The truth table needs to enumerate all the conditions. In this case, our conditions are as follows:

- C1: does the username exist?
- C2: does the password match? Note that this condition is not independent of the first one.
- C3: is the number of consecutive failed login less than 3?

Our actions also need to be enumerated:

- A1: reset number of consecutive failed login and proceed to next step
- A2: increment number of consecutive failed login
- A3: indicate "username/password does not match" and ask user to try again
- A4: ask user to call in

Now, we can construct the decision table:

C1	C2	C3	A1	A2	A3	A4
T	T	T	X			
T	T	F				X
T	F	T		X	X	
T	F	F				X
F	T	T			X	
F	T	F			X	
F	F	T			X	
F	F	F			X	

Because our conditions are not truely independent, you can see that some rows can be merged. Particularly, if the username does not match, it makes no

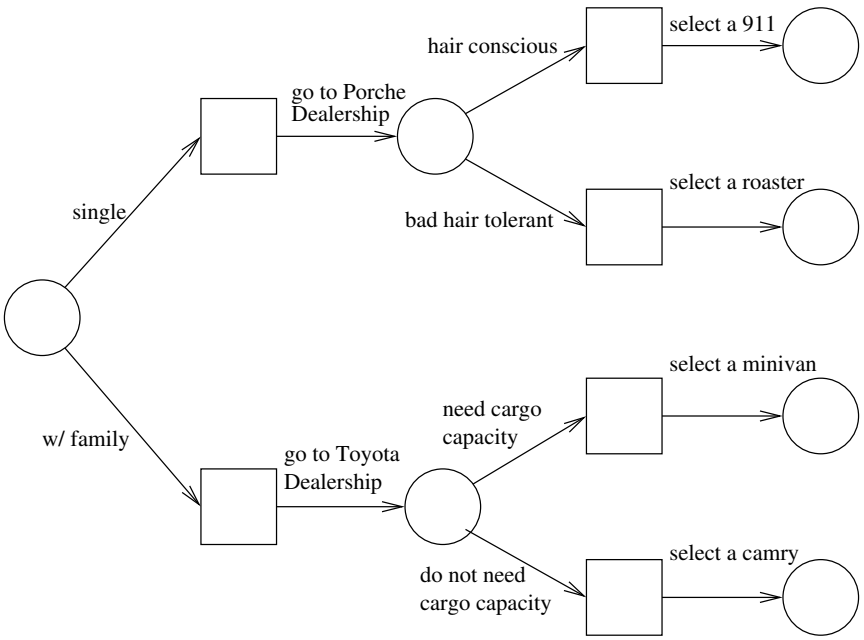


Figure 11.2: An example of a decision tree.

sense to see if the password match, or to check the number of consecutive failed login. For these situations, we can specify "?" for a condition to indicate "don't care" or "not applicable". The simplified table is as follows:

C1	C2	C3	A1	A2	A3	A4
T	T	T	X			
T	?	F				X
T	F	T		X	X	
F	?	?			X	

11.2.4 Decision Tree

Decision tables are best for enumerating all possible combinations of conditions. However, decision tables cannot express actions interleaved with decisions. Actions are often interleaved with decisions because some decisions must be made based on results of actions.

In a decision tree, a bubble represents a decision point, whereas a square represents where an action begins. A line from a bubble to a square represents a decision, a line from s square to a bubble represents an action.

Figure 11.2 is an example of a decision tree.

Like a flow chart, a decision tree is easy to follow. This is mainly because of the use of arrows to show the direction of flow. Note that decision trees typically cannot replace decision tables, nor the other way around. A decision

tree is best suited when decisions are interleaved with actions because actions produces results (conditions) that affect decisions.

11.2.5 State Diagram

A state diagram may resemble a flow chart or a decision tree, but it is actually completely different from those two presentations of logic.

Object Orientation

A state diagram is a logic representation that is suitable for object-oriented modeling. Object-oriented programming (OOP) and object-oriented modeling (OOM) are subjects that deserve courses by themselves. However, we can describe the approach in a nutshell for our purposes.

Not surprisingly, the whole idea of "object orientation" is to see from the perspectives of objects. What *is* an object, then? An object is an item that exists during the operation of an information system. An object has properties that collectively define its state, as well as methods that can be invoked.

In object-oriented programming, popular examples of objects include GUI elements. For example, a push-button is an object. It has properties such as coordinates, color and state (pushed or released). A list box, on the other hand, has many methods in addition to properties such as coordinate and size. A list box has methods to clear items inside it.

In object-oriented modeling, a common example of objects is a "session". A session is an object that has properties such as start time, user ID, session ID and time of the last activity. A session also has methods such as "connect", "discconnect", "check for timeout" and etc.

Most objects have states. For example, a "push button" object has a state of "pushed" and a state of "released". A session object has the states of "disconnected", "authenticating" and "connected". A state diagram illustrates how an object can change from one state to another.

What is a state?

According to *Webster's Revised Unabridged Dictionary (1913)*, a state is (meaning useful in our context) "The circumstances or condition of a being or thing at any given time."

For example, water molecules collectively can be in at least three physical states: solid (ice), liquid (water) and gaseous (water vapor). At any give time, a small group of water molecules can be at any one of these three states.

Note that what disciminates one state from another is always chosen. For example, in the case of water molecules, one can include volume as a part of the state. As a result, there is an infinite amount of frozen states because ice shrinks (like most other solid objects) as temperature decreases.

As a result, the creator of a state diagram must determine what distinguishes one state from another based on context. If it serves no practical purpose to

maintain several individual states, it is best to combine such states into one.

For example, let us consider a button object. It has two states: pushed and released. When a button is being pushed, one can decide to have the following states: "pushed for less than one second", "pushed for at least one but less than two seconds", "pushed for at least two but less than three seconds", and etc. Unless it *serves* a particular purpose to track the amount of time that a button is being pushed, all of these states should be combined to a state simply called "pushed".

It should be noted that, *normally*, an object is at one and only one state in most cases.

Transitions

An object cannot be at no state. As a result, an object makes a transition from one state to the next in theoretically no time. A change of state is a transition.

A transition has several attributes. The "trigger" of a transition states what events external to the object can trigger a transition. The "guard condition" of a transition specifies a condition that must be true before a transition occurs (despite the availability of a trigger). The "action" of a transition specifies an action that can be taken when a transition occurs.

A special trigger is the "null trigger". A null trigger is always there. Any transition that does not specify a specific trigger is implicitly using a null trigger. A transition with a null trigger does not wait for any external event, it can take place as soon as the guard condition becomes true.

A special guard condition is "true" itself. Any transition that does not specify a guard condition is implicitly specifying "true" as the guard condition. A transition with "true" as guard condition takes place as soon as the specified trigger takes place.

A special action is the "null action", which does nothing. This is the default action when no specific action is specified in a transition.

Notation

A state is generally drawn as a elipse in a state diagram, whereas a transition is an arrow that indicates the direction of change. The trigger, guard condition and action are usually indicated as *trigger*[*guard condition*]/*action*.

A solid circle marks the "start state", while a solid circle in a hollow circle marks the "end state". For a particular object, it is at the start state immediately after the object starts to exist, and it is at the end state immediately before it is destroyed.

Examples

Figure 11.3 illustrates a state diagram that describes the general behavior of a checkbox. Note that there is no guard condition nor action (called "effect" in Poseidon). In this diagram, "click" is an external event generated when an end

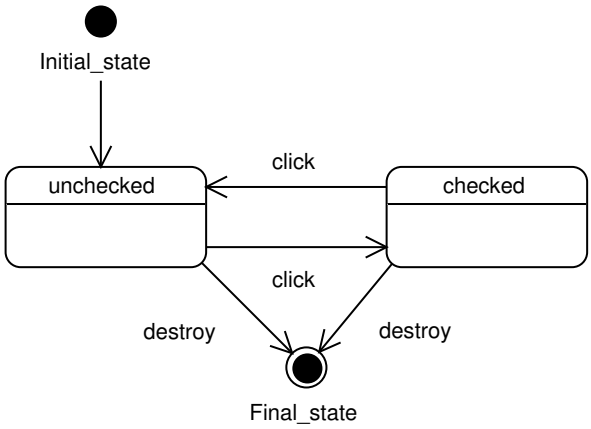


Figure 11.3: Simple state diagram to illustrate states of a checkbox.

user clicks on the checkbox. Interestingly, "destroy" is also an event. This is because a checkbox never knows when the containing window is going to destroy it.

Figure 11.4 is a more advanced version. In this diagram, the transitions between the "checked" and "unchecked" states are guarded by the condition "enabled". In other words, if the control is disabled, it cannot change state. Furthermore, there is an action "(callback)()" associated with the transitions between the "checked" and "unchecked" states.

11.3 But Which Representation?

No single representation is best for all cases. This section compares and contrast the representations based on a set of criteria.

11.3.1 Representation evaluation criteria

Novice friendliness

This criterion states how readily a representation can be understood by a novice.

Although programmers and test engineers are technical enough to understand practically any representation of logic, the same cannot be said for technical writers and the client.

A more novice friendly representation means that more parties can participate in the review of the system behavioral specifications. This, in returns, translates to better feedback at an earlier stage so that errors, confusions and misunderstanding can be rectified as early as possible.

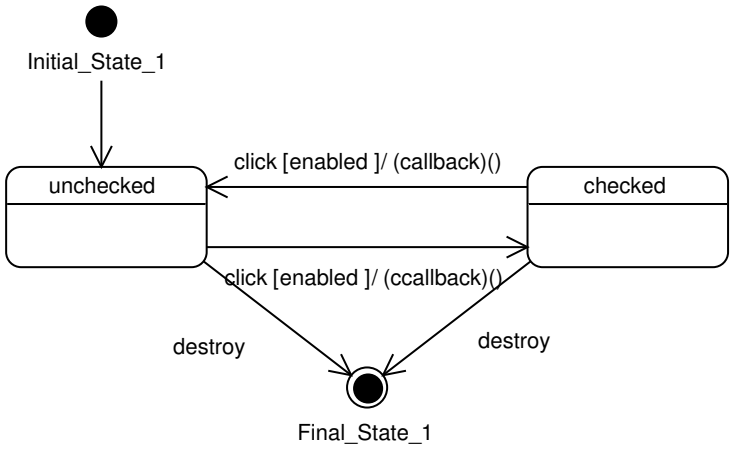


Figure 11.4: More advanced state diagram to illustrate states of a checkbox.

Rule checking tools

Most representations of logic are intended for people. As a result, most do not have tools to check and enforce rules.

Having tools that can check for rule violations can help to detect confusing and wrong logic from the beginning. This is an important criterion.

Inherently structured logic

Representations of logic can be divided into two main categories. One is structured, while the other one is unstructured.

An inherently structured representation means a logical construct must be completely contained in another. A representation that is not inherently structured is one that has no such requirement, or one that does not have logical constructs that can include other logical constructs

Structure has a lot of importance when logic is specified in great details.

Supports hierarchies

A representation that can structure components hierarchically is like a map that supports zooming at different levels.

Note that this is not the same property as "Inherently structured logic", although some representations can do both. A hierarchy means that an individual item at one level can be expanded to a full diagram or textual representation of logic.

Although it is theoretically possible to represent the logic of an entire system in one single diagram or as one text description, it is not practical. A person

can only track a relatively small number of items at any given time due to the limitations of short-term memory. As a result, the practical complexity of each diagram or text description must be limited.

Given this limitation, the only way to represent the logic of a complex system is to use a hierarchy of diagrams and/or text descriptions.

Allowance of "taboo" constructs

Certain constructs are considered "taboo" in modeling and programming because they lead to many problems (testing, debugging, maintenance). One of these constructs is "go to", which allows a program to jump from a point to virtually any other point in the same level (hierarchy) of representation.

Although the allowance of such constructs does not necessarily mean analysts will use them, it is generally not a good idea to even allow such constructs be written in the first place.

Conciseness

Conciseness is density. How many pages do we need to represent the same amount of logic?

A more concise logic representation requires fewer pages to represent the logic.

This criterion is generally speaking a low priority one because the number of pages is hardly an issue when compared to the other criteria.

11.3.2 Representation evaluation

This section evaluates the representations we have discussed in this chapter. However, instead of listing the representations, this subsection is organized by application. This way, you can first identify the application, then you can choose the appropriate representation.

Object oriented *state* logic

Use state diagrams. The states in a state diagram are possible states of an object (of a particular type). As a result, the diagram is inherently object oriented.

When a transition has a trigger, it specifies how an object *reacts* to external events. Note that an event can be a request of another object that is inside the same information system.

State diagrams are, generally speaking, easy to understand because of the use of arrows. However, state diagrams can become difficult to understand when more advanced concepts, such as concurrent states and composite states, are utilized. One of the most important factors to make a state diagram easy to follow is to use proper captions for the triggers, guard conditions and actions (effects).

State diagrams have simplistic rules to enforce. For example, most tools make sure a transition has an originating state and a destination state. However, most tools cannot handle logic errors, especially in the guard condition.

There is no inherently structured constructs in a state diagram.

State diagrams allow hierarchical layering. In other words, a "state" in one state diagram can expand to a state diagram. This means it is possible to organize the diagrams so that there are only about 7 to 10 states in each diagram. This is illustrated in figure 11.5.

There is little or no "taboo" constructs in state diagrams.

State diagrams are diagrams, therefore they are generally speaking not very concise.

Multiple independent conditions, multiple actions

If you have multiple independent conditions that will combine to determine what actions to take, decision tables may be a suitable format for capturing that logic.

Decision tables are easy to understand if they are properly labeled. The conditions and actions should be properly labeled.

There is not many rules to enforce, except that all possible combinations be enumerated. This can be generated by a macro or a small program.

Decision tables do not have inherently structured constructs.

A huge decision table can be broken down into smaller ones. This process can be repeated until the size of each table is manageable.

There is no "taboo" constructs to speak of.

Decision tables are both concise and easy to incorporate into comments of program code. This is an plus for programmers because now it is possible to integrate parts of the system specification document into program code.

Interleaved decisions and actions with no loops

If you need to express sequences of interleaved actions and decisions, *and that decisions do not cross paths*, decision trees are suitable representations.

Decision trees are easy to interpret because of the use of arrows. The conditions for decisions should be labeled clearly.

There are very few rules to enforce. One is that branches of decisions cannot merge.

Each decision is, inherently, a decision tree by itself. As a result, decision trees do have inherent structures.

The inherent structure of a decision tree lends a decision tree to hierarchical representation. Each "end node" of a tree can be expanded to its own decision tree.

There is no "taboo" constructs in decision trees.

Decision trees are concise when compared to other diagrams.

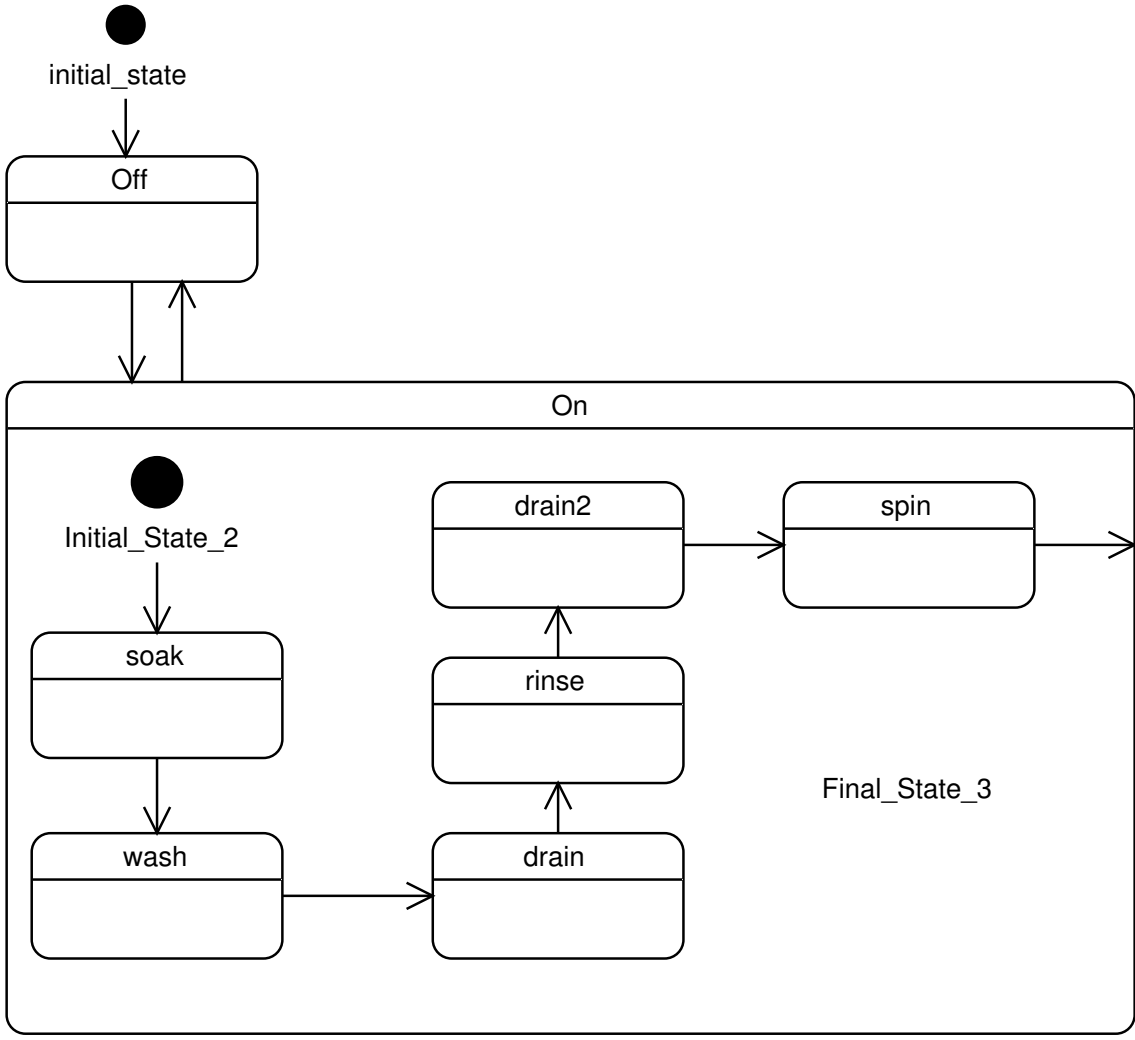


Figure 11.5: Washing machine example to illustrate hierarchical state diagrams.

General Logic

When you need to specify general logic (that is not suitable for the previous categories), you can choose between a flowchart and pseudocode.

Pseudocode may seem more difficult to understand at the beginning, but it becomes easy after some practice. Because of the lack of arrows, pseudocode requires the reader be familiar with control constructs. Flowcharts, by comparison, are easy to understand because a beginning only needs to follow the arrows.

Some flowcharting tools check certain rules (like only one outgoing arrow from a process). However, *most* drawing applications do not check rules for flowcharting. Pseudocoding is also "ruleless" when done with a regular editor. However, it is fairly easy to define macros in most word processors so that pseudocode is properly formatted when it is created.

Flowcharts do not have inherently structured constructs. On the other hand, all construct constructs in pseudocode are inherently structured. This is a big plus for pseudocode because structured constructs result in fewer mistakes.

Both flowcharts and pseudocode can be represented in hierarchical layers. A process in a flowchart can be represented by its own flowchart, and a statement can be represented by a piece of pseudocode.

Because of the lack of structure, it is *easy* to express tabooed logic in a flow chart. In fact, it is easy to draw flowcharts that are difficult to implement in a modern programming language. By comparison, pseudocode is close enough to most modern structured programming languages that tabooed logic can be outlawed simply by stating "goto is not allowed".

Pseudocode is much more concise than flowcharts because it is textual, not graphical. This is a big plus because it is relatively easy to put multiple layers of a hierarchy in view concurrently. A single flowchart takes up a bit of space, which makes it impractical to open multiple flowcharts (as different layers of a hierarchy). This makes it more difficult for an analyst (or other participants of an IT project) to quickly drill into details from an overview, or vice versa.

Chapter 12

Activity Diagram

This chapter is entirely dedicated to activity diagrams in the UML. Activity diagrams (ADs) are, perhaps, the most difficult type of diagrams to master in the UML.

12.1 Its purposes

Activity diagrams is yet another method to express behavioral logic. In many ways, it resembles flowcharts. However, ADs can be far more complex than flowcharts. ADs can express many modern concepts, such as concurrency and information flow.

12.2 Simple ADs

Let's begin with some simpler ADs. All ADs start with an "Initial State" and end with a "Final State". We'll discuss the differences between ADs and state diagrams later in this chapter. To make our terms less confusing, let us use the word "step" instead of "state" in ADs.

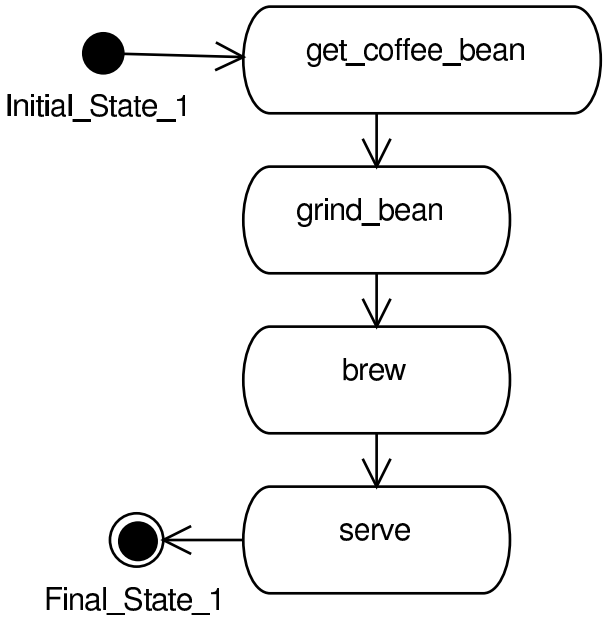
Steps in an AD are connected by transitions. As a result, the simplest AD is as follows:

Of course, this AD is fairly useless!

12.3 Steps

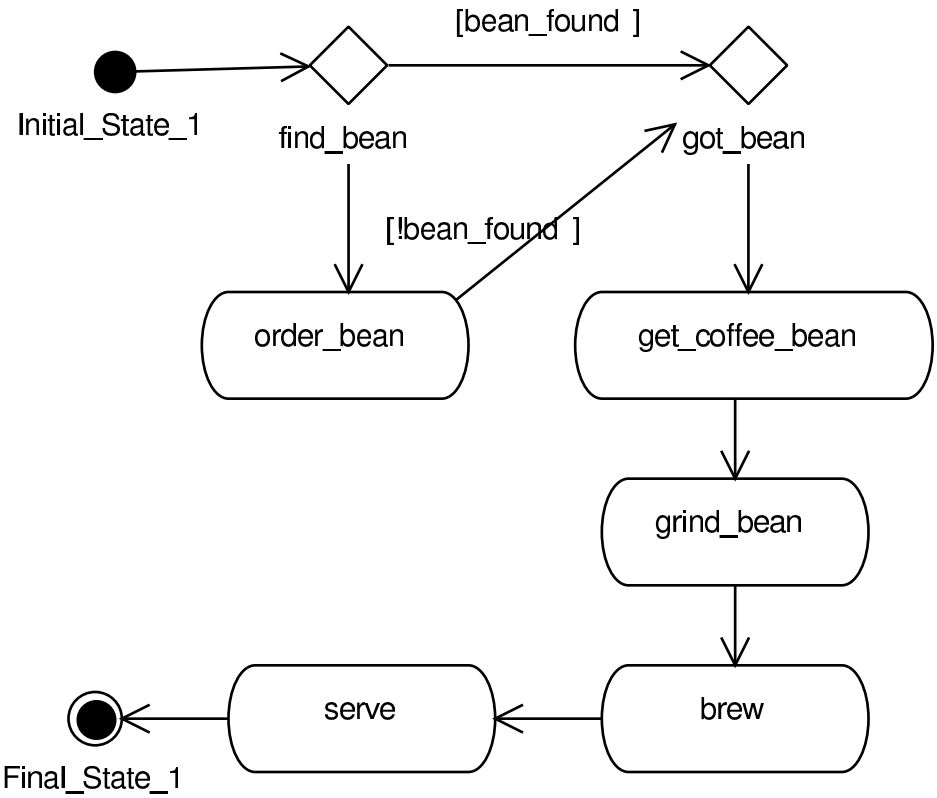
In many ways, a "step" in an AD is very similar to a "state" in a state diagram. However, the intention of a "step" is to express an activity. As a result, it is common to specify an "entry-action" for a step in an AD.

The following is a simple sequence of steps:



12.4 Decisions

Although one can use guard conditions in a transition to specify decisions, ADs have an explicit construct to do this. The "Branch" construct explicitly express decisions. In our example, we can do the following:

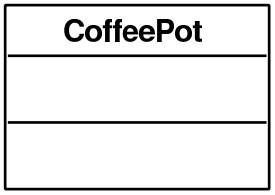


Note that the first diamond "find_bean" represents a decision. All its outgoing transitions must have a guard condition. The second diamond "got_bean" represents a path merge. Strictly speaking, the path merge step is not required, but many people use one to represent the merging of two *alternative* paths.

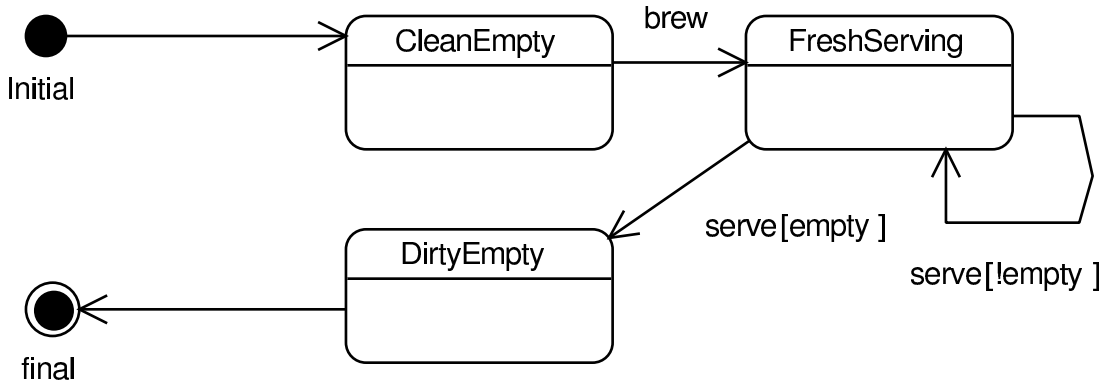
12.5 Data Flow

This is where things get interesting.

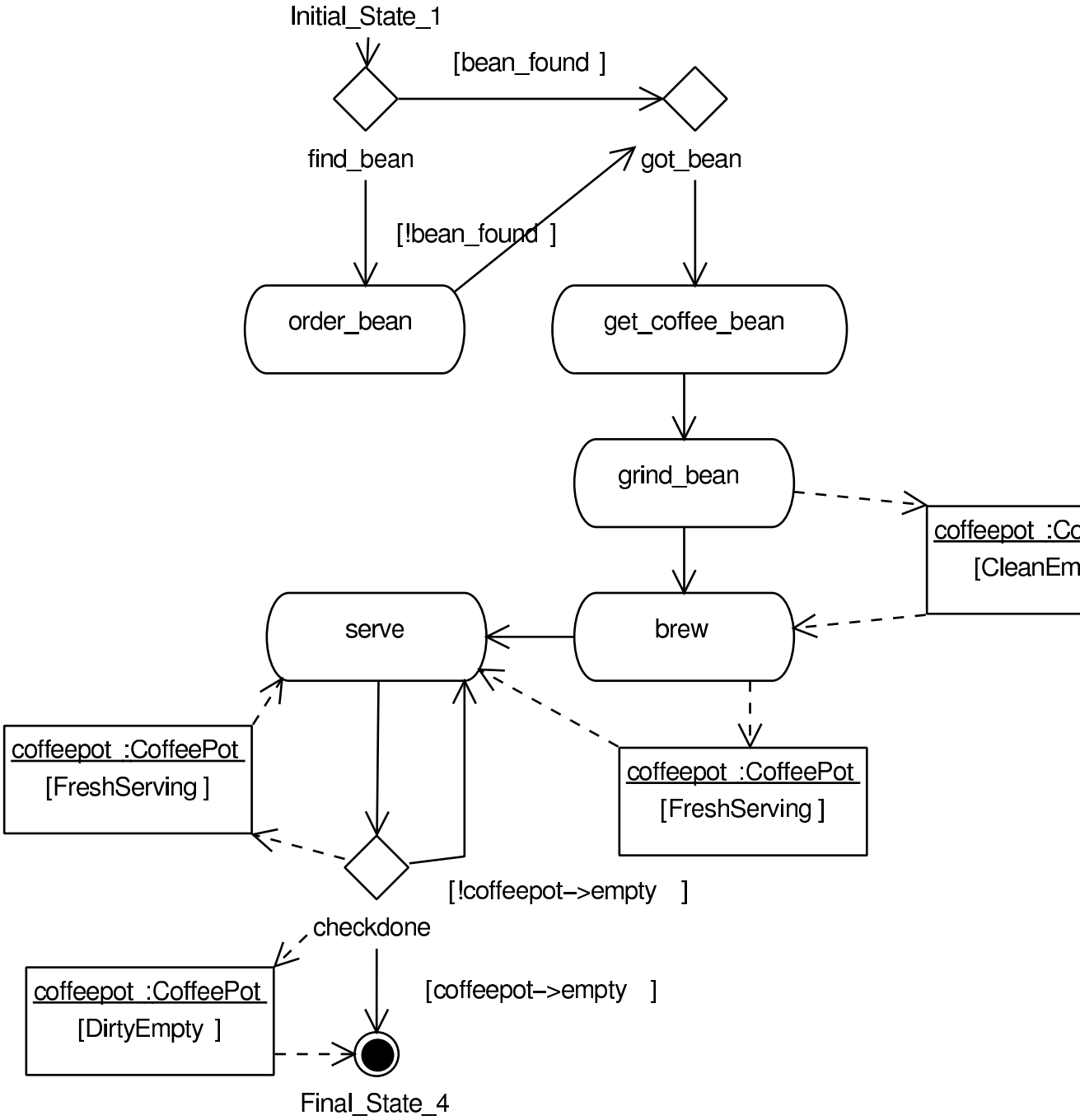
First, we need to recall class diagrams and state diagrams. Each object needs to belong to a class. As a result, we need to first construct a class (from which to create an object).



Next, we create a state diagram for objects of this class, as follows:



Now, we enhance our previous activity diagram with object flow:



In this diagram, "coffeepot" (all lowercase) is the name of the object that has a type of "CoffeePot". We give it a name so we know which coffee pot we are talking about. The dotted arrows indicate data flow rather than the usual

control flow/transition.

According to this diagram, a coffee pot that is clean and empty is the output of step "grind_bean". Of course, we really should have another step to get a clean pot, but let's just say that that is merged to the step to grind bean.

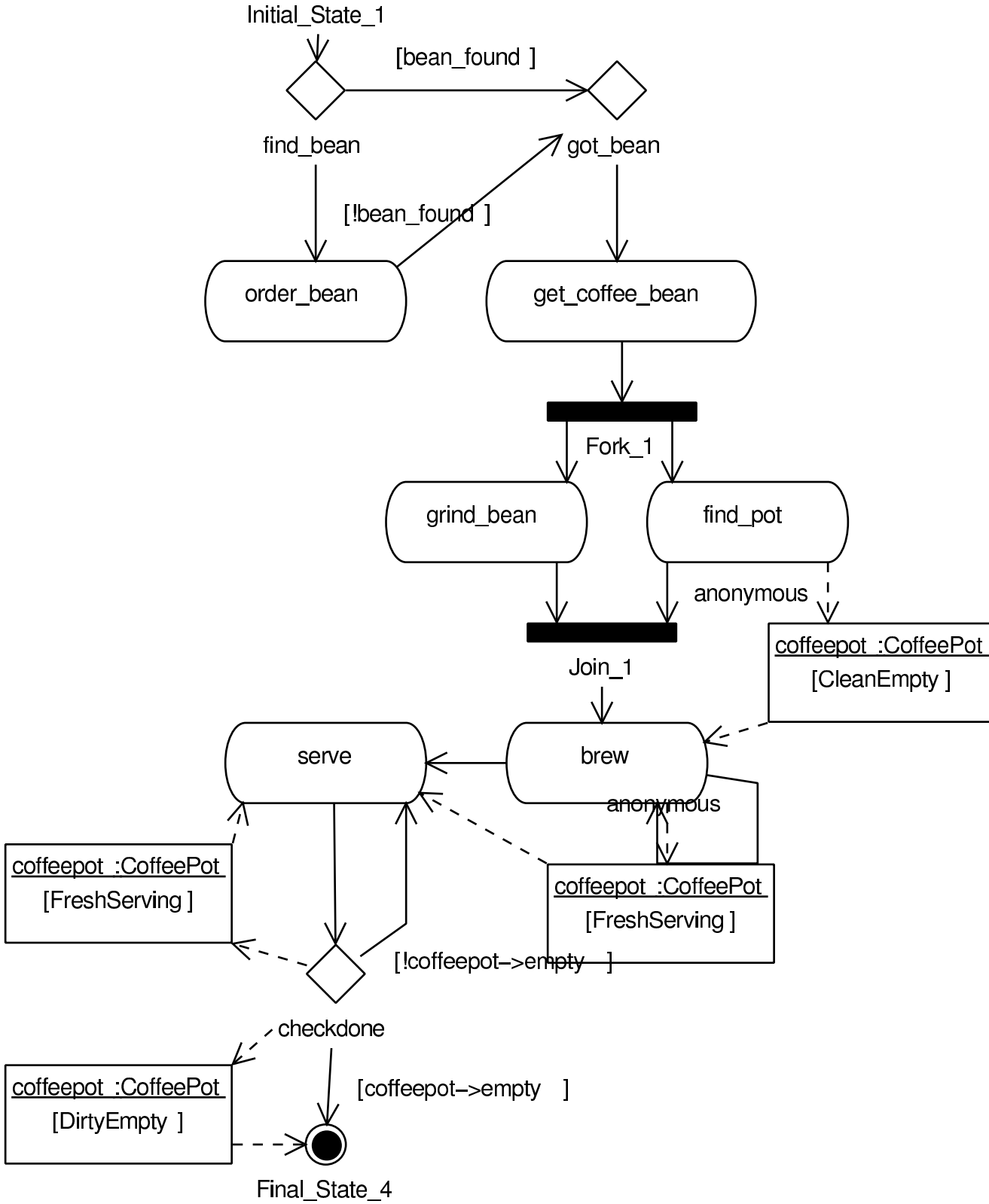
The important part, however, is that the "clean and empty" coffee pot is the input to the step "brew". To make this AD correct, we really need an object to represent a batch of beans.

The output of step "brew" is a pot of coffee that is fresh and ready to be served. This becomes the input of the step "serve". After each serving, we check to see if the pot is empty. If not, the pot is fed back to step "serve". Otherwise, the "empty and dirty" pot is fed to the final state, and the activity concludes.

12.6 Concurrency

ADs can express concurrency. This is an important feature for modern information systems. Languages like Java are thread aware. This means that the UML is capable of specifying system behavior at a level that is closer to the actual code.

In our example, we can parallelize the grinding of bean and the search for a clean pot. This is expressed as follows:


Initial_State_1
[bean_found]
find_bean
got_bean
[!bean_found]
order_bean
get_coffee_bean
Fork_1
grind_bean
find_pot
anonymous
coffeepot :CoffeePot
[CleanEmpty]
Join_1
serve
brew
anonymous
coffeepot :CoffeePot
[FreshServing]
coffeepot :CoffeePot
[FreshServing]
[!coffeepot->empty]
checkdone
coffeepot :CoffeePot
[DirtyEmpty]
[coffeepot->empty]
Final_State_4

Note that solid horizontal bar is used to split an execution thread into two *concurrent* threads. It also combines two threads into one. It is easy to use these symbols to construct an AD that makes no sense. Be very careful when you specify concurrency!

Chapter 13

Data and Organization

In the previous chapter, we discussed various means to document the behavior of an information system. However, in any realistic information system, just having fragments of specification of behavior is far from sufficient to document the system.

This chapter discusses the other two key ingredients to document an information system.

13.1 Data

The diagrams from the previous chapter do not include any discussion of the explicity representation of data. Data is an essential ingredient in the specifications of an information system. Afterall, information is stored as data.

13.1.1 Variables

In the representations of behavior, most will involve the use of variables. For example, in the pseudocode of the log in process, the following may be specified:

```
repeat
  prompt for userid and password
  if userid exists and failed attempts is exceeded then
    print error message
  else
    if userid and password do not match then
      print error message
      if userid exists then
        add one to the number of failed attempts
      end if
    end if
  end if
until userid and password match or failed attempts exceed limit
```

Let us now examine the processes that we need to participate to locate the variables.

Identify Variables

Variables are easy to identify from pseudocode (or the caption of processes of a flowchart). Variables are usually nouns.

In our example, the variables are as follows:

- userid
- password
- number of failed attempts

Identify Scopes

The scope of a variable defines where a variable is visible. In our example, the variable "userid" needs a scope that extends beyond the mentioned code. This is because the "userid" is needed later to retrieve information specific to the user.

On the other hand, the variable "password" is not needed after the mentioned code. As a result, the scope of "password" may be limited to the mentioned code.

The scope of the variable "number of failed attempts", however, is a little more difficult to determine. This is because "number of failed attempts" is not exactly a variable!

Identify Lifespan

Each variable has a "lifespan". The lifespan of a variable determines when a variable starts to exist, and when it ceases to be. Note that the lifespan of a variable is not necessarily the same as its scope. Furthermore, the lifespan of a variable is determined by the behavior of the system as it performs operation, and not determined by the static description of behavior.

In other words, just because a variable is not visible/accessible, it doesn't mean that the variable is gone. A analogy is car being observed by a helicopter. Normally, the car is visible to the helicopter. However, as the car is passing through a tunnel, the helicopter loses visual contact temporarily. During this period of loss of visual contact, the car continues to exist. As the car emerges from the other end of the tunnel, it becomes visible again to the helicopter.

In our example, the scope and lifespan of the variables "userid" and "password" coincides. In other words, "password" only needs a lifespan that is limited to our code fragment, while "userid" needs a lifespan that is beyond the execution of the code fragment.

The lifespan of "number of failed login" is interesting because "number of failed login" is not exactly a variable.

13.1.2 Persistant Objects

In our previous example, "number of failed login" is not exactly a variable. This is because it is a property of a persistant object.

A persistant object is one that has a lifespan that is independent of scope.

In the case of our example, the persistant object is a record of a user. The "userid" variable is used to retrieve the full record pertaining to this user. This record is a persistant object because its existence starts with the registration of the user, and it ends with the deletion of the record (when the user decides to withdraw membership, for example). For all we know, the existence of a record like this can survive even generations of the information system.

A persistent object can have many properties. A user record has many properties, such as first name, last name, gender, street address, etc. The "number of failed attempts" is merely one of the many properties of the record of a user.

This is why "number of failed attempts" is not really a variable. It is a property of a persistant object. It does, however, uniquely identify one particular property of a user record.

A persistant object is also called a non-volatile object (not to be confused with the reserved word `volatile` in C and C++). Outside of object-oriented programming, a persistant object is most commonly known as a record in a database table.

13.1.3 Class Diagram

What is a class?

A class is a template from which objects are created. A class is like a cookie cutter, whereas an object is a cookie. A class is also like a definition you can look up in a dictionary, whereas an object is a real item described by the dictionary definition.

This concept is essential to object-oriented modeling. Not only does a class describe an object (from which it was created), but relationships among classes help to organize data in an information system.

In the UML, a class is represented by a rectangle with three compartments. The first one is the name of the class. Refer to figure 13.1 for an example.

Attribute

An attribute in a class specifies a "slot" to fill with actual data for createdobjects. For example, consider the class "professor". Tak is an object of the class "professor". The class "professor" has an attribute "last degree earned". This means every objects of the class "professor" must fill in this slot. For Tak (the object), this attribute has a value of "Ph.D.".

Attributes are also known as "fields" in more traditional programming vocabulary. An attribute is *somewhat* related to a column in a database table.

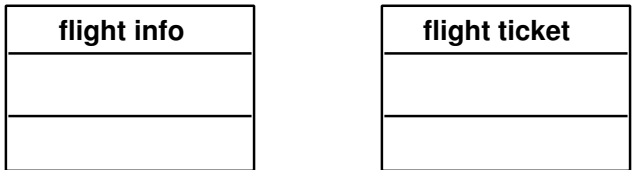


Figure 13.1: A plain class.

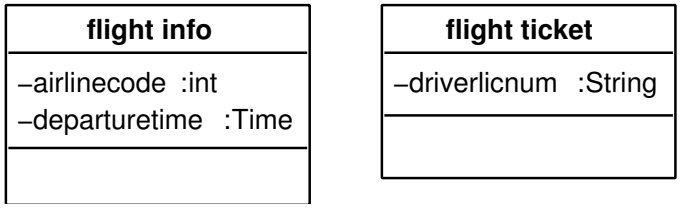


Figure 13.2: Classes with attributes.

However, an attribute can be far more complex than a column. This is because the "type" of an attribute (of a class) can be another class.

For example, the class "session" create all session objects. This class contains an attribute "lastActive" which has a type of class "dateTime". The class "dateTime", in return, has attributes such as "seconds", "minutes", "hours", etc.

Note that attributes can be marked "private", "protected" or "public". An attribute marked "public" means the attribute is accessible at all times, no matter which part of the system specification wants to access the attribute. We will discuss "private" and "protected" a little later.

Being able to use classes as types of attributes makes it possible to use a hierarchy to represent complex data items.

Figure 13.2 is an example of a class with some attributes.

Operations

A class may contain operations. An operation of a class is also known as a method in OOP.

For example, the class "session" may have an operation called "login". This means all objects created from the class "session" know how to perform the "login" operation.

Just like an attribute can be marked "private", "protected" and "public", an operation can be marked by the same three access rights. The access rights

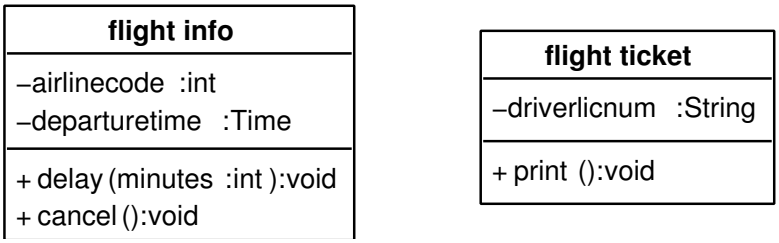


Figure 13.3: Classes with operations.

have the same meanings for operations.

If an attribute or an operation is marked "private", it means only operations *of the same class* has access to the marked attribute or operation.

Figure 13.3 is an example of classes with operations.

Generalization

A class can generalize another class. For example, the class "account" is a generalzation of "user account". This means all objects that are derived from "user account" automatically inherits attributes and operations from "account".

Although *all* attributes and operations are inherited from a general class, not everything is accessible. If an attribute or an operation is marked "private", a derived class no longer has access to it. If an attribute or operation is marked "protected", it means the attribute or operation is only accessible by the class (in which it is defined) or classes derived from this class.

To say a class "A" is a generalization of "B" does not mean that "B" has an attribute of type "A". Technically, when "A" generalizes "B", all attributes and operations of "A" are *automatically* inherited by "B". All objects created by "B" *automatically* know how to perform operations specified by "A". To use an analogy, saying "vertebrate" generalizes "mammal" means all animals that are "mammal" have the attributes and operations of "vertebrate": having a spine and the ability to transmit signals through the spine.

On the other hand, if "B" has a attribute "X" that is of type "A", it merely means that attribute "X" is of type "A". For example, the "session" class has an attribute "lastActive" that is of type "dateTime".

What is the difference? For one thing, if class "B" is derived from class "A" ("A" is a general class of "B"), all the attributes marked public *and protected* are accessible in class "B". If class "B" contains an attribute "X" of type "A", "B" has no access to attributes and operations defined in "A" marked "protected" or "private".

The other difference is that if "B" is derived from "A", all the attributes of type "A" can, in fact, refer to an object of type "B". An analogy is that "meat"

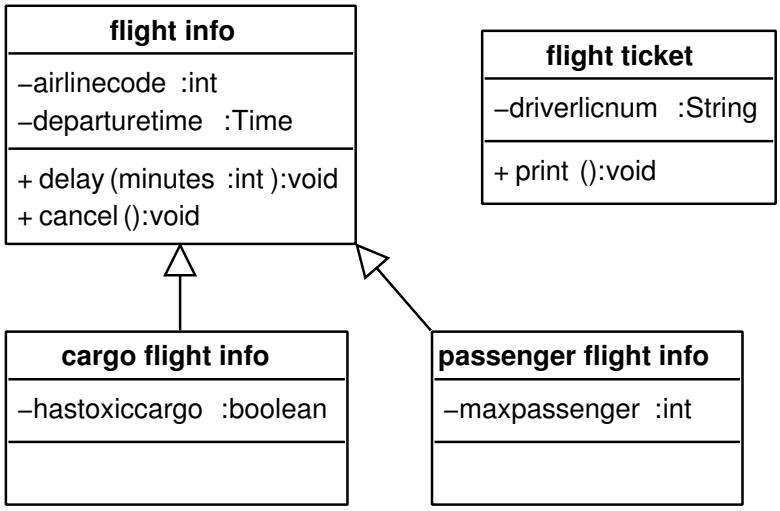


Figure 13.4: Generalization of classes.

is a generalization of "pork". This means if a recipe calls for "a pound of meat", you can use a pound of pork.

In figure 13.4, both "cargo flight info" and "passenger flight info" are generalized by "flight info".

Association

A class can be associated with another via the "association" relationship. "Association", in this case, really should have been called "reference". This is because the relationship is directed, and it really means the ability to refer to objects of another type.

For example, if class "A" has a reference to class "B", this means an object of class "A" refers to *a fixed number* of object of class "B".

This relationship is useful because in some cases, it is wasteful or just wrong to use attributes to "include" something else. For example, an object of class "flight ticket" needs to somehow relate to the flight information such as departure time, gate and etc. However, it is wasteful to put in such attributes for each and every "flight ticket" object. Afterall, all flight tickets of the same flight have the same departure time, gate and etc. As a result, it make more sense to have each "flight ticket" object to *refer to* a "flight" object. The "flight" object contains the departure time, gate and etc.

In figure 13.5, a "flight ticket" object has a reference to a "passenger flight info" object. The use of arrow is important here because it indicates that while an object of class "flight ticket" has a reference to an object of class "passenger

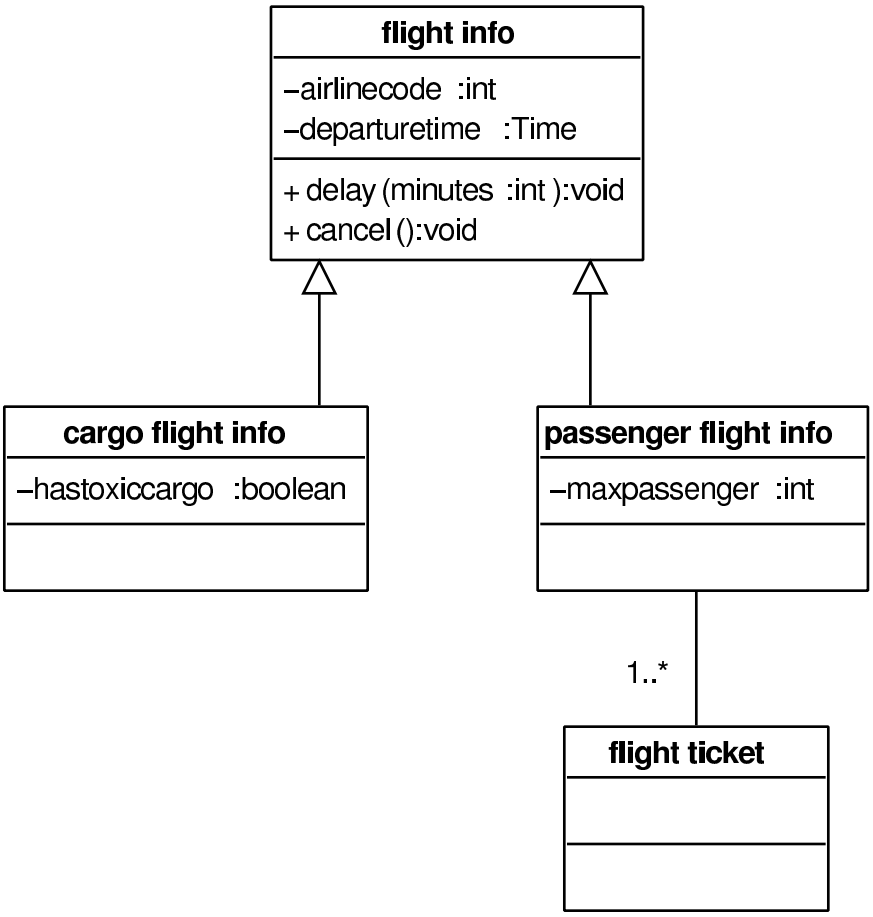


Figure 13.5: Association of classes.

flight info", the inverse is not true. In other words, an object of "passenger flight info" does not have any reference to objects of class "flight ticket".

Note the multiplicity in this diagram. A "passenger flight info" object has multiple objects of class "flight ticket" referring to it.

Aggregation

Using our previous example, it is helpful to know, given a particular flight, the tickets already issued. This may be used when a flight is delayed so that all passengers can be notified.

An aggregation relationship is kind of like an association (reference) relationship, except that there is no limit of number of objects. In our example, we can define an aggregation relationship from the "flight" class to the "flight ticket" class. This allows an object of "flight" class refer to *any number* of

"flight ticket" objects.

When a flight is canceled, we will eventually destroy the underlying object of "flight" type. What will happen to the "flight ticket" objects? Using the aggregation relationship, the "flight ticket" objects continue to exist even after the related "flight" object is deleted. If the application needs to delete all "flight ticket" objects when the related "flight" ticket is deleted, we need to read the next section.

Figure 13.6 is an example of aggregation of classes. An object of class "cargo flight into" has references to objects of class "package tracking info". This is proper because objects of class "package tracking info" must continue to exist even after a "cargo flight info" object is deleted.

Composition

If a class "A" has a composition relationship to another class "B", it means an object of type "A" treats a number of objects of type "B" as *components*.

In our example, we should consider changing from "aggregation" to "composition". This is because when a "flight" object is deleted, we also want the related "flight ticket" objects be deleted.

Figure 13.7 is an example with composition. In this case, the black diamond specifies that objects of class "flight ticket" are considered components of an object of class "passenger flight info". The lack of arrow indicates the association is bidirectional. This means an object of class "flight ticket" always has a reference to an object of class "passenger flight info", while an object of class "passenger flight info" has the reference to at least one object of class "flight ticket".

13.2 Examples!

13.2.1 Grading System

Let us examine an information system that aides instructors in tracking grades. At the top level, we have the "college" class. A "college" class object represents an individual college, such as ARC. At the next level, we have the "course" class. A "course" class object represents a particular course offered by a college. Whether there should be an intermediate class "department" depends on whether the system needs to represent departments or not.

At this point, our class diagram has two classes. It is reasonable that a "college" class object has references to multiple "course" class objects because this is helpful when a catalog needs to be printed. At the same time, it is also reasonable that each "course" object refer to the "college" object that represents the college offering the course.

It is clear that a college may offer any number of courses. The next question is whether the relationship should be an aggregate one or a composition one. In other words, when a college object is removed, should all the course objects associated with it be deleted as well? In a simple situation, the answer is

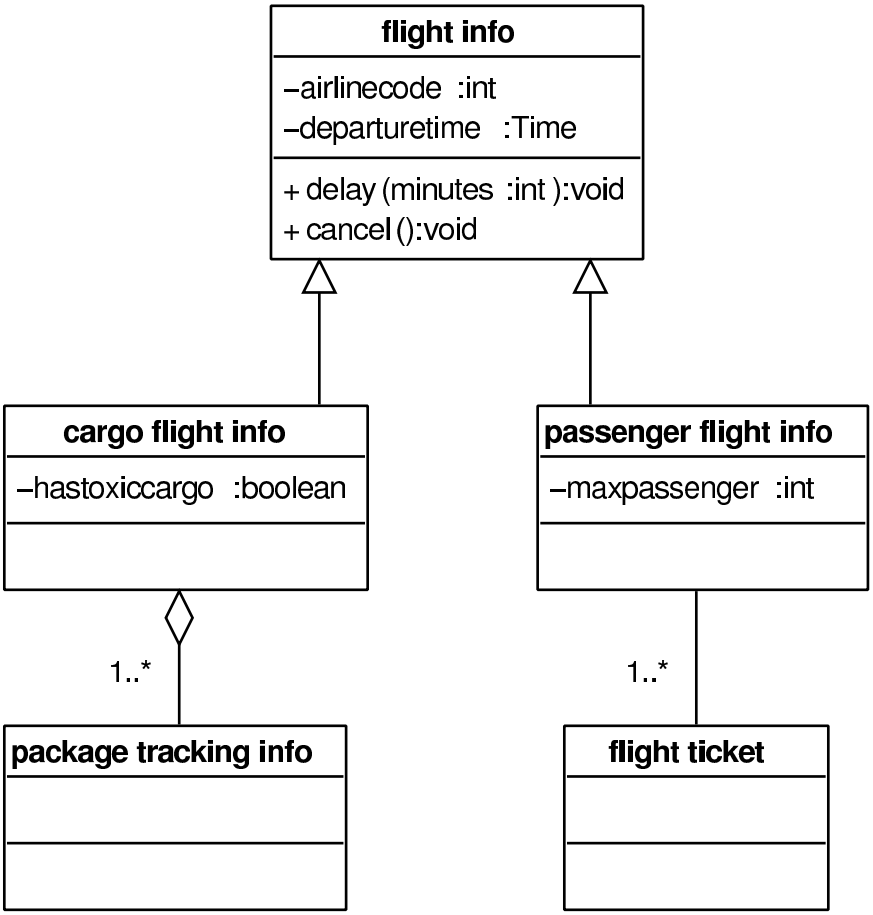


Figure 13.6: Class aggregation. An object of class "cargo flight info refers to at least one objects of class "package tracking info". The presense of no arrow means that the association is bidirectional.

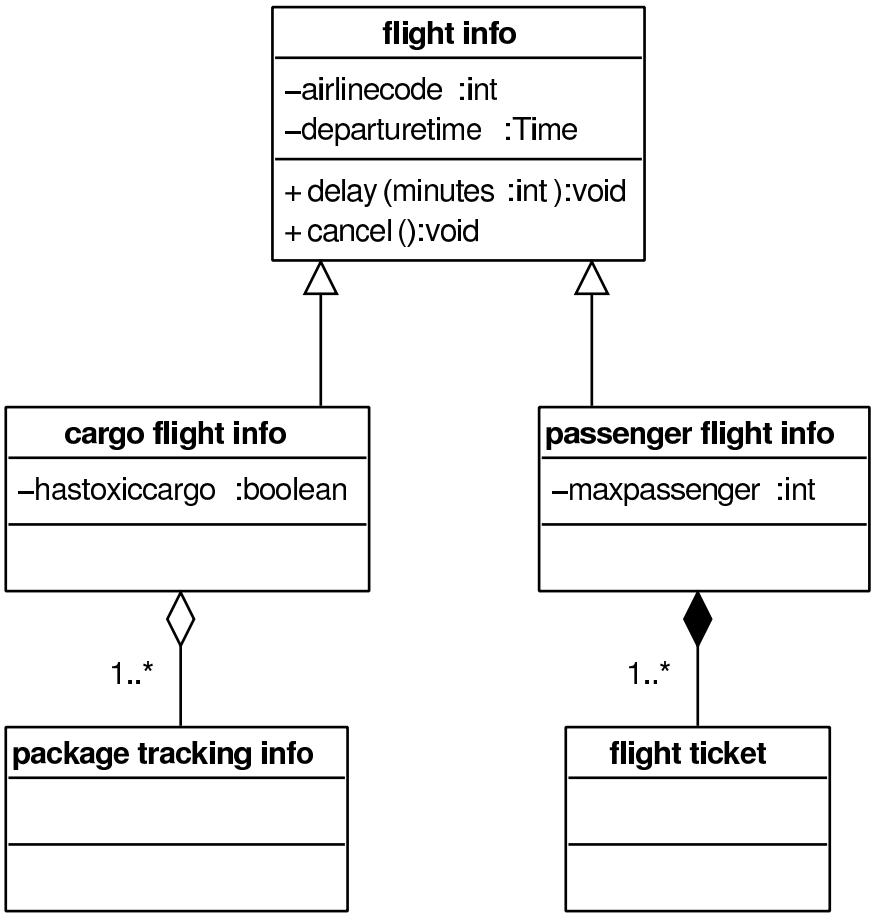


Figure 13.7: Example of class composition.

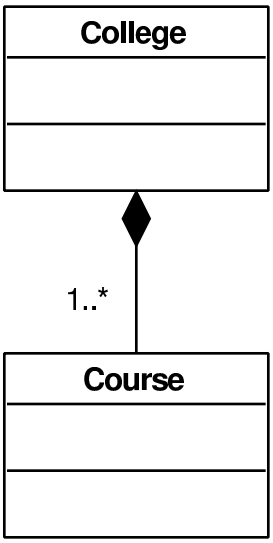


Figure 13.8: Grading system top level classes.

probably yes. As a result, the "composition" relation should be specified. The multiplicity should be specified to "1..*" because a college should offer at least one course.

The resulting diagram looks like figure 13.8.

Similarly, for each course in the catalog, multiple classes may be offered. The only difference, however, is that a course may be listed in a catalog without any offered classes in the schedule. As a result, the class "session" has a multiplicity of "*", which means it can be zero. The new diagram is figure 13.9.

At the next level, we may be tempted to add the class "Student". While each student does require a "Student" object for representation, we do not want to make the "Student" class associated with the "Session" class directly. This is because a student can take multiple classes. While it is okay to specify the multiplicity of both sides of an association to multiple (either "1..*" or "*"), it is generally considered impractical to do so.

As a result, it is better to define another class called "Enrollment". Each "Enrollment" object represents the enrollment of a particular student to a particular class (session). Obviously, if a student is expelled from the college, all the associated enrollment objects should be deleted as well. The same applies when a class is cancelled, all the associated "enrollment" objects should be removed. As a result, we have figure 13.10.

For a particular class, there can be any number of "Work" objects. Each "Work" object can represent an examination or a homework assignment. For each student enrolled in a class, there should be one "Work Result" object for each "Work" object. Another perspective is that each enrollment object should be associated to any number of "work result" objects, and each "work" object

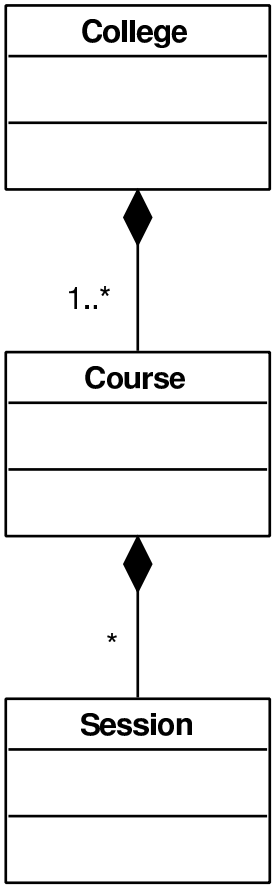


Figure 13.9: Grading system classes, including “Session”.

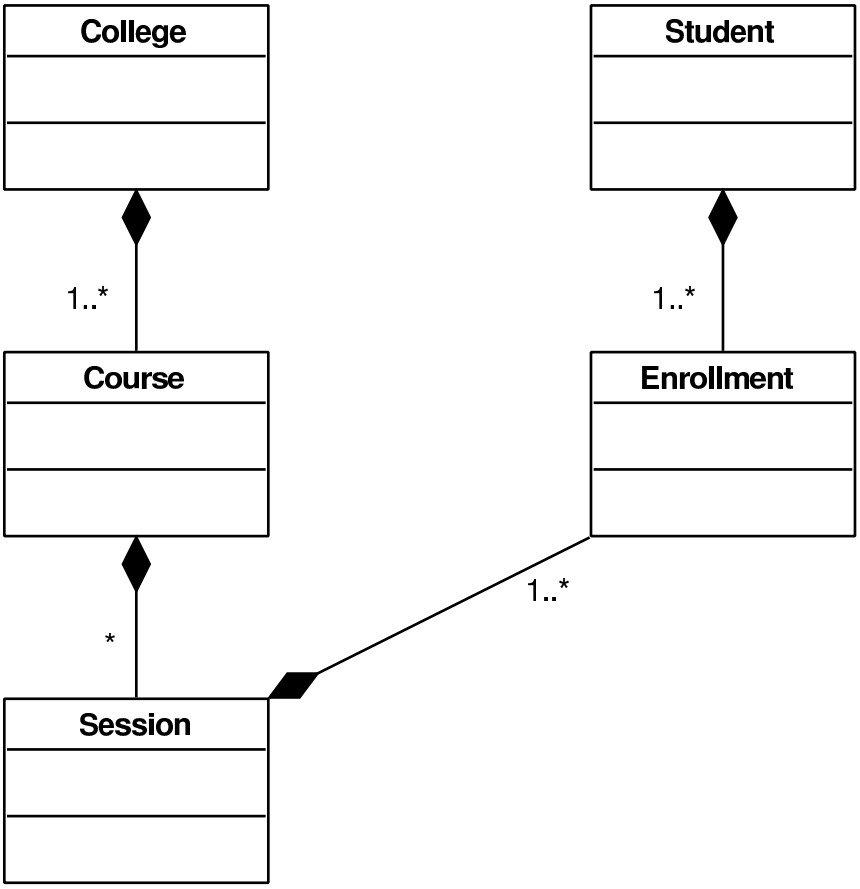


Figure 13.10: Grading system classes, including Student and Enrollment.

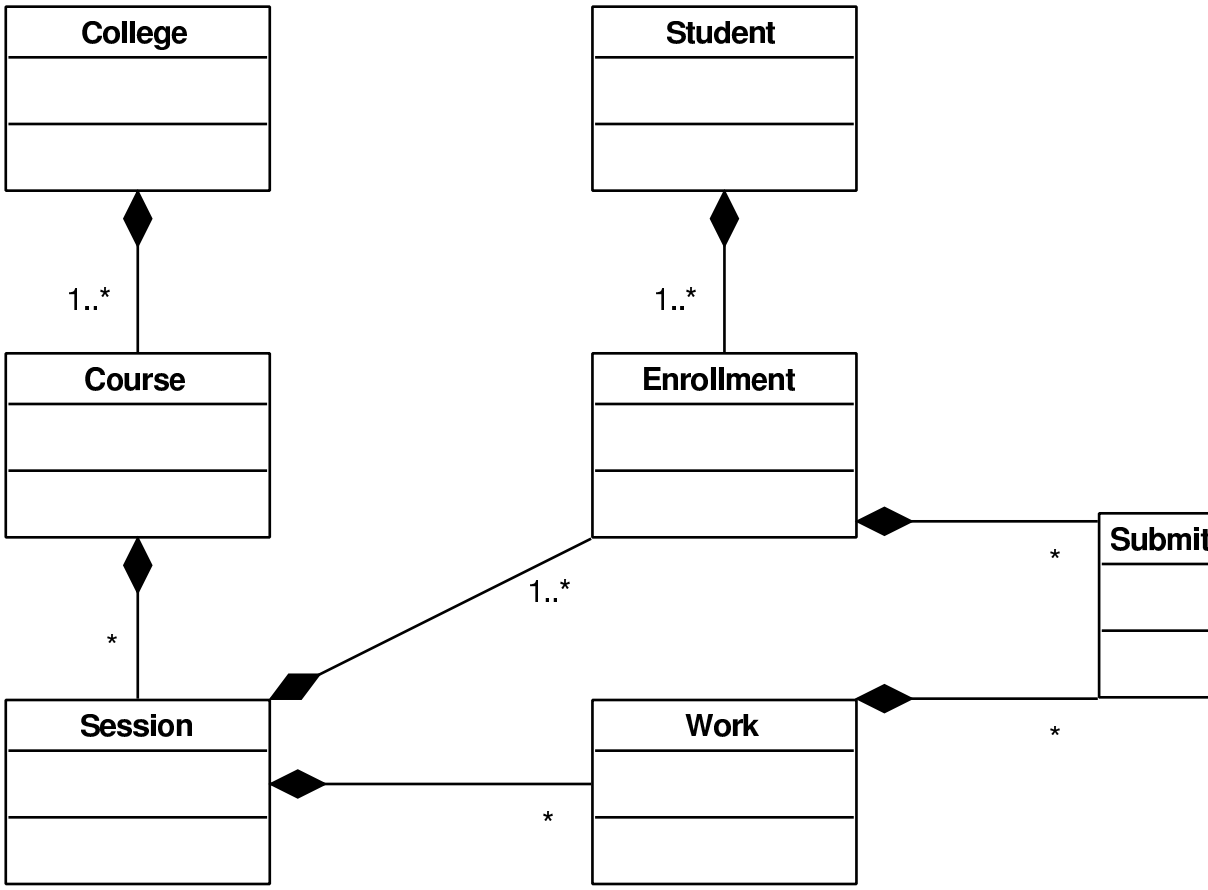


Figure 13.11: Grading system classes, including Work and "Work Result" classes.

should also be associated with any number of "work result" objects.

The result is captured in figure 13.11.

Chapter 14

An Example to Analyze: nRecipe

14.1 Purpose of this Chapter

The main purpose of this chapter is to let the class practice what they have learned so far. This is meant to be an in-class exercise done by groups. Each group should work on the problem independently (from other groups). At the end of the exercise, a presentation should be made about the specifications produced as the result of the analysis.

14.2 nRecipe Background

nRecipe is an imaginary network-based application for people to share recipes and to use online tools to help out in the preparation of food.

14.3 Requirements

nRecipe has the following actors:

- browser represents users who are not registered with nRecipe.
- user represents users who are registered with nRecipe.
- admin represents administrative users to maintain and update the system

14.3.1 The "Browser"

The "Browser" actor has the following use cases.

Find Recipe

The "Browser" actor can attempt to find a recipe. The search can be based on a combination of attributes of a recipe, including the ingredients, style, ethnicity, cost of materials, length of time to prepare, expertise required and etc.

This use case also includes the display of a recipe as a static document.

Leave Feedback

The "Browser" actor can also add feedback to a recipe. A feedback is not made public unless the owner of a recipe (must be a registered user) allows.

14.3.2 The "User"

The "User" actor can do everything that the "Browser" can, but with the addition of some more use cases.

The "User" actor has the following additional use cases.

Add/Edit Recipe

One of the best features of becoming a registered user is the ability to share recipes with other users. Once a recipe is added, it can also be edited. However, the deletion of a recipe must be done by an administrator.

Bookmark

A registered user can bookmark a recipe found in the "Find Recipe" use case. Note that "Bookmark" is not a use case. It is a feature of "Find Recipe" that is only enabled for registered users.

Step Guide

A registered user can use a feature called step guide. This feature in the display process of "Find Recipe" allows a registered use simple keystrokes to navigate steps of a recipe.

The main rationale of this feature is to allow a chef/cook easily access a recipe with minimum mouse clicks (none is best) and keystrokes.

14.4 Analysis (Exercise)

14.4.1 Sequence Diagrams

Use sequence diagrams to capture what you think the system should behave. I am leaving the details to your imagination and creativity. Since the log in, log out features are pretty much the same for all online systems, let's focus on the fun part of this system.

Particularly, focus on the use case "Find Recipe" because it includes quite a few features that are specialized to this application.

Draw 4 to 5 sequence diagrams to illustrate how a registered user may use the "Step Guide" feature in "Find Recipe". For simplicity, you can assume the user has already logged in and found a recipe or retrieved one from bookmarks.

14.4.2 Class Diagrams

Use class diagrams to capture how classes relate to each other. Let us start with the following classes:

- user: a "user" object represents a registered user.
- bookmark: a "bookmark" object represents a bookmarked recipe for a particular registered user.
- recipe: a "recipe" object represents a particular recipe.
- feedback: a "feedback" object represents a feedback message regarding a recipe.

Figure out and document how these classes relate to each other using the three types of relationships. Indicate the multiplicity of each end of a relationship according, and explain your rationale when it may not be obvious.

You can introduce additional classes if you feel they are needed.

14.4.3 Logic

Document the logic of the information system to generalize the sequences presented in the sequence diagrams. You can use any form of logic representation, including flowcharts, pseudocode, state diagrams, decision trees and decision tables.

Focus on the "Step Guide" feature in the "Find Recipe" use case. My suggestion is to use a state diagram to represent reactive logic, then choose one of pseudocode, flowchart, decision tree or decision table to specify actions to take during transitions. The state diagram should describe states and transitions of an object of the "Step Guide" GUI representation.

Part IV

Design

Chapter 15

What is the Design Phase?

In previous chapters, we discuss the analysis of an information system. The analysis process breaks a large and complex system into smaller and simpler components. This process to divide and conquer is repeated until the components are small and simple enough to be understood directly with no ambiguity.

At the end of the analysis process, we end up with a pile of specification documents. Such specification documents describe, complete and with no ambiguity, exactly how the modeled information system should behave. However, the specification does not suggest how to put the information system together, it merely indicates how the system behaves.

Let us borrow photography as an analogy. The systems requirement is analogous to the general "feel" of a scene. It describes what needs to be captured by a picture. For example, the "requirements" of a picture may be "vastness of the ocean and the sky, belittling human existance."

The systems specification, on the other hand, is the resuit of the analysis of the requirements in order to figure out what basic elements should be included. In the photography analogy, the "specification" may suggest "include the sky, ocean with no boundary; include artifacts of mankind that is aged; the artifacts should appear small."

The design of an information system is the synthesis and structuring of elements discovered in the analysis process. In the photography analogy, the "design" is the process to choose a time, an angle, a particular lens (with a particular angle of view) in order to include all the elements from the analysis process. An example of the result may be "take the picture at dusk with distant cloud, wait for an aged fishing boat to pass by, use a wide angle with one third beach and two thirds sky, include foot prints leading to the waterfront, place camera as close to the ground as possible."

In other words, the design process is mostly about deciding how to organize elements discovered in the analysis process for efficiency, ease of maintenance, flexibility and cost-effectiveness.

Chapter 16

Object-Oriented Design

Object-orientation design is becoming increasing important because of the use of object-oriented programming languages. C++, Java and Visual Basic are all considered object-oriented. Object-orientation is also a useful way to look at an information system because it "naturally" defines boundaries of objects, making it easier to make good design decisions.

16.1 Boundary, Control and Data

An information system, in OOD, is usually divided into three layers. The three layers, "boundary", "control" and "data", are explained in following subsections.

16.1.1 Boundary

The boundary layer consists of a group of boundary classes. Each boundary class describes the interface of the information system with respect to a use case and an actor of the use case.

In figure 16.1, each association can potentially be represented by a unique boundary class. However, multiple actors of the same use case may share the same boundary class. In this example, the "faculty" actor may share the same boundary class with the "reader" actor with respect to the "Grade Work" use case.

A boundary class represents the data and interface logic of a use case from the perspective of one or more actors. Be careful not to include any logic that is beyond display and data entry. In our example, the boundary class for "Grade Work" may be similar to the one displayed in figure 16.2.

Note that the "update..." methods are only responsible to redisplay a GUI element on the screen. It should defer to another class to figure out what to display. Likewise, the "select..." methods are only responsible to intercept the change of selection of a list box or the depression of a button as a trigger to

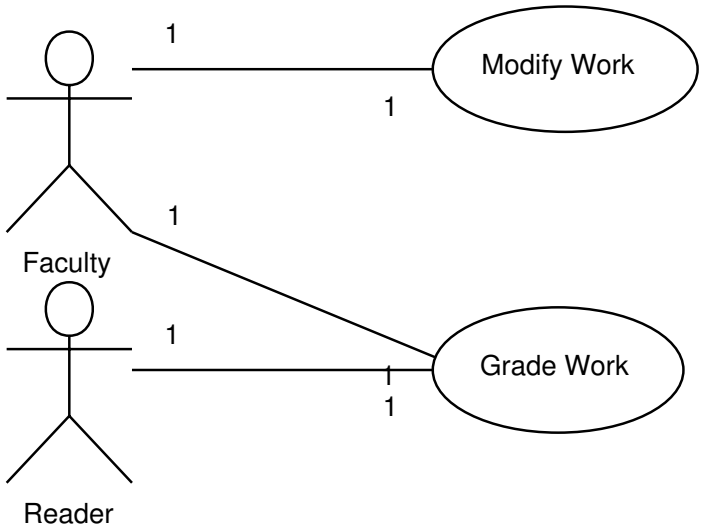


Figure 16.1: Example for the discussion of boundary classes.

change something on the screen. The boundary class may, then, specify the logic to update all the dependent GUI elements. For example, upon the selection of a different course, the boundary class may update the list of sections automatically and blank out all other elements until a section is selected.

Do *not* specify any logic that is beyond the logic related to the display and data entry aspects. For example, in the "Grade_Work_Boundary" class, do not specify the logic to query a database to retrieve a list of courses that can be graded by a particular grader. By the same token, do not specify any logic that assigns letter grade according to a numeric score.

In general, an object of a boundary class represents a particular active session when the system is in use.

16.1.2 Control

A use case has exactly one control class. The control class of a use case specifies the logic for that use case. It also contains attributes that are necessary for the logic.

Generally speaking, an object of a control class is allocated to a particular active session when the system is in use.

Figure 16.3 illustrates a class diagram that includes both the boundary class and control class in our "Grade_Work" use case.

Note that the "Grade_Work_Control" class needs to maintain the graderID, courseID and etc. in order to query a database. As a result, it is not necessary

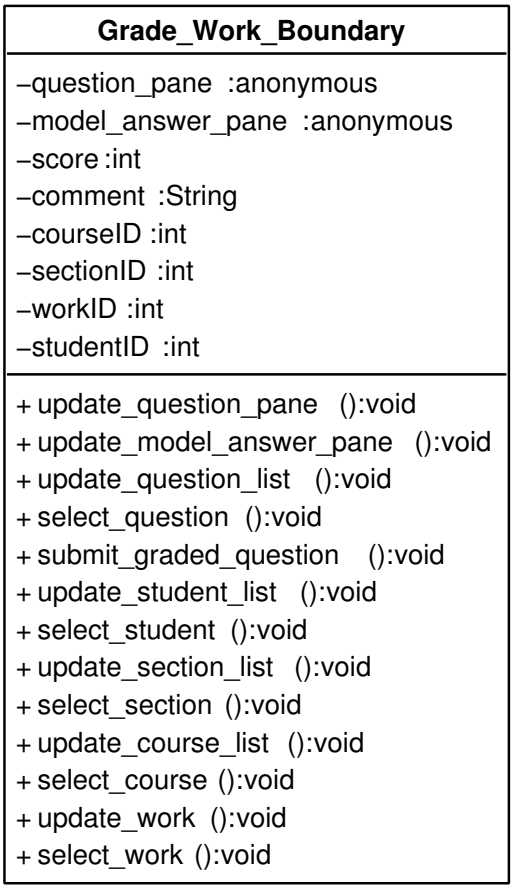


Figure 16.2: An example of a boundary class.

for the "Grade_Work_Boundary" class to maintain such information.

Note that the "find..." methods are used to return information. For example, the "update_question_list" method of a "Grade_Work_Boundary" object calls the "find_graded_questions" and "find_ungraded_questions" methods of a corresponding "Grade_Work_Control" object.

Similarly, the "set..." methods are used to indicate changes. When a grader selects a new question, for example, the "select_question" method of a "Grade_Work_Boundary" object calls the "set_question" method of a corresponding "Grade_Work_Control" object.

This separation of methods may seem unnecessary and cumbersome. However, it separates the logic of display/data entry from the "business" logic. As a result, a change of "business logic" may not need any update of the boundary class at all.

For example, in version 1.3 of the system, the "business" logic may allow a grader submit a score for a question without any comment. After numerous complaints from students, in version 1.5, the system requires a non-empty comment field for a score submission. This change only necessitates changes to the "Grade_Work_Control" class but not the "Grade_Work_Boundary" class.

16.1.3 Data

Each data class is much like a table in a database in a more traditional systems model.

Objects of data classes are often non-volatile. On the other hand, objects of control classes and boundary classes are usually volatile. This means that objects of data classes persist even across system power cycling.

The derivation of data classes from control classes is somewhat of a "black art of design". We'll explore this in the next chapter.

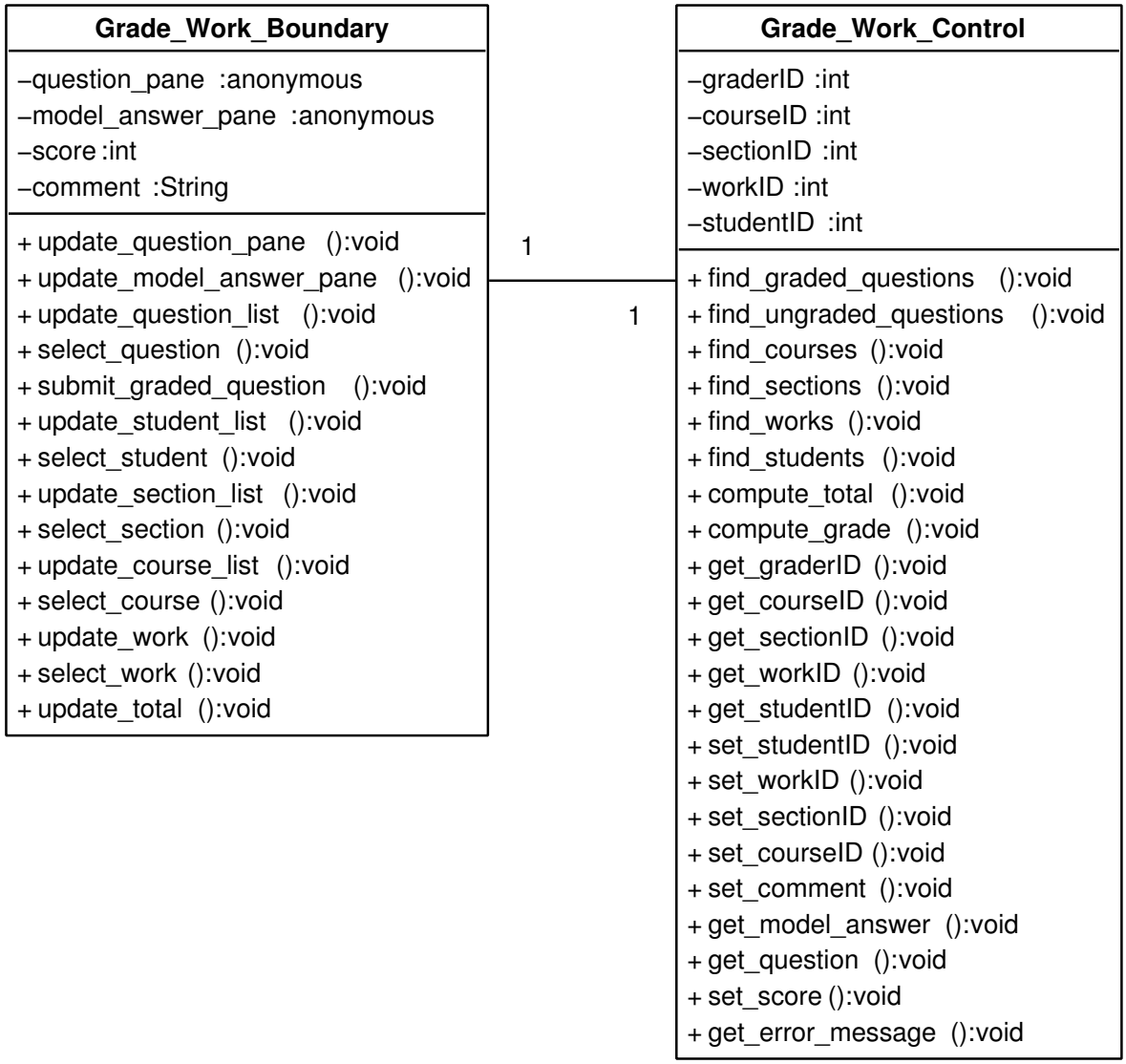


Figure 16.3: Class diagram showing a boundary class with a corresponding control class.

Chapter 17

Database/Class Design

This chapter is a shallow treatment of database design. If you are interested in database design, you need to take a few additional classes.

17.1 Terminology

The design of a database is traditionally done in an entity relationship diagram (ERD). However, the UML generalizes ERDs into class diagrams. This section bridges the traditional terminology to the newer terminology.

A **table** in a database is represented by a textbfclass in a class diagram. Well, almost. In database design, a table implies both the *structure* as well as the *data*. A class, on the other hand, only includes the *structure*. Strictly speaking it is the **schema** of a table that is equivalent to a class.

A table has **rows** and **columns**. A column in a table is an **attribute** of a class. A row in a table is an **object** "manufactured" from a class. It is also common to say a row in a table is an instance of a class.

Tables in a database are often related in a **relational** database. In practice, it means that a column of one table may contain the **foreign key** into the **primary key** of another table. This is represented by a relationship/link/association in a class diagram.

A **primary key** of a table is a column (or a collection of columns) such that each row can be uniquely identified. A **foreign key** is a column (or a collection of columns) of a table that is used to lookup rows in another table.

17.2 The Everything-in-one-table Approach

This is the most intuitive approach. Unfortunately, it does not scale well as the number of records increases. It is also very rigid and inefficient. But, let's take a look anyway.

Let's say I am trying to represent the concept of a class in an application that tracks the overall grade of each students. The most intuitive solution is as follows:

- A class has a name, such as CISP457
- A class has an instructor, such as Tak Auyeung
- A class has a meeting place, such as room 121
- A class has a meeting time, such as 5:30pm
- A class has up to 30 students
  - A student has a name
  - A student has an ID
  - A student has an overall grade

You *can* put all this into one table. However, it is a bad design for the following reasons:

- To make a new "class", we need to duplicate the name, description and everything that has to do with the "course" and not the class.
- If a class only has 16 students, the record still reserves room for 30 students. This is wasteful.
- If we need to increase the max. capacity to 40, we'll need to change the table's schema. It also leads to more waste.
- It becomes difficult to search by student.
- If a student is enrolled in many classes, there will be duplicate information (name) stored in the system.

For these reasons, the above everything-in-a-table approach is not exactly helpful.

17.3 Canonicalization / Normalization

This is the process to make the tables of a database as "simple" as possible. A good reference is at http://dev.mysql.com/tech-resources/articles/intro-to-normalization.html. This section presents a somewhat digested form of the same material.

17.3.1 First Normal Form (Horizonal/Row)

Each table in a database that is in first normal form has the following properties:

- Each column is responsible for only one individual piece of information.
  - This means we do not want to use commas to separate different pieces of individual data in one column
  - For example, we do not want to have a single `PROF` column, but use `AUYEUNG, ANTOS` as the value if both professors co-teach a class.
- No two columns are used to represent data that is similar or the same in nature.
  - This means we do not want to use `PROF1` and `PROF2` in our current example.

The essence of the first normal form is that *multiplicity of more than one should not be handled in a single row.* Or, in other words, values of the same row must be related by one-to-one relationships.

17.3.2 Second Normal Form (Verical/Column)

Each table in a database that is in second normal form has the following attributes:

- Unless a column is a foreign key, it cannot contain duplicate values on different rows.

In our example, the classroom `RM121` is the value of the `CLASSROOM` column of all classes scheduled for room 121. This is a violation of the second normal form.

17.3.3 Third Normal Form (Dependency)

Each table in a database that is in third normal form has the following attributes:

- All columns of a table must directly depend on the primary key of the table.

In most tables for users, we lump the following columns together:

- `ADDR1`: First line of address, usually mandatory.
- `ADDR2`: Second line of address, usually optional.
- `STATE`: State.
- `ZIP`: ZIP code.

- `Country`: Country code.

And, like most such tables, there is usually an unique field to identify a user, such as `USERID`.

First of all `ADDR1` really has two pieces of information, the street name and street number. So, according to the first normal form, we should have split it into `STNUM` and `STREET`. In the proper third normal form, we have the following columns in a user table:

- `STNUM`: street number
- `STREETID`: a street ID, not an actual street name. This is a foreign key.

A separate table tracks streets, there are the columns of that table:

- `ID`: primary key, probably an integer.
- `NAME`: name of the street, such as "Main Street".
- `ZIP`: the ZIP code, such as "94231-5323". This column is a foreign key into the next able.

Now, we need another table to indicate which ZIP code belongs to which state:

- `ZIP`: the ZIP code, this is the primary key of this table. Example of value: "94231-5323".
- `STATEID`: the ID of a state. This is an integer, it is also a foreign key to the next table.

The following table defines states:

- `ID`: the primary key, an integer type.
- `NAME`: state name, such as "California".
- `SHORT`: the short name, such as "CA".
- `COUNTRYID`: the country code (an integer) that represents the country. This is a foreign key to the next table.

Finally, we have the country table:

- `ID`: the primary key, an integer type.
- `NAME`: country name, such as "The United States of American".
- `SHORT`: the short name of a country, such as "US".

As you can see, make a third normal form table is not that easy. In this example, we have to create three additional tables: ZIP, STATE and COUNTRY.

17.3.4 Tell me again, why?

The three normal forms are not just there as a intellectual exercise. Each one serves specific purposes. In this section, let us reexamine the three normal forms, and relate them to actual system performance factors.

1NF

"No column should contain multiple individual data items, and no two columns should contain similar information that is interchangeable."

Let us examine what happens when a table is not in 1NF. For our first example, let us consider a row containing `"ANTOS,AUYEUNG"` as `lastnames` of professors. While this looks fine to a person, it is not so in a database.

We can safely assume that `lastnames` is indexed. This means the database maintains some additional data that makes it fast to access a particular value. In the index, `"ANTOS,AUYEUNG"` is considered one item. In other words, if we try to search for `AUYEUNG`, the index does not match `"ANTOS,AUYEUNG"`.

In many database implementations, we can also use non-exact matches. However, non-exact matches are very slow precisely because the index structure cannot be utilized. As a result, even though *technically* we can still locate the row using just `AUYEUNG` or `ANTOS`, the efficiency is very poor when we need to resort to non-exact matching functions.

For our second example, let us consider the case where we have two columns, `PROF1` and `PROF2`. This way, we can have `ANTOS` in `PROF1`, and `AUYEUNG` in `PROF2`. What is the problem this time around?

This construct makes it extremely difficult to construct queries. Because we usually don't know if a professor is listed as the first or second professor, we need to use `OR` to locate a row that may contain the professor as `PROF1` or `PROF2`. Without much discussion of SQL itself, the query needs to read like the following:

```
SELECT ..... WHERE PROF1 = "AUYEUNG" OR PROF2 = "AUYEUNG";
```

Efficiency is also affected. Even though we can create an index for `PROF1` and another one for `PROF2`, we need to search in two different indices.

In summary, 1NF is necessary to make indices efficient and queries simpler.

2NF

The second normal form states that "no column can contain duplicate entries unless they are foreign keys". Again, let's see what happens when a table does not conform to 2NF.

Let's say we have a column called `COURSENUM`, and different rows use values like `CISA315`, `CISP457` and etc. First of all, it requires at least 7 bytes for each course number on each row. Secondly, what if there is typo mistake, or a case inconformity?

The solution is to use another table called `COURSE` to store information about courses, including the course number. In the `COURSE` table, we can use a primary key that is simply an integer. A four-byte integer gives us more than 4 billion possible numbers, which should be fine as the primary key for courses.

In the `SECTION` table, instead of storing the course number, we now store a foreign key into the `COURSE` table. This yields a better storage efficiency right away (three bytes per row). However, more importantly, it is now impossible to mispell course numbers.

This is because most databases can enforce the integrity between a foreign key and a primary key. It is impossible to use a foreign key in the `SECTION` table that has no match in the `COURSE` table.

In summary, 2NF usually improves storage efficiency, and it always improves data integrity.

3NF

In its third normal form, "all columns in a table except for the primary key must depend on the primary directly and alone." 3NF is sometimes arguable whether it really helps in terms of real life performance (speed or storage). In many cases, 3NF is a "purity" factor that does not have positive practical impacts.

Nonetheless, 3NF still has its value. Even in our somewhat extreme example of dependencies among address components, there are some advantages of using the 3NF.

The enforcement that a street name must exist in the ZIP table helps preventing typo mistakes. The same applies when the city and state of an address is not typed, but rather implied by the ZIP code. This gives the end user an additional chance to catch mistakes. It is *possible* that "Main Street" exists for two different cities in the same state, and the end user may not catch the mistake (wrong city). However, it is still much better than using an open format for address entering.

So, what is the drawback of using 3NF? It increases the complexity of the database. Instead of using one single table, we now have 3 or more tables. The most often complaint, however, has to do with the user interface and completeness of the other tables. In the case of entering addresses, The city and state are now automatically filled in. The street name, on the other hand, needs to be selected from a list.

Whoever implements the system need to ensure that all street names are in the street name table, or allow new entries to be created. Both can be done, but it requires extra effort.

What if the user interface needs to be flexible to allow the old school "open format"? The system can check for consistency and report problems. For example, if "Main Street" is spelled as "Main St", an intelligent system should still be able to find a match, even though it takes more computation time. If "Davis" is mispelled as "Davies", the system can use the ZIP code to know that the user probably meant "Davis", and flag it as a "corrected" field.

Part V

Conclusion

Chapter 18

What's Next?

What's next? The answer, obviously, depends on your objectives. This chapter attempts to explore a few directions.

18.1 Occupations

In the discussion of systems analysis, we briefly touched on several related occupations. This section describes the occupations related to systems analysis so that you get a better idea of who is doing what. For those who do not have a determined career, I hope this helps your decision making process.

18.1.1 Systems Analysts

Systems analysts form the glue or cement of a systems development team. Let's revisit the main responsibilities of a systems analyst, and comment on favorable personality traits.

Communication with "the client".

A systems analyst discusses requirements, objectives and details of an information system with the client. The client, in most cases, is not an expert in computer related technologies. As a result, a systems analyst must know what types of questions to ask, how to ask the questions, and how to interpret the response.

For this responsibility, a systems analyst must communicate well both in written *and* spoken form. In addition, a systems analyst must dress professionally (at least while meeting with a client) and behave professionally. Good interpersonal communication skills are a plus.

Documenting the Systems Specifications.

Systems analysts document the specifications of an information system. The specifications can be documented in text, diagrams, charts and any other means that is appropriate. The *language* of systems specifications evolves over time, as do the tools. The topics we covered in this course is just the tip of the iceberg.

Note that documenting the systems specifications is a big responsibility because other processes of systems development hinge on it.

For this responsibility, a systems analyst must be logical, detail oriented and careful. "Logical" because most the design of an information system is, aferall, logical. "Detail oriented" because in the design of an information system, it is the details that makes or breaks the final implementation. "Careful" because if the specifications of a system is wrong, the rest of the process will fall apart.

Communicate with technical staff.

Even though a systems analyst should have broad and up-to-date knowledge about information systems, he/she cannot know everything about an information system. For instance, is 2Gb/s a sustainable bandwidth between a storage subsystem and a mainframe?

In order to ensure a system meets all the requirements, a systems analyst often needs to communicate with experts in various areas. This means a systems analyst must have sufficient background and overall technical knowledge. Otherwise, a discussion with an expert in a particular area can be ineffective.

It is important for a systems analyst to keep an eye on the trends of different information system technologies. *How* things work is not necessarily important to an analyst, but knowing *what* a particular device can do is important.

Problem solving.

In the design or redesign of an information system, a systems analyst may encounter problem solving sessions. A "problem" does not automatically mean a "bug". A problem is simply an issue that cannot be dealt with using established techniques and knowledge.

For example, an existing information system may require the database be shut down before data can be backed up. What if the client requests that the daily downtime (to backup the system) be completely eliminated?

Problem solving requires as much research and logical thinking as creative thinking. In our example, the first question is "has this been done already?" Researching on the internet and contacting the database vendor may yield canned solutions. If there is no available solution, the next logical step is to consider options. Do we need to change the database backend to one that supports live backup? Is it possible to deny just update queries while the database is being backed up? Is the client okay with denying update queries for the duration of backing up? Can we freeze one of the RAID images, apply pending queries, and back that up? Is it cost effective to run two redundant database servers so that we can pause one for backup? What do we do to synchronize the two servers?...

Team playing.

I know, this term is so cliche. Nonetheless, this is particularly important for systems analysts. The reason is quite obvious: a systems analyst gets to play with the whole team, all the time.

18.1.2 Code Developers/Programmers

Programmers implement the software of an information system based on the systems specifications. This group of people is as vital to an IT project as any

other group. Although the focus of this course is not on programming, we see enough elements of logic and systems behavior specifications to get a glimpse of programming as a career.

Understanding the systems specifications.

This part is crucial. Although a programmer needs not create any activity diagram or state diagram, he/she still needs to understand the systems specifications documented by systems analysts.

In reality, it is actually somewhat *rare* that programmers understand enough systems specifications to get the job done right. Whether a specification is documented in the UML or other languages, most programmers consider it as an internal document for the systems analysts. Or, they only refer to it when there is a question.

This is, unfortunately, one of the major causes of wasted resources. A systems specification document should be understood by all participants of a information systems development team. This ensures that everyone understands the information system exactly the same way.

Writing programs.

Programmers, naturally, write programs. Even with a systems specification available, writing the actual code to implement an information system can still be a challenging task.

Unlike a systems specification document, program code must conform to the syntax (grammar) of an artificial language. Most *good* programmers master a programming language much like a normal person with a second or even the native language. Also unlike diagrams in a systems specification, program code tends to be text-only. This also means it requires more abstract and symbolic thinking than creating diagrams.

Problem solving.

Problem solving in the context of programming often means debugging. Why is a program not behaving the way "it is supposed to"? The ability of effectively debug programs differentiates good programmers from the average.

Debugging and problem solving in programming require intense focus. Not everyone has the necessary intensity. Intense focus also often translate to "no sense of passage of time." Programmers tend to stay up late not because it is a desirable/fashionable thing to do, but simply because they do not really sense or care about the passage of time.

Pausing a debugging session as 5pm is an option, but it is very ineffective. Resuming a debugging session on the following day is difficult. This is because in the process of debugging, a lot of details and train(s) of thought are maintained in the mind. Pausing the process often means most of these pieces of information is lost. Only a small subset of the states of a debugging mind can be documented.

18.1.3 Human-Factor Engineer

This position is not a traditional component of a systems development team. However, more information system teams are including HFEs.

The main responsibilities of an HFE is to make an information system as seamless as possible to the business and end-users. Some tasks of an HFE overlap with those of a traditional systems analyst. However, an HFE does have unique qualifications and responsibilities.

An HFE usually has a background in psychology, and an HFE focuses more on the psychological factors of an information system. An HFE is much more concerned about how a user sees and use an information system than how things are done in the information system.

An HFE can be a "traditional" systems analyst wearing a different hat, or a different individual.

The following is a quote from a job description:

"The Human Factor Engineer Contractor (HFE) will provide specific support for midrange single function products. Human Factor support is defined by, but not limited to product definition and customer requirements development through user needs assessment, prototyping, task analysis, product evaluation and inspection, usability testing, participation in usability team meetings and product design reviews. The HFE support will include product specific development for design and written specification for the user interface to be implemented into the product. The HFE will also provide product designers and process engineers with technical information and design solutions using human factors engineering principles. In addition, the HFE will assist with the integration of these ergonomic principles in product software/hardware design in the early stages of development. The HFE will also perform task analysis and customer user requirements studies to generate new product concepts and hardware/software user interface requirements and design solutions. Lastly, the HFE will work with designers to develop a basic framework for each product usability study."

The matching requirements are quoted as follows:

"Requires a Masters degree or equivalent in a human factors related discipline (Engineering/ Computer Science/ Psychology/). Ability to prototype user interface concepts and in-depth knowledge of hardware/software design, user-centered design process, usability evaluation techniques, perception, cognition, experimentation, and statistics. Ability to work in cross-functional and cultural teams and to communicate and evangelize ideas. Demonstrated knowledge of human factors principles relevant to software/hardware user interface. Ability to do rapid prototyping.

Desired: Proven experience (2-5 years) in all aspects of the product life cycle including contextual inquiry, product design, usability testing, and product release."

User modeling.

An HFE needs to first understand what type of people are interacting with an information system. The modeling of end-users is independent of the information system. On the other hand, user modeling requires analytical skills as well as interpersonal skills.

Business Process Review

While it is the responsibilities of a systems analyst to understand the document the business processes/procedures from the client, it is the responsibilities of the HFE to review and suggest improvements based on human factors.

User Interface Design

With an HFE in the team, the basic rules to design user interfaces should be determined by him/her. Note that an HFE may not design each and every window and dialog in an information system.

Although the basic rules of GUI design is common across systems, specific systems have specific special requirements. For example, medical systems have special rules to conform to, which most likely require a specialized set of rules for GUI design.

18.2 Career Paths

Selection a career is a big decision. While I cannot assume any responsibilities for your success or failure, I can tell you what I know and what I think of different career paths.

18.2.1 Systems Analyst

In certain organizations, the career path of a systems analyst can begin as humbly as a computer operator. Due to the somewhat "fuzzy" job description, systems analysts often do not require bachelor's or master's degree in a computer related field.

Systems analyst positions are somewhat open-ended. Because project management is an integral process closely related to systems analysis, many systems analyst also wear the "project management" hat. This opens up career opportunities in upper management (beyond the scope of information systems projects).

As far as salary is concerned, the range is huge for systems analysts. Successful systems analysts for big corporations can exceed 300k/yr. [1], while the entry level salary is in the 40k/yr range. For the most part, the salary is proportional to the exponent of responsibilities and importance.

18.2.2 Code Developer

Programmers, especially when compared to systems analysts, are more pigeonholed. This means the entry requirements, career paths and salary range are more defined. Most jobs require a bachelor's degree in computer science or computer engineering.

Successful programmers often get promoted to team supervisors, but seldom beyond that. The salary range of programmers range from 30k/yr for beginners to about 150k/yr for exceptional and experienced ones.

[1] According to a CIS professor who worked in the industry for 30+ years.

18.3 How does this relate to you?

This course, CISP457, is required by the Computer Networking Management degree, the Database Management degree and the Microcomputer Applications degree. What does systems analysis have anything to do with the degree you are seeking?

We'll leave this one as an open discussion in the class.

Chapter 19

Keeping Yourself Up-to-date

No class can prepare you for the world, especially in the technology fields. This chapter discusses some resources (mostly free) that can help you stay up-to-date with computer/IT stuff.

19.1 SlashDot

http://slashdot.org is a member-contributed news site. It's moto is "News for Nerds. Stuff that matters."

You can sign up for a free membership, or just browse without registering. Topics at Slashdot range from software to sci-fi. While most of the articles are somewhat light hearted, Slashdot does keep readers up-to-date about technology and open source software.

Slashdot is definitely geek-oriented. One of the main "features" of slashdot is reader contributed discussion. Most topics get between one hunderd to two hundreds comments. I find that reading comments can be quite informative. Most other news sites do not include reading comments, or do not have enough readers to comment.

An active reader-base is important to keep discussions balanced. Otherwise, a Microsoft-aligned news site can post articles that are skewed to Microsoft, and an open source news site can post articles that are skewed to open source software.

Slashdot is also unique in that it is also very open source oriented. Many readers of Slashdot are users or proponents of open source software.

19.2 ComputerWorld

http://www.computerworld.com is a serious website for IT professionals and

CEOs. Articles at this site tends to be more business (and corporate business at that) oriented than technology oriented.

19.3 News.com

http://news.com.com is a somewhat serious computer technology oriented site. Its articles tend to be more technical than ComputerWorld. It is also organized under various subcategories. I consider News.com to be a balanced site that is technical, practical and complete.

19.4 The Register

http://theregister.co.uk is a British computer/information technology news site. It is similar to News.com in many ways, but has more focus on European news.