Chapter 1
Introduction

Contents

> "There is nothing new under the sun but there are lots of old things we don't know."
> —Ambrose Bierce, *The Devil's Dictionary*

Software engineering is a discipline for solving business problems by designing and developing software-based systems. As with any engineering activity, a software engineer starts with *problem definition* and applies tools of the trade to obtain a *problem solution*. However, unlike any other engineering, software engineering seems to require great emphasis on *methodology* or *method* for managing the development process, in addition to great skill with tools and techniques. Experts justify this with the peculiar nature of the problems solved by software engineering. These "wicked problems" can be properly defined only after being solved.

This chapter first discusses what software engineering is about and why it is difficult. Then we give a brief preview of software development. Next, cases studies are introduced that will be used throughout the book to illustrate the theoretical concepts and tools. Software object model forms the foundation for concepts and techniques of modern software engineering. Finally, the chapter ends by discussing hands-on projects designed for student teams.

1.1 What is Software Engineering?

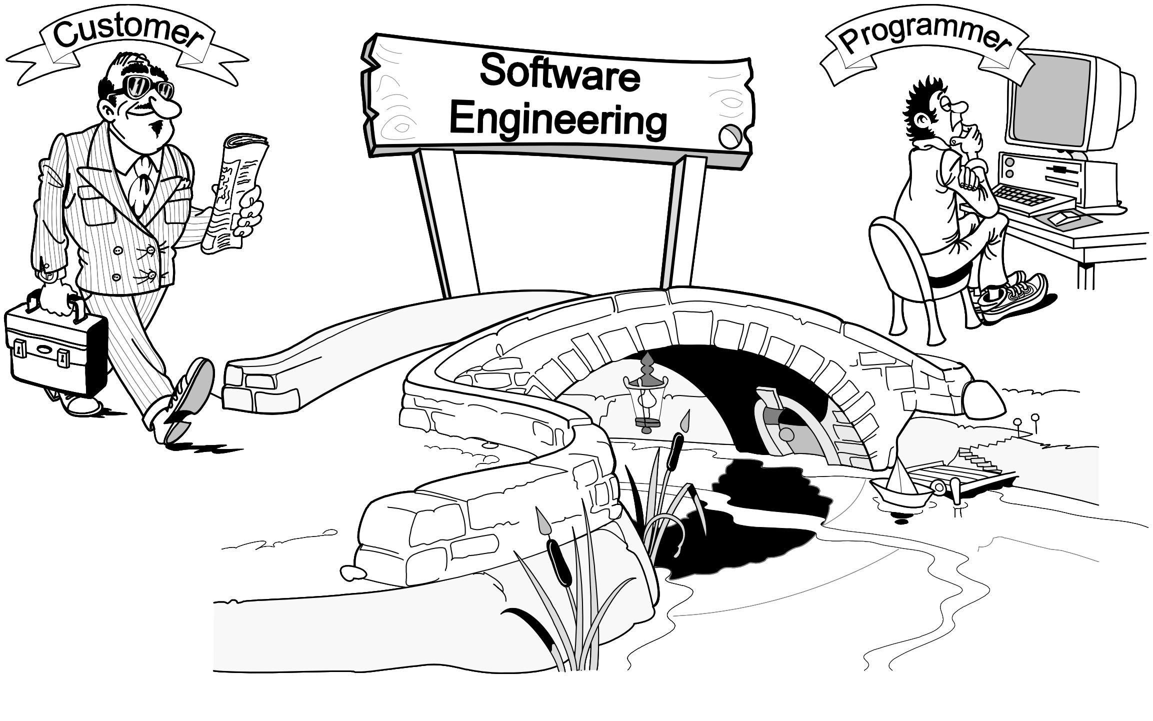
> "To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be." —Anonymous

> "Computer science is no more about computers than astronomy is about telescopes." —Edsger W. Dijkstra

The purpose of software engineering is to develop software-based systems that let customers achieve business goals. The customer may be a hospital manager who needs patient-record software to be used by secretaries in doctors' offices; or, a manufacturing manager who needs software to coordinate multiple parallel production activities that feed into a final assembly stage. Software engineer must understand the customer's business needs and design software to help meet them. This task requires

- The ability to quickly learn new and diverse disciplines and business processes
- The ability to communicate with domain experts, extract an abstract model of the problem from a stream of information provided in discipline-specific jargon, and formulate a solution that makes sense in the context of customer's business
- The ability to design a software system that will realize the proposed solution and gracefully evolve with the evolving business needs for many years in the future.

Software engineering is often confused with programming. Software engineering is the creative activity of understanding the business problem, coming up with an idea for solution, and designing the "blueprints" of the solution. Programming is the craft of implementing the given blueprints (Figure 1-1). Software engineer's focus is on *understanding* the interaction between the system-to-be and its users and the environment, and *designing* the software-to-be based on this understanding. Unlike this, programmer's focus is on the program code and ensuring that the code faithfully implements the given design. This is not a one-way process, because sometimes



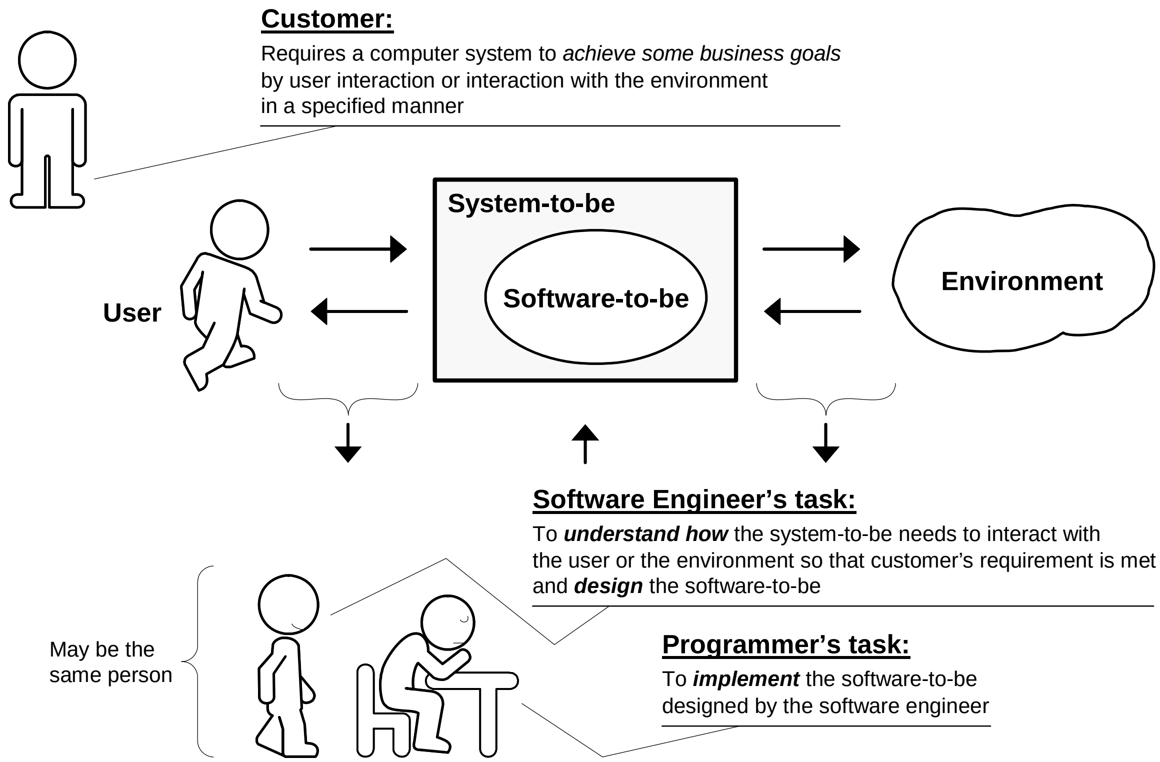
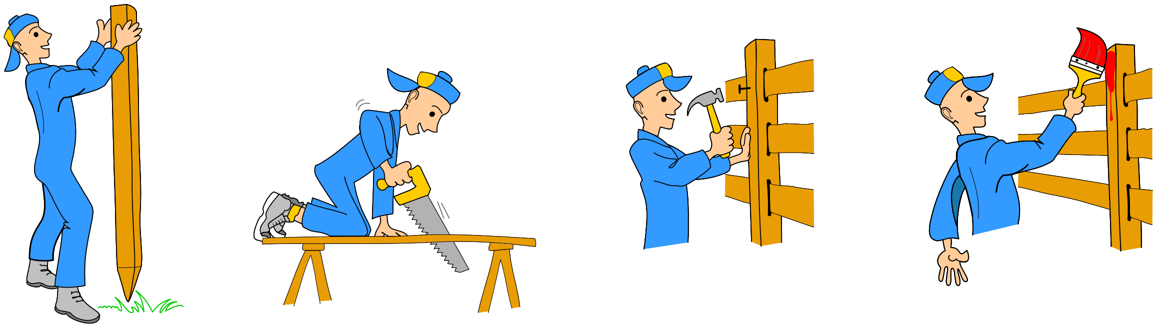

Customer:
Requires a computer system to *achieve some business goals* by user interaction or interaction with the environment in a specified manner
System-to-be
Software-to-be
User
Environment
Software Engineer's task:
To ***understand how*** the system-to-be needs to interact with the user or the environment so that customer's requirement is met and ***design*** the software-to-be
May be the same person
Programmer's task:
To ***implement*** the software-to-be designed by the software engineer

Figure 1-1: The role for software engineering.

the designs provided by the "artist" (software engineer) cannot be "carved" in "marble" (programming infrastructure) as given, and the "craftsman" (programmer) needs to work closely with the designer to find a workable solution. In an ideal world, both activities would be done by the same person to ensure the best result; in reality, given their different nature and demands, software engineering and programming are often done by different people.

Some people say software engineering is about writing loads of documentation. Other people say software engineering is about writing a running code. It is neither one. Software engineering is about understanding business problems, inventing solutions, evaluating alternatives, and making design tradeoffs and choices. It is helpful to document the process (not only the final solution) to know what alternatives were considered and why particular choices were made. But software engineering is not about writing documentation. Software engineering is about delivering value for the customer, and both code and documentation are valuable.



Setting posts Cutting wood Nailing Painting

Figure 1-2: Illustration of complexity on the problem of scheduling construction tasks.

I hope to convey in this text that software is many parts, each of which individually may be easy, but the problem is that there are too may of them. It is not the difficulty of individual components; it is the multitude that overwhelms you—you simply lose track of bits and pieces. Let me illustrate this point on a simple example. Suppose one wants to construct a fence around a house. The construction involves four tasks: setting posts, cutting wood, painting, and nailing (Figure 1-2). Setting posts must precede painting and nailing, and cutting must precede nailing. Suppose that setting posts takes 3 units of time, cutting wood takes 2 units of time, painting takes 5 units of time for uncut wood and 4 units of time otherwise, and nailing takes 2 units of time for unpainted wood and 3 units of time otherwise. In what order should these tasks be carried out to complete the project in the shortest possible time?

It is difficult to come up with a correct solution (or, solutions) without writing down possible options and considering them one by one. It is hard to say why this problem is complicated, because no individual step seems to be difficult. After all, the most complicated operation involves adding small integer numbers. Software engineering is full of problems like this: all individual steps are easy, yet the overall problem may be overwhelming.

Mistakes may occur both in understanding the problem or implementing the solution. The problem is, for discrete logic, closeness to being correct is not acceptable; one flipped bit can change the entire sense of a program. Software developers have not yet found adequate methods to handle such complexity, and this text is mostly dedicated to present the current state of the knowledge of handling the complexity of software development.

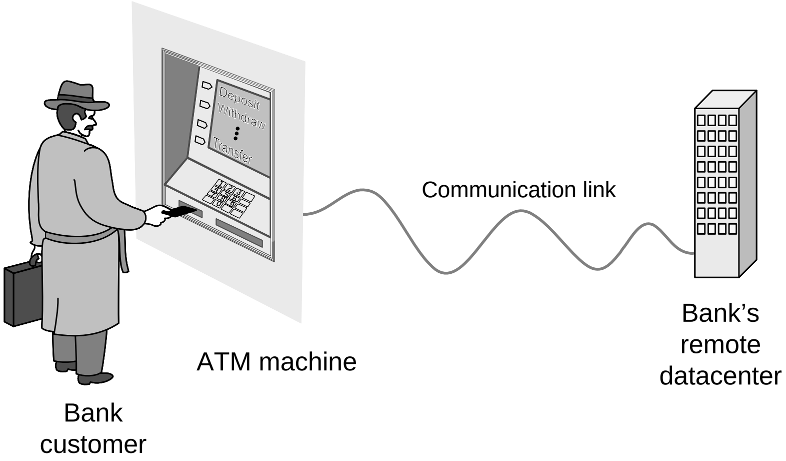


Figure 1-3: Example: developing software for an Automatic Teller Machine (ATM).

Software engineering relies on our ability to think about space and time, processes, and interactions between processes and structures. Consider an example of designing a software system to operate an automatic banking machine, known as Automatic Teller Machine (ATM) (Figure 1-3). Most of us do not know what is actually going on inside an ATM box; nonetheless, we could offer a naïve explanation of how ATM machines work. We know that an ATM machine allows us to deposit or withdraw money, and we can imagine how to split these activities into simpler activities to be performed by imaginary little "agents" working inside the machine. Figure 1-4 illustrates how one might imagine what should be inside an ATM to make it behave as it does. We will call the entities inside the system "concepts" because they are imaginary. As seen, there are two types of concepts: "workers" and "things."

We know that an ATM machine plays the role of a bank window clerk (teller). The reader may wonder why we should imagine many virtual agents doing a single teller's job. Why not simply imagine a single virtual agent doing the teller's job?! The reason that this would not help much is because all we would accomplish is to transform one complicated and inscrutable object (an ATM machine) into another complicated and inscrutable object (a virtual teller). To understand a complex thing, one needs to develop ideas about relationships among the parts inside. By dividing a complicated job into simpler tasks and describing how they interact, we simplify the problem and make it easier to understand and solve. This is why imagination is critical for software engineering (as it is for any other problem-solving activity!).

Of course, it is not enough to uncover the static structure of the system-to-be, as is done in Figure 1-4. We also need to describe how the system elements ("workers" and "things") interact during the task accomplishment. Figure 1-5 illustrates the working principle (or operational principle) of the ATM model from Figure 1-4 by a set of step-by-step interactions.

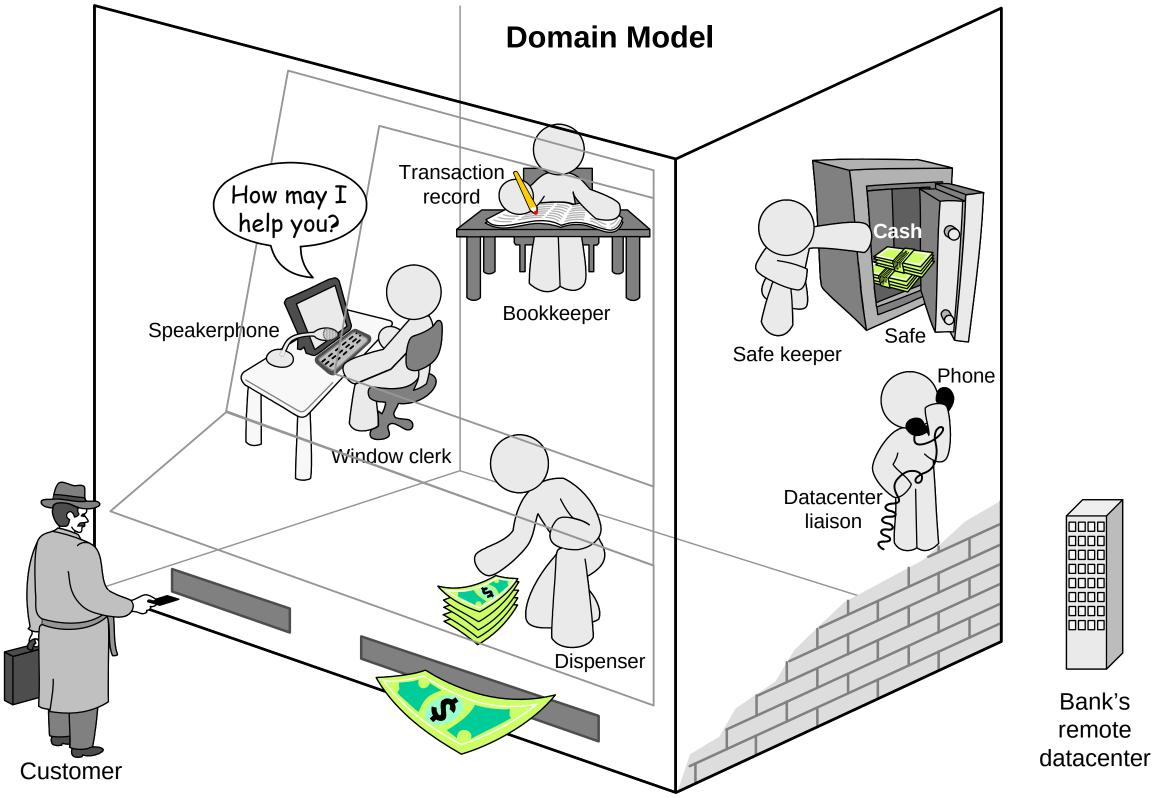


Figure 1-4: Imagined static structure of ATM shows internal components and their roles.

P

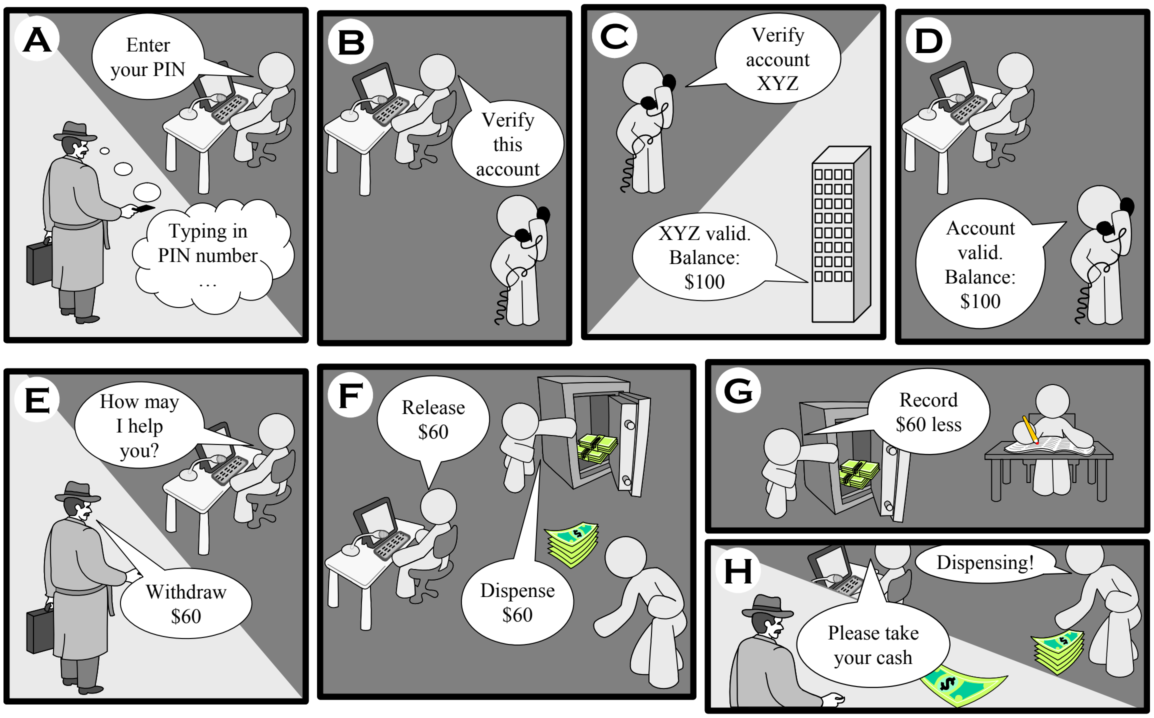


Figure 1-5: Dynamic interactions of the imagined components during task accomplishment.

rogramming language, like any other formal language, is a set of symbols and rules for manipulating them. It is when they need to meet the real world that you discover that associations can be made in different ways and some rules were not specified. A novice all too often sees only benefits of building a software product and ignores and risks. An expert sees a broader picture and anticipates the risks. After all, dividing the problem in subproblems and conquering them piecewise does not guarantee logical rigor and strict consistency between the pieces. Risks typically include conditions such as, the program can do what is expected of it and then some more, unexpected capabilities (that may be exploited by bad-intentioned people). Another risk is that not all environment states are catalogued before commencing the program development. Depending on how you frame your assumptions, you can come up with a solution. The troubles arise if the assumptions happen to be inaccurate, wrong, or get altered due to the changing world.

1.1.1 Why Software Engineering Is Difficult (1)

> "Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases." —Norman R. Augustine

If you are a civil engineer building bridges then all you need to know is about bridges. Unlike this, if you are developing software you need to know about *software domain* (because that is what you are building) and you need to know about the *problem domain* (because that is what you are building a solution for). Some problems require extensive periods of dedicated research (years, decades, or even longer). Obviously, we cannot consider such problem research as part of software engineering. We will assume that a theoretical solution either exists, or it can be found in a relatively short time by an informed non-expert.

A further problem is that software is a *formal* domain, where the inputs and goal states are well defined. Unlike software, the real world is *informal* with ill-defined inputs and goal states. Solving problems in these different domains demands different styles and there is a need to eventually reconcile these styles. A narrow interpretation of *software engineering* deals only with engineering the software itself. This means, given a precise statement of what needs to be programmed, narrow-scope software engineering is concerned with the design, implementation, and testing of a program that represents a solution to the stated problem. A broader interpretation of software engineering includes discovering a solution for a real-world problem. The real-world problem may have nothing to do with software. For example, the real-world problem may be a medical problem of patient monitoring, or a financial problem of devising trading strategies. In broad-scope software engineering there is no precise statement of what needs to be programmed. Our task amounts to none less than engineering of change in a current business practice.

Software engineering is mainly about modeling the physical world and finding good abstractions. If you find a representative set of abstractions, the development flows naturally. However, finding abstractions in a problem domain (also known as "application domain") involves certain level of "coarse graining." This means that our abstractions are unavoidably just *approximations*—we cannot describe the problem domain in perfect detail: after all that would require working at the level of atomic or even subatomic particles. Given that every physical system has very many parts, the best we can do is to describe it in terms of only some of its variables. Working with approximations is not necessarily a problem by itself should the world structure be never changing. But, we live in a changing world: things wear out and break, organizations go bankrupt or get acquired or restructured, business practices change, government regulations change, fads and fashions change, and so on. On a fundamental level, one could argue that the second law of thermodynamics works against software engineers (or anyone else trying to build models of the world), as well. The second law of thermodynamics states that the universe tends towards increasing disorder. Whatever order was captured in those comparatively few variables that we started with, tends to get dispersed, as time goes on, into other variables where it is no longer counted as order. Our (approximate) abstractions necessarily become invalid with passing time and we need to start afresh. This requires time and resources which we may not have available. We will continue discussion of software development difficulties in Sections 2.4.5 and 2.6.3.

Software development still largely depends on heroic effort of select few developers. Product line and development standardization are still largely missing, but there are efforts in this direction. Tools and metrics for product development and project management are the key and will be given considerable attention in this text.

1.1.2 Book Organization

Chapter 2 offers a quick tour of software engineering that is based on software objects, known as Object-Oriented Software Engineering (OOSE). The main focus is on tools, not methodology, for solving software engineering problems. Chapter 3 elaborates on techniques for problem understanding and specification. Chapter 4 describes metrics for measuring the software process and product quality. Chapter 5 elaborates on techniques for problem solution, but unlike Chapter 2 it focuses on advanced tools for software design. Chapter 6 describes structured data representation using XML. Chapter 7 presents software components as building blocks for complex software. Chapter 8 introduces service-oriented architectures and Web services.

I adopt an incremental and iterative refinement approach to presenting the material. For every new topic, we will scratch the surface and move on, only to revisit later and dig deeper.

The hope with metaphors and analogies is that they will evoke understanding much faster and allow “cheap” broadening it, based on the existing knowledge.

1.2 Software Engineering Lifecycle

> The Feynman Problem-Solving Algorithm:
> (i) Write down the problem (ii) think very hard, and (iii) write down the answer.

Any product development can be expected to proceed as an organized process that usually includes the following phases:

- Planning / Specification
- Design
- Implementation
- Evaluation

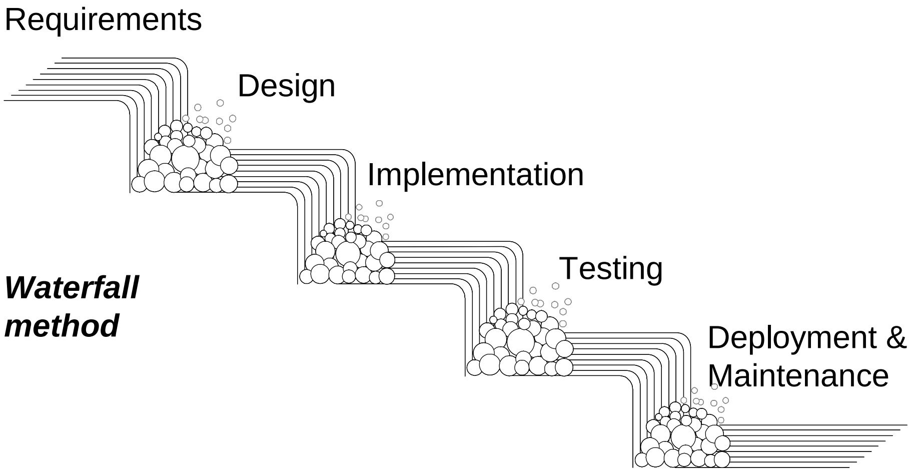


Figure 1-6: Waterfall process for software development.

So is with software development. The common software development phases are as follows:

1. Requirements Specification
   - Understanding the usage scenarios and deriving the *static* domain model
2. Design
   - Assigning responsibilities to objects and specifying detailed *dynamics* of their interactions under different usage scenarios
3. Implementation
   - Encoding the design in a programming language
4. Testing
   - Individual classes/components (unit testing) and the entire system (integration testing)
5. Operation and Maintenance
   - Running the system; Fixing bugs and adding new features

The lifecycle usually comprises many other activities, some of which precede the above ones, such as marketing survey to determine the market need for the planned product. This text is restricted to engineering activities, usually undertaken by the software developer.

The early inspiration for software lifecycle came from other engineering disciplines, where the above activities usually proceed in a sequential manner (or at least it was thought so). This method is known as **Waterfall Process** because developers build monolithic systems in one fell swoop (Figure 1-6). It requires completing the artifacts of the current phase before proceeding to the subsequent one. In civil engineering, this approach would translate to: finish all blueprints neatly before starting construction; finish the construction before testing it for soundness; etc. There is also psychological attraction of the waterfall model: it is a linear process that leads to a conclusion by following a defined sequence of steps. However, over the years developers realized that software development is unlike any other product development in these aspects:

- Unlike most other products, software is *intangible* and hard to visualize. Most people experience software through what it does: what inputs it takes and what it generates as outputs
- Software is probably the most *complex* artifact—a large software product consists of so many bits and pieces as well as their relationships, every single one having an important role—one flipped bit can change the entire sense of a program
- Software is probably the most *flexible* artifact—it can be easily and radically modified at any stage of the development process, so it can quickly respond to changes in customer requirements (or, at least it is so perceived)

Therefore, software development process that follows a linear order of understanding the problem, designing a solution, implementing and deploying the solution, does not produce best results. It is easier to understand a complex problem by implementing and evaluating pilot solutions. These insights led to adopting *incremental and iterative* (or, *evolutionary*) development methods, which are characterized by:

1. Break the big problem down into *smaller pieces* (increments) and *prioritize* them.
2. In each iteration *progress through* the development in more depth.
3. Seek the customer *feedback* and change course based on improved understanding.

Incremental and iterative process seeks to get to a working instance[1] as soon as possible. Having a working instance available lets the interested parties to have something tangible, to play with, make inquiries and receive feedback. Through this experimentation (preferably by end users), unsuspected deficiencies are discovered that drive a new round of development using failures and the knowledge of things that would not work as a springboard to new approaches. This greatly facilitates the consensus reaching and building the understanding of all parties of what needs to be developed and what to expect upon the completion. So, the key of incremental and iterative methods is to progressively deepen the understanding or "visualization" of the target product, by both advancing and retracting to earlier activities to rediscover more of its features. A popular incremental and iterative process is called **Unified Process** [Jacobson *et al.*, 1999]. Methods that are even more aggressive in terms of short iterations and heavy customer involvement are characterized as **Agile**. The customer is continuously asked to prioritize the remaining work items and provide feedback about the delivered increments of software.

All lifecycle processes have a goal of incremental refinement of the product design, but different people hold different beliefs on how this is to be achieved. This has been true in the past and it continues to be true, and I will occasionally comment on different approaches. Personally, I enthusiastically subscribe to the incremental and iterative approach, and in that spirit the exposition in this text progresses in an incremental and iterative manner, by successively elaborating the software lifecycle phases. For every new topic, we will scratch the surface and move on, only to revisit later and dig deeper.

A quick review of existing software engineering textbooks reveals that software engineering is largely about management. Project management requires organizational and managerial skills

[1] This is not necessarily a prototype, because "prototype" creates impression of something to be thrown away after initial experimentation. Conversely, a "working instance" can evolve into the actual product.

such as identifying and organizing the many tasks comprising a project, allocating resources to complete those tasks, and tracking actual against expected/anticipated resource utilization. Successful software projects convey a blend of careful objective evaluation, adequate preparation, continuous observation and assessment of the environment and progress, and adjusting tactics.

It is interesting to compare the issues considered by Brooks [1975] and compare those of the recent agile methods movement—both put emphasis on *communication* of the development team members. My important goal here is, therefore, to present the tools that facilitate communication among the developers. The key such tools are:

- *Modular design*: Breaking up the system in modules helps to cope with complexity; we have already seen how the ATM system was made manageable by identifying smaller tasks and associated "modules" (Figure 1-4). Modules provide building blocks or "words" of a language when describing complex solutions.
- *Symbol language*: The Unified Modeling Language (UML) is used similar to how the symbols such as ∞, $\int$, ∂, and $\cap$, are used in mathematics. They abbreviate the exposition of the material and facilitate the reader's understanding of the material.
- *Project and product metrics*: Metrics for planning and measuring project progress, and metrics for measuring the quality of software products provide commonly agreeable tools for tracking the work quality and progress towards the completion.
- *Design heuristics*: Also known as *patterns*, they create a design language for naming and describing the best practices that were proven in many contexts and projects.

Decomposing a problem into simpler ones, so called divide-and-conquer approach, is common when dealing with complex problems. In software development it is embodied in *modularity*: The source code for a module can be written and maintained independently of the source code for other modules. As with any activity, the value of a structured approach to software development becomes apparent only when complex problems are tackled.

1.2.1 Symbol Language

"Without images we can neither think nor understand anything." —Martin Luther (1483-1546)

"There are only 10 types of people in this world. Those who know binary, and those who don't."
—Unknown

As part of a design process, it is essential to communicate your ideas. When describing a process of accomplishing a certain goal, person actually thinks in terms of the abbreviations and symbols as they describe the "details" of what she is doing, and could not proceed intelligently if she were not to do so. George Miller found in the 1950s that human short-term memory can store about seven items at a time [Miller, 1957]. The short-term memory is what we use, for instance, to remember a telephone number just long enough to look away from the paper on which it is written to dial the number. It is also known as *working memory* because in it information is assumed to be processed when first perceived. It has been likened to the RAM (random access memory) of a computer. Recall how many times you had to look back in the middle of dialing, particularly if you are not familiar with the area code, which makes the number a difficult 10 digits! It turns out that the Miller's hypothesis is valid for any seven "items," which could be anything, such as numbers, faces, people, or communities—as we organize information on higher

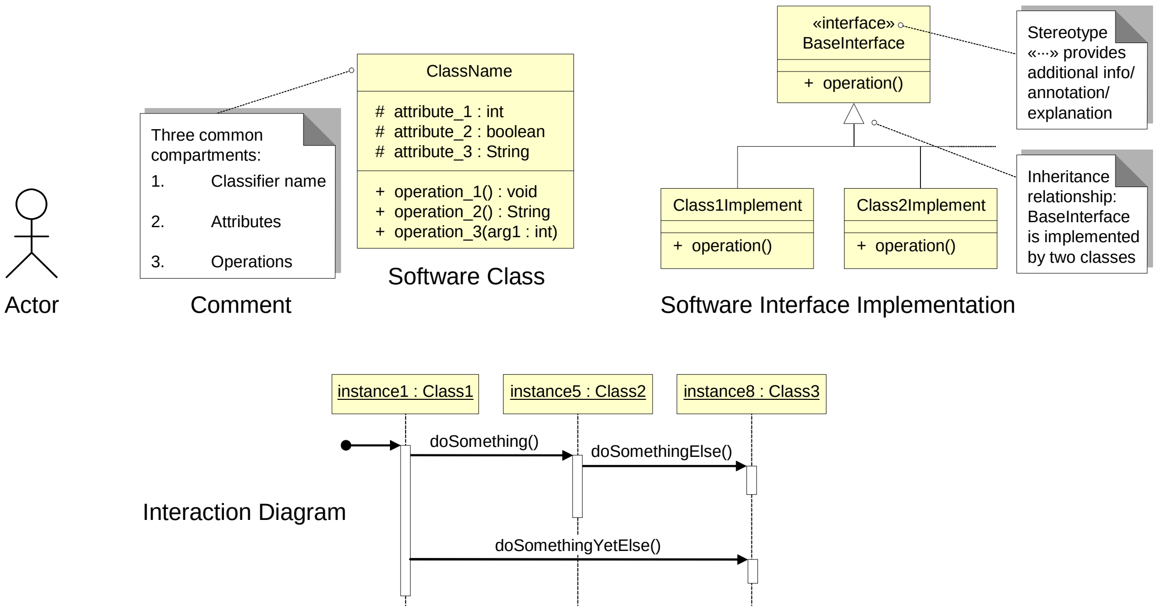


Figure 1-7: Example UML symbols for software concepts.

levels of abstraction, we can still remember seven of whatever it is. This item-level thinking is called *chunking*. Symbols can be easier chunked into patterns, which are represented by new symbols. Using symbols and hierarchical abstraction makes it easier for people to think about complex systems.

Diagrams and symbols are indispensible to the software engineer. Program code is not the best way to document a software system, although some agile methodologists have claimed that it is (more discussion in Section 2.1.1). Code is precise, but it is also riddled with details and idiosyncrasies of the programming language. Because it is essentially text, is not well-suited for chunking and abstraction. The visual layout of code can be used to help the reader with chunking and abstraction, but it is highly subjective with few widely accepted conventions.

Our primary symbol language is UML, but it is not strictly adhered to throughout the text. I will use other notations or an ad-hoc designed one if I feel that it conveys the message in a more elegant way. I would prefer to use storyboards and comic-strip sequences to represent that problem and solution in a comprehensible manner. On the other hand, they are time-consuming and often ambiguous, so we will settle for the dull but standardized graphics of the UML.

Example UML symbols are shown in Figure 1-7. To become familiar with UML, you can start at http://www.uml.org, which is the official standard's website. People usually use different symbols for different purposes and at different stages of progression. During development there are many ways to think about your design, and many ways to informally describe it. Any design model or modeling language has limits to what it can express and no one view of a design tells all. For example, strict adherence to a standard may be cumbersome for the initial sketches; contrariwise, documenting the completed design is always recommended in UML simply because so many people are already familiar with UML.

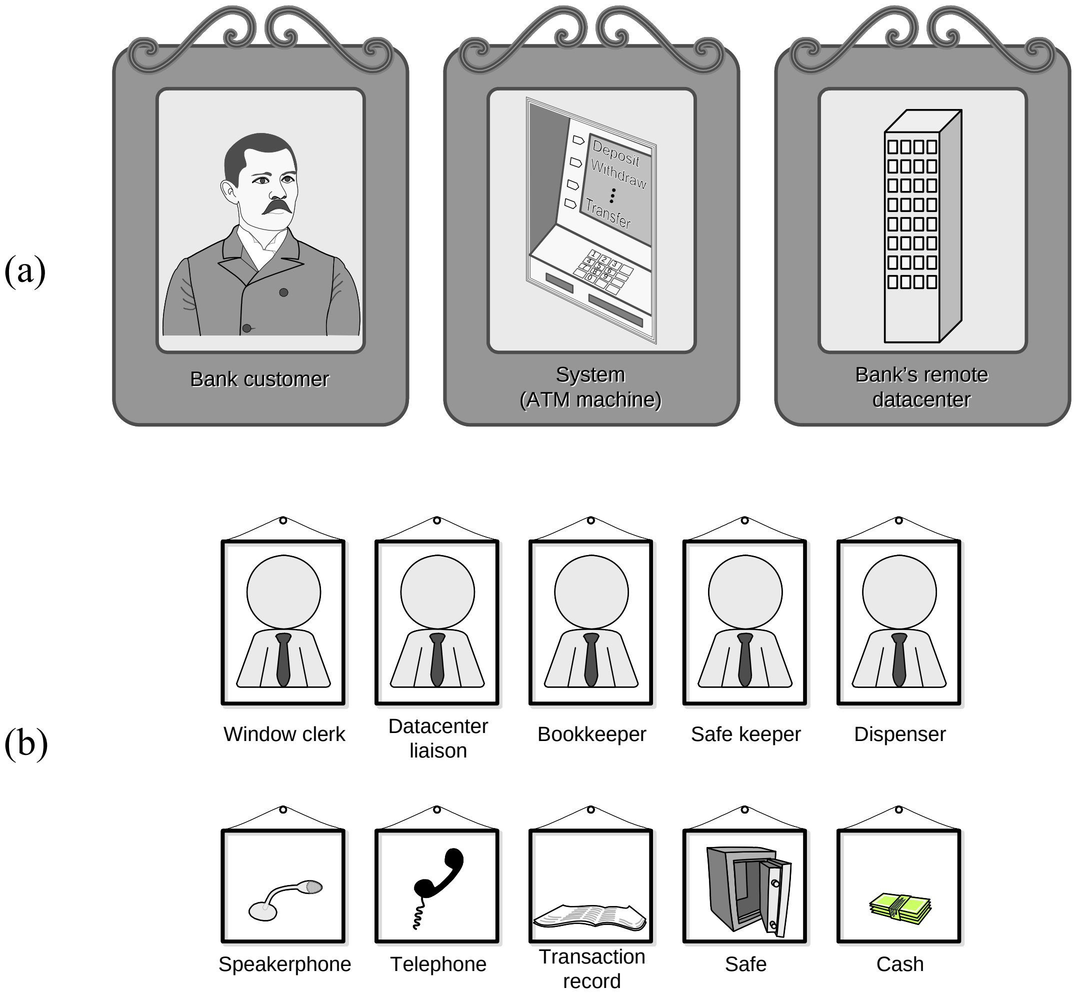


Figure 1-8: Gallery of actors (a) and concepts (b) of the system under discussion. The actors are relatively easy to identify because they are *external* to the system and visible; conversely, the concepts are hard to identify because they are *internal* to the system, hence invisible/imaginary.

As can be observed throughout this text, the graphic notation is often trivial and can be mastered relatively quickly. The key is in the skills in creating various models—it can take considerable amount of time to gain this expertise.

1.2.2 Requirements Analysis and System Specification

We start with the *customer statement of work* (also known as *customer statement of requirements*), if the project is sponsored by a specific customer, or the *vision statement*, if the project does not have a sponsor. The statement of work describes what the envisioned system-to-be is about, followed by a list of *features*/*services* it will provide or tasks/activities it will support.

Given the statement of work, the first step in the software development process is called ***requirements analysis*** or ***systems analysis***. During this activity the developer attempts to understand the problem and delimit its scope. The result is an elaborated statement of

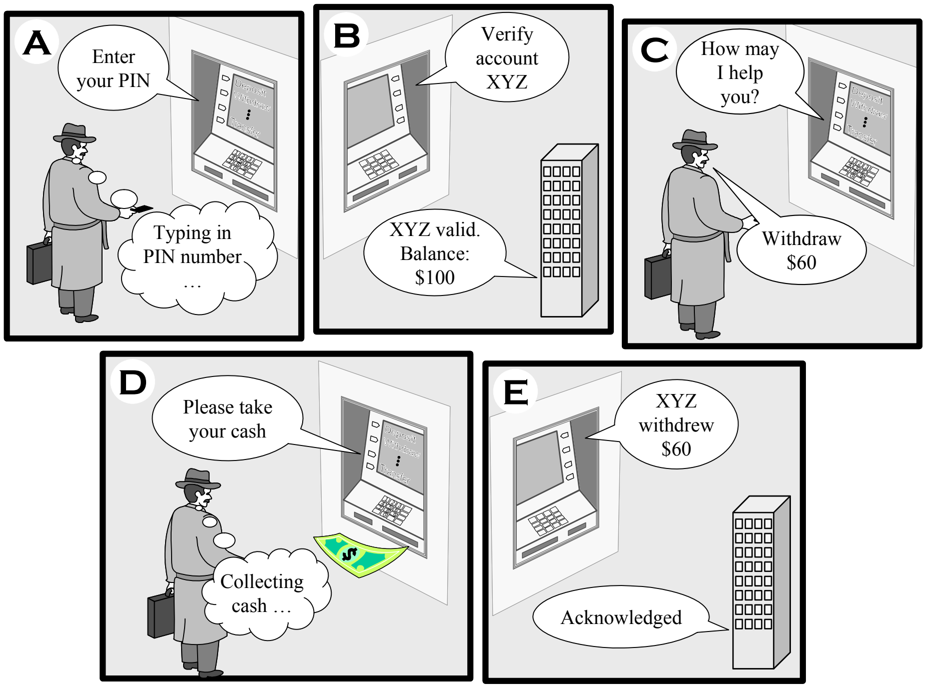


Figure 1-9: Scenario for use case "Withdraw Cash." Unlike Figure 1-5, this figure only shows interactions of the actors and the system.

requirements. The goal is to produce the system specification—the document that is an exact description of what the planned system-to-be is to do. Requirements analysis delimits the system and specifies the services it offers, identifies the types of users that will interact with the system, and identifies other systems that interact with ours. For example, the software engineer might ask the customer to clarify if the ATM machine (Figure 1-3) will support banking for customers of other banks or only the bank that owns the ATM machine. The system is at first considered a black box, its services ("push buttons") are identified, and typical interaction scenarios are detailed for each service. Requirement analysis includes both *fact-finding* of how the problem is solved in the current practice as well as *envisioning* how the planned system might work.

Recall the ATM example from Figure 1-3. We identified the relevant players in Figure 1-4. However, this may be too great a leap for a complex system. A more gradual approach is to start considering how the system-to-be will interact with the external players and defer the analysis of what happens inside the system until a later time. Figure 1-8(a) shows the players external to the system (called "actors"). If the ATM machine will support banking for customers of other banks, then we will need to identify additional actors.

A popular technique for requirements analysis is *use case modeling*. A set of use cases describes the elemental tasks a system is to perform and the relation between these tasks and the outside world. Each use case description represents a dialog between the user and the system, with the aim of helping the user achieve a business goal. In each dialog, the user initiates *actions* and the system responds with *reactions*. The use cases specify *what information must pass the boundary of the system* in the course of a dialog (without considering what happens *inside* the system). Because use cases represent recipes for user achieving *goals*, each use-case name must include a *verb* capturing the goal achievement. Given the ATM machine example (Figure 1-3), Figure 1-9 illustrates the flow of events for the use case "Withdraw Cash."

Use cases are only a beginning of software engineering process. When we elaborate use cases of a system, it signifies that we know *what* the system needs to accomplish, not *how*; therefore, it is *not* just "a small matter of system *building*" (programming) that is left after we specify the use cases. Requirements analysis is detailed in Sections 2.2 and 2.4.

1.2.3 Object-Oriented Analysis and the Domain Model

> "...if one wants to understand any complex thing—be it a brain or an automobile—one needs to develop good sets of ideas about the relationships among the parts inside. ...one must study the parts to know the whole." —Marvin Minsky, *The Emotion Machine*

Use cases consider the system as a black box and help us understand how the system as a whole interacts with the outside word. The next step is to model the inside of the system. We do this by building the *domain model*, which shows what the black box (the system-to-be) encloses. Given a service description, we can imagine populating the black box with domain concepts that will do the work. In other words, use cases elaborate the system's *behavioral characteristics* (sequence of stimulus-response steps), while the domain model details the system's *structural characteristics* (system parts and their arrangement) that make it possible for the system to behave as described by its use cases.

It is useful to consider a metaphor in which software design is seen as **creating a virtual enterprise** or an **agency**. The designer is given an enterprise's mission description and hiring budget, with the task of hiring appropriate workers, acquiring things, and making it operational. The first task is to create a list of positions with a job description for each position. The designer needs to identify the positions, the roles and responsibilities, and start filling the positions with the new workers. Recall the ATM machine example from Figure 1-3. We need to identify the relevant players internal to the system (called "concepts"), as illustrated in Figure 1-8(b).

In the language of requirements analysis, the enterprise is the system to be developed and the employees are the domain concepts. As you would guess, the key task is to hire the right employees (identify good concepts, or abstractions). Somewhat less critical is to define their relationships and each individual's attributes, which should be done only if they are relevant for the task the individual is assigned to. I like this metaphor of "hiring workers" because it is in the spirit of what Richard Feynman considered the essence of programming, which is "getting something to do something" [Feynman *et al.*, 2000]. It also sets up the stage for the important task of assigning responsibilities to software objects.

The idea for conducting object-oriented analysis in analogy to setting up an enterprise is inspired by the works of Fritz Kahn. In the early 20th century, Kahn produced a succession of books illustrating the inner workings of the human body, using visual metaphors drawn from industrial society. His illustrations drew a direct functional analogy between human physiology and the operation of contemporary technologies—assembly lines, internal combustion engines, refineries, dynamos, telephones, etc. Kahn's work is aptly referred to as "illustrating the incomprehendable" and I think it greatly captures the task faced by a software engineer. The interested reader should search the Web for more information on Kahn.

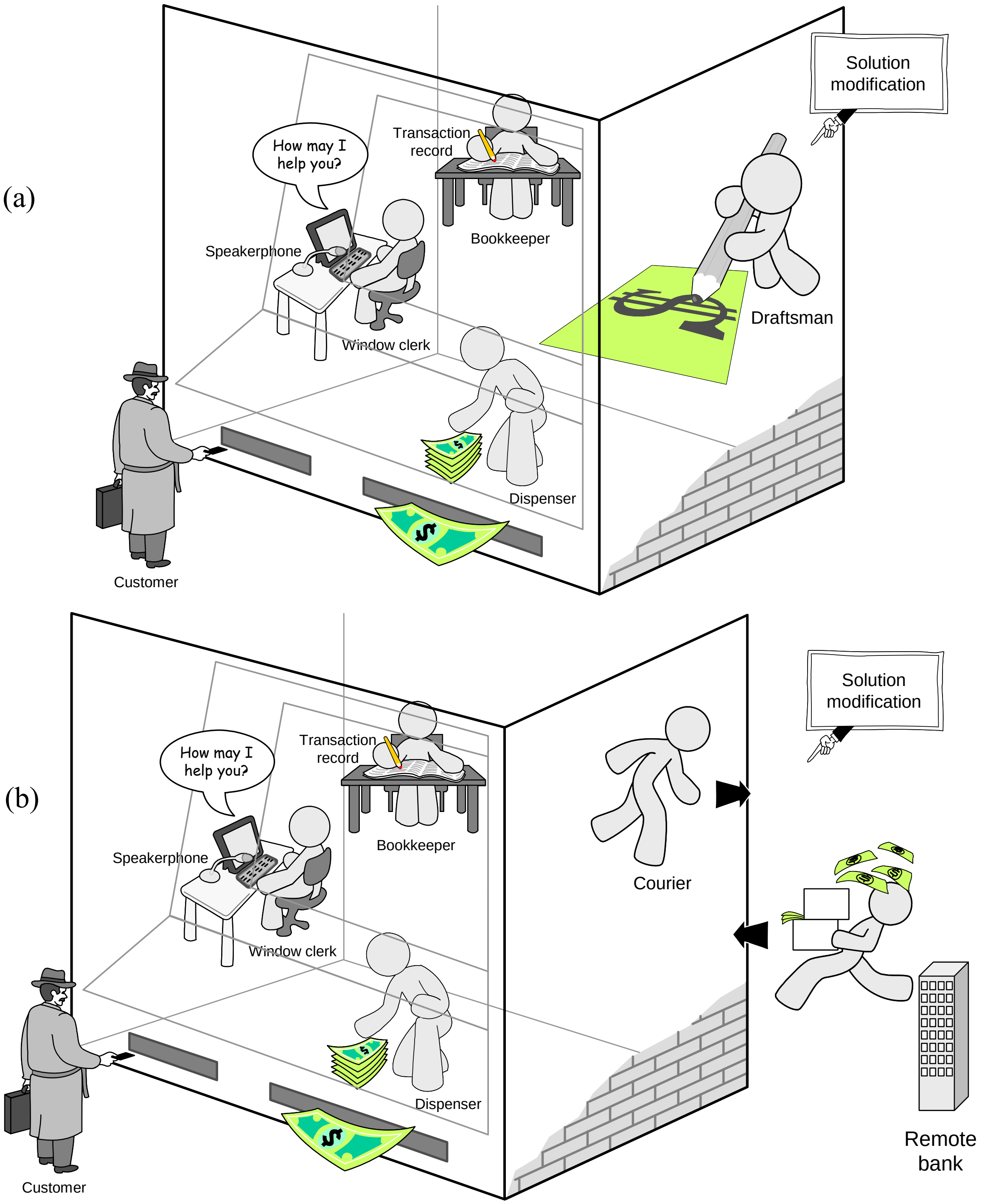


Figure 1-10: Alternative solutions for an ATM system. (Compare to Figure 1-4)

Domain analysis is more than just letting our imagination loose and imagining any model for the system-to-be. Design problems have unlimited number of alternative solutions. For example, consider again the design for an ATM system from Figure 1-4. One could imagine countless alternative solutions, two of which are shown in Figure 1-10. In Figure 1-10(a), we imagine having a draftsman to draw the banknotes requested by the customer then and there. In Figure 1-10(b), we imagine having a courier run to a nearest bank depository to retrieve the requested monies. How do we know which solution is best, or even feasible? Implementing and evaluating all imaginable solutions is impossible, because it takes time and resources. Two factors help constrain the options and shorten the time to solution:

- Knowing an existing solution for the same or similar problem
- Analyzing the elements of the external world that the system-to-be will interact with

I created the solution in Figure 1-4 because I have seen how banks with human tellers operate. I know that solutions in Figure 1-10 would take an unacceptable amount of time for each withdrawal, and the problem statement does not mention having a stack of blank paper and ink at disposal for solution in Figure 1-10(a), or having a runner at disposal for solution in Figure 1-10(b). The problem statement only mentions a communication line to a remote datacenter. There is nothing inherent in any of these solutions that makes some better than others. What makes some solutions "better" is that they copy existing solutions and take into account what is at our disposal to solve the problem. The implication is that the analyst needs to consider not only *what* needs to be done, but also *how* it can be done—what are feasible ways of doing it. We need to know what is at our disposal in the external world: do we have a stack of blank papers, ink, or a courier to run between the ATM and a depository? If this information is not given, we need to ask our customer to clarify. For example, the customer may answer that the system-to-be will have at disposal only a communication line to a remote datacenter. In this case, we demand the details of the communication protocol and the format of messages that can be exchanged. We need to know how will the datacenter answer to different messages and what exceptions may occur. We also need to know about the hardware that accepts the bank cards and disposes banknotes. How will our software be able to detect that the hardware is jammed?

Our abstractions must be grounded in reality, and the grounding is provided by knowing what is at the disposal in the external world that the system-to-be can use to function. This is why we cannot delimit domain analysis to what the black box (software-to-be) will envelop. Rather, we need to consider entities that are both external and internal to the software-to-be. The external environment constrains the problem to be solved and by implication constrains the internal design of the software-to-be. We also need to know what is implementable and what not, either from own experience, or from that of a person familiar with the problem domain (known as the "domain expert").None of our abstractions is realistic, but some are *useful* and others are not.

Object-oriented analysis is detailed in Section 2.5.

1.2.4 Object-Oriented Design

"Design is not just what it looks like and feels like. Design is how it works."—Steve Jobs

The act of design involves assigning form and function to parts so to create an esthetical and functional whole. In software development, the key activity in the design phase is *assigning responsibilities* to software objects. A software application can be seen as a set or community of interacting software objects. Each object embodies one or more roles, a role being defined by a set of related responsibilities. Roles, i.e., objects, collaborate to carry out their responsibilities. Our goal is to create a design in which they do it in a most efficient manner. Efficient design contributes to system performance, but no less important contribution is in making the design easier to understand by humans.

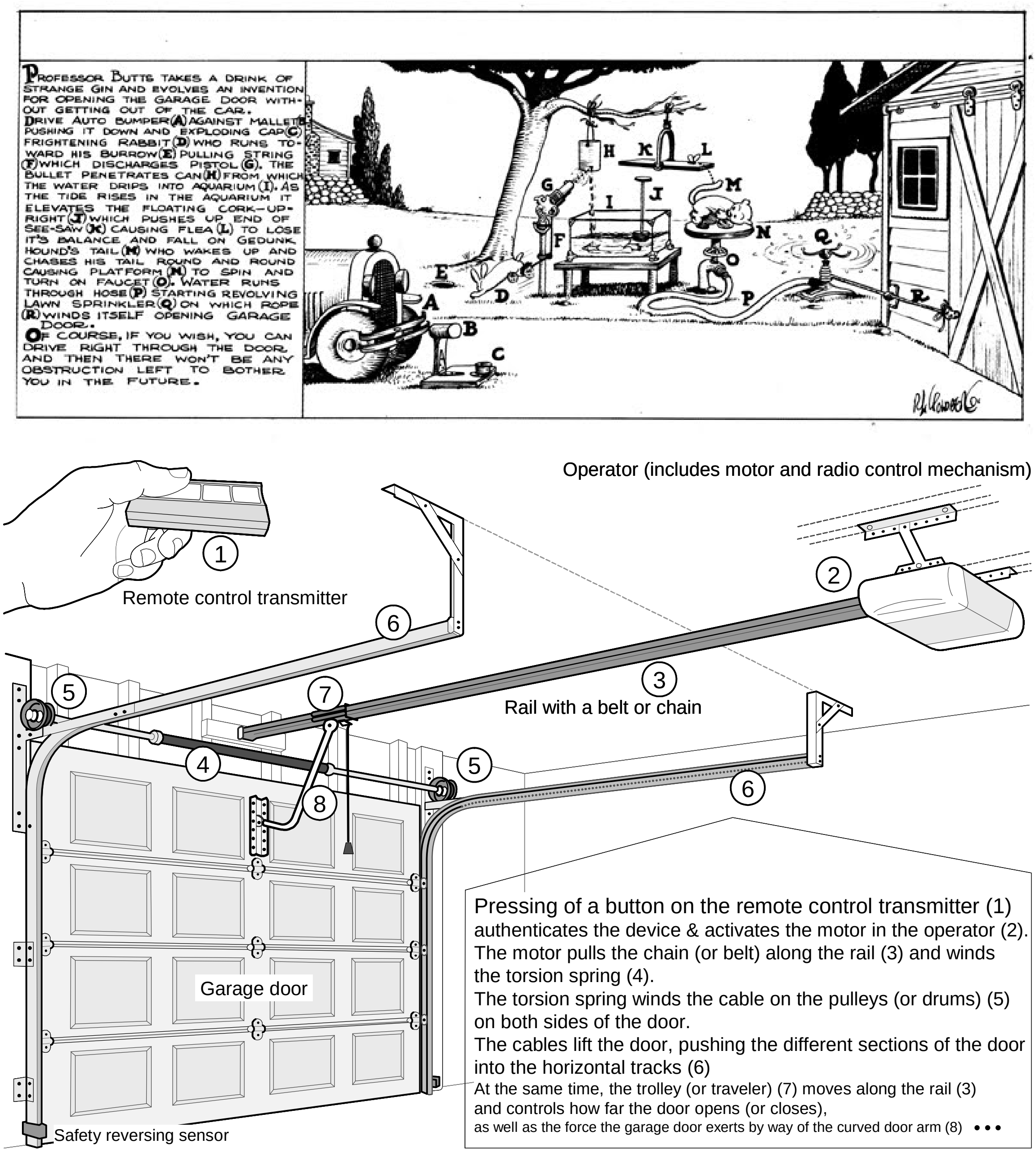
Design is the creative process of searching how to implement all of the customer's requirements. It is a problem-solving activity and, as such, is very much subject to trial and error. Breaking up the system into modules and designing their interactions can be done in many ways with varying quality of the results. In the ATM machine example, we came up with one potential solution for step-by-step interactions, as illustrated Figure 1-5. The key question for the designer is: is this the best possible way to assign responsibilities and organize the activities of virtual agents? One could solve the same design problem with a different list of players and different organization of their step-by-step interactions. As one might imagine, there is no known way for exactly measuring the optimality of a design. Creativity and judgment are key for good software design. Knowledge of rules-of-thumb and heuristics are critical in deciding how good a design is. Luckily, most design work is routine design, where we solve a problem by reusing and adapting solutions from similar problems.

So, what kinds of designs are out there? Two very popular kinds of software designs are what I would call Maurits Escher[2] and Rube Goldberg[3] designs. Both are fun to look at but have little practical value. Escher designs are impossible to implement in reality. Goldberg designs are highly-complicated contraptions, which solve the problem, but they are very brittle. If anything changes in the underlying assumptions, they fail miserably.

A key problem of design is that we cannot know for sure if a design will work unless we implement it and try it. Therefore, a software engineer who is also a skilled programmer has advantage in software design, because he knows from experience how exactly to implement the abstract constructs and what will or will not work. Related to this issue, some agile methodologists claim that program code is the only faithful representation of program design. Although it may be faithful, code alone is insufficient to understand software design. One also needs diagrams to "see the forest for the trees." Code also usually does not document the design objectives, alternative designs that were considered, merits of different designs, and the rationale for the chosen designs.

[2] Maurits Cornelis Escher (1898-1972) is one of the world's most famous graphic artists, known for his so-called impossible structures, such as Ascending and Descending, Relativity, his Transformation Prints, such as Metamorphosis I, Metamorphosis II and Metamorphosis III, Sky & Water I or Reptiles.

[3] Reuben Lucius Goldberg (Rube Goldberg) (1883-1970) was a Pulitzer Prize winning cartoonist, sculptor, and author. He is best known for his "inventions"—elaborate sets of arms, wheels, gears, handles, cups, and rods, put in motion by balls, canary cages, pails, boots, bathtubs, paddles, and live animals—that take simple tasks and make them extraordinarily complicated.



PROFESSOR BUTTS TAKES A DRINK OF STRANGE GIN AND EVOLVES AN INVENTION FOR OPENING THE GARAGE DOOR WITHOUT GETTING OUT OF THE CAR.
DRIVE AUTO BUMPER (A) AGAINST MALLET (B) PUSHING IT DOWN AND EXPLODING CAP (C) FRIGHTENING RABBIT (D) WHO RUNS TOWARD HIS BURROW (E) PULLING STRING (F) WHICH DISCHARGES PISTOL (G). THE BULLET PENETRATES CAN (H) FROM WHICH THE WATER DRIPS INTO AQUARIUM (I). AS THE TIDE RISES IN THE AQUARIUM IT ELEVATES THE FLOATING CORK-UPRIGHT (J) WHICH PUSHES UP END OF SEE-SAW (K) CAUSING FLEA (L) TO LOSE IT'S BALANCE AND FALL ON GEDUNK HOUND'S TAIL (M) WHO WAKES UP AND CHASES HIS TAIL ROUND AND ROUND CAUSING PLATFORM (N) TO SPIN AND TURN ON FAUCET (O). WATER RUNS THROUGH HOSE (P) STARTING REVOLVING LAWN SPRINKLER (Q) ON WHICH ROPE (R) WINDS ITSELF OPENING GARAGE DOOR.
OF COURSE, IF YOU WISH, YOU CAN DRIVE RIGHT THROUGH THE DOOR AND THEN THERE WON'T BE ANY OBSTRUCTION LEFT TO BOTHER YOU IN THE FUTURE.

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
Operator (includes motor and radio control mechanism)
1
Remote control transmitter
2
6
3
Rail with a belt or chain
5
7
4
5
8
6
Garage door
Safety reversing sensor

Pressing of a button on the remote control transmitter (1) authenticates the device & activates the motor in the operator (2).
The motor pulls the chain (or belt) along the rail (3) and winds the torsion spring (4).
The torsion spring winds the cable on the pulleys (or drums) (5) on both sides of the door.
The cables lift the door, pushing the different sections of the door into the horizontal tracks (6)
At the same time, the trolley (or traveler) (7) moves along the rail (3) and controls how far the door opens (or closes),
as well as the force the garage door exerts by way of the curved door arm (8) • • •

**Figure 1-11: Top row: A Rube Goldberg machine for garage door opening.
Bottom row: An actual design of a garage door opener.**

Consider the garage-door opener designs in Figure 1-11. The top row shows a Rube Goldberg design and the bottom row shows an actual design. What makes the latter design realistic and what is lacking in the former design? Some observations:

- The Rube Goldberg design uses complex components (the rabbit, the hound, etc.) with many unpredictable or uncontrollable behaviors; conversely, a realistic design uses specialized components with precisely controllable functions

- The Rube Goldberg design makes unrealistic assumptions, such as that the rabbit will not move unless frightened by an exploding cap.
- The Rube Goldberg design uses unnecessary links in the operational chain.

We will continue discussion of software design when we introduce the object model in Section 1.4. Recurring issues of software design include:

- *Design quality evaluation*: Optimal design may be an unrealistic goal given the complexity of real-world applications. A more reasonable goal is to find criteria for comparing two designs and deciding which one is better. The principles for good object-oriented design are introduced in Section 2.6 and elaborated in subsequent chapters.
- *Design for change*: Useful software lives for years or decades and must undergo modifications and extensions to account for the changing world in which it operates. Chapter 5 describes the techniques for modifiable and extensible design.
- *Design for reuse*: Reusing existing code and designs is economical and allows creating more sophisticated systems. Chapter 7 considers techniques for building reusable software components.

Other important design issues include design for security and design for testability.

1.2.5 Project Effort Estimation and Product Quality Measurement

I will show, on an example of hedge pruning, how project effort estimation and product quality measurement work hand in hand with incremental and iterative development, particularly in agile methods. Imagine that you want to earn some extra cash this summer and you respond to an advertisement by a certain Mr. McMansion to prune the hedges around his property (Figure 1-12). You have never done hedge pruning before, so you will need to learn as you go. The first task is to negotiate the compensation and completion date. The simplest way is to make a guess that you can complete the job in two weeks and you ask for a certain hourly wage. Suppose that Mr. McMansion agrees and happily leaves for vacation. After one week, you realize that you are much behind the schedule, so to catch up you lower the quality of your work. After two weeks, the hedges are pruned and Mr. McMansion is back from vacation. He will likely find many problems with your work and may balk at paying for the work done.

Now suppose that you employ incremental and iterative hedge pruning. You start by dividing the hedges into smaller sections, because people are better at guessing the relative sizes of object parts than the absolute size of an entire object. Suppose that you came up with the partitioning labeled with encircled numbers ① to ⑧ in Figure 1-12. Think of hedge pruning as traveling along the hedge at a certain *velocity* (while pruning it). The velocity represents your work *productivity*. To estimate the travel *duration*, you need to know the length of the path (or, path *size*). That is

$$\text{Travel duration} = \frac{\text{Path size}}{\text{Travel velocity}} \qquad (1.1)$$

Because you have never pruned hedges, you cannot know your velocity, so the best you can do is to guess it. You could measure the path size using a tape measure, but you realize there is a

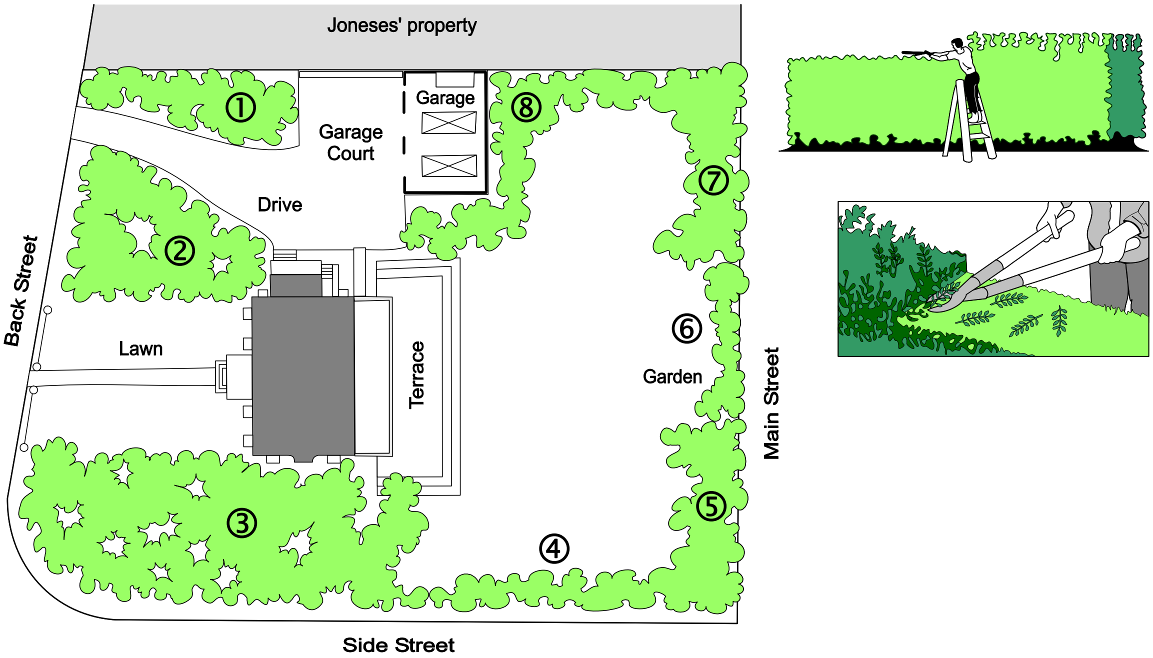
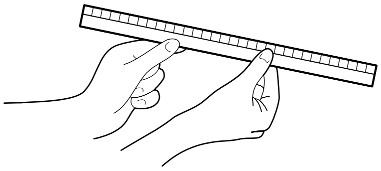


Figure 1-12: Example for project estimation: Formal hedge pruning.

problem. Different sections seem to have varying difficulty of pruning, so your velocity will be different along different sections. For example, it seems that section ③ at the corner of Back and Side Streets (Figure 1-12) will take much more work to prune than section ⑥ between the garden and Main Street. Let us assume you assign "pruning points" to different sections to estimate their size and complexity. Suppose you use the scale from 1 to 10. Because section ③ seems to be the most difficult, so we assign it 10 pruning points. The next two sections in terms of difficulty appear to be ② and ⑧, and relative to section ③ you feel that they are at about 7 pruning points. Next are sections ①, ⑤, and ⑦, and you give them 4 pruning points. Finally, section ④ gets 3 pruning points and section ⑥ gets 2 pruning points. The total for the entire hedge is calculated simply by adding the individual sizes

$$\text{Total size} = \sum_{i=1}^{N} \left(\text{points - for - section } i\right) \qquad (1.2)$$

Therefore, the total for the entire hedge is 10 + 2×7 + 3×4 + 3 + 2 = 41 pruning points. This represents your *size estimate* of the entire hedge. It is very important that this is a *relative-size estimate*, because it measures how big individual sections are relative to one another. So, a section estimated at four pruning points is expected to take twice as long work as a section estimated at two pruning points.



How accurate is this estimate? Should section ④ be weighted 3.5 points instead of 3? There are two parts to this question: (a) how accurate is the relative estimate for each section, and (b) is it appropriate to simply add up the individual sizes? As for the former issue, you may wish to break down the hedge sections into smaller parts, because it is easier to do eyeballing of smaller parts and comparing to one another. Section ③ is particularly large and it may be a good idea to split it up to smaller pieces. If you keep subdividing, in the extreme instead of eyeballing hedge sections you could spend weeks and count all the branches and arrive at a

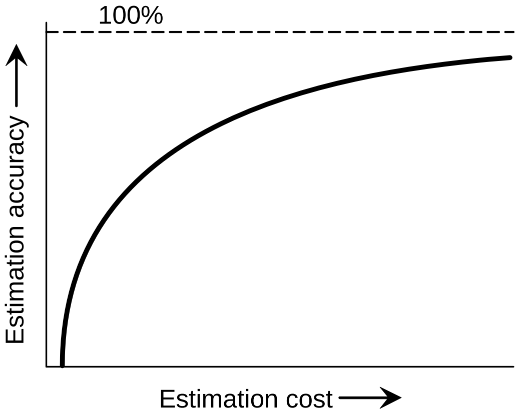


Figure 1-13: Exponential cost of estimation. Improving accuracy of estimation beyond a certain point requires huge cost and effort (known as the law of diminishing returns).

much more accurate estimate. You could even measure density of branches in individual sections, their length, hardness, etc. Obviously, there is a point beyond which only minor improvement in estimation accuracy is brought at a huge cost (known as the law of diminishing returns). Many people agree that the cost-accuracy relationship is exponential (Figure 1-13). It is also interesting to note that, in the beginning of the curve, we can obtain huge gains in accuracy with modest effort investment. The key points for size estimation are that (1) the pieces should be fairly *small* and (2) they should be of *similar* size, because it is easier to compare the relative sizes of small and alike pieces.

As for the latter issue about equation (1.2), the appropriateness of using a linear summation, a key question is if the work on one section is totally *independent* on the work on another section. The independence is equivalent to assuming that every section will be pruned by a different person and each starts with an equal degree of experience in hedge pruning. I believe there are confounding factors that can affect the accuracy of the estimate. For example, as you progress, you will learn about hedge pruning and become more proficient, so your velocity will increase not because the size of some section became smaller but because you became more proficient. In Section 2.2.3 I will further discuss the issue of linear superposition in the context of software project estimation.

All you need now is the velocity estimate, and using equation (1.1) you can give Mr. McMansion the estimate of how long the entire hedge pruning will take. Say you guess your velocity at 2 pruning points per day. Using equation (1.1) you obtain 41/2 ≈ 21 working days or 4 weeks. You tell Mr. McMansion that your initial estimate is 21 days to finish the work. However, you must make it clear that *this is just a guess, not a hard commitment*; you cannot make hard commitments until you do some work and find out what is your actual productivity (or "velocity"). You also tell Mr. McMansion how you partitioned the work into smaller items (sections of the hedge) and ask him to *prioritize* the items, so that you know his preferred ordering. Say that Mr. McMansion prefers that you start from the back of the house and as a result you obtain the work backlog list shown in Figure 1-14. He will inspect the first deliverable after one week, which is the duration of one *iteration*.

Here comes the power of *iterative and incremental* work. Given Mr. McMansion's prioritized backlog, you pull as many items from the top of the list as will fit into an iteration. Because the

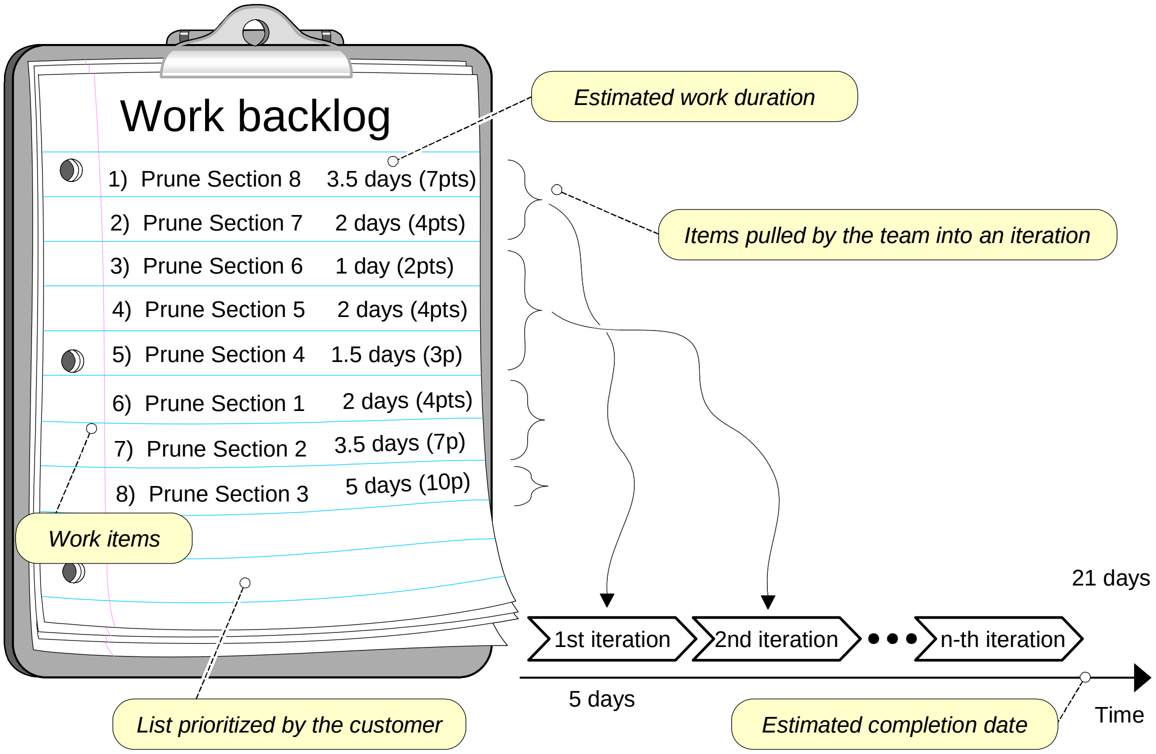

Work backlog
1) Prune Section 8 3.5 days (7pts)
2) Prune Section 7 2 days (4pts)
3) Prune Section 6 1 day (2pts)
4) Prune Section 5 2 days (4pts)
5) Prune Section 4 1.5 days (3p)
6) Prune Section 1 2 days (4pts)
7) Prune Section 2 3.5 days (7p)
8) Prune Section 3 5 days (10p)
Estimated work duration
Items pulled by the team into an iteration
Work items
List prioritized by the customer
1st iteration
2nd iteration
n-th iteration
21 days
5 days
Time
Estimated completion date

Figure 1-14: The key concepts for iterative and incremental project effort estimation.

first two items (sections ⑧ and ⑦) add up to 5.5 days, which is roughly one week, i.e., one iteration, you start by pruning sections ⑧ and ⑦. Suppose that after the first week, you pruned have about three quarters of the hedges in sections ⑧ and ⑦. In other words after the first iteration you found that your actual velocity is 3/4 of what you originally thought, that is, 1.5 pruning point per day. You estimate a new completion date as follows.

Total number of remaining points = 1/4 × 11 points remaining from sections ⑧ and ⑦
+ 30 points from all other sections
≈ 33 points

Estimated completion date = 22 days + 5 days already worked = 27 days total

You go to Mr. McMansion and tell him that your new estimate is that it will take you 27 days total, or 22 more days to complete the work. Although this is still an estimate and may prove incorrect, you are much more confident about this estimate, because it is based on your own experience. Note that you do not need to adjust your size estimate of 41 pruning points, because the relative sizes of hedge sections have not changed! Because of this velocity adjustment, you need to calculate new work durations for all remaining items in the backlog (Figure 1-14). For example, the new durations for sections ⑥ and ⑤ will be 1.3 days and 2.7 days, respectively. As a result, you will pull into the second iteration the remaining work from the first iteration plus sections ⑥ and ⑤. Section ④ that was originally planned for the second iteration (Figure 1-14) will be left for the third iteration.

It is important to observe that initially you *estimate* your velocity, but after the first increment you use the *measured* velocity to obtain a more accurate estimate of the project duration. You may continue measuring your velocity and re-estimating the total effort duration after each increment,

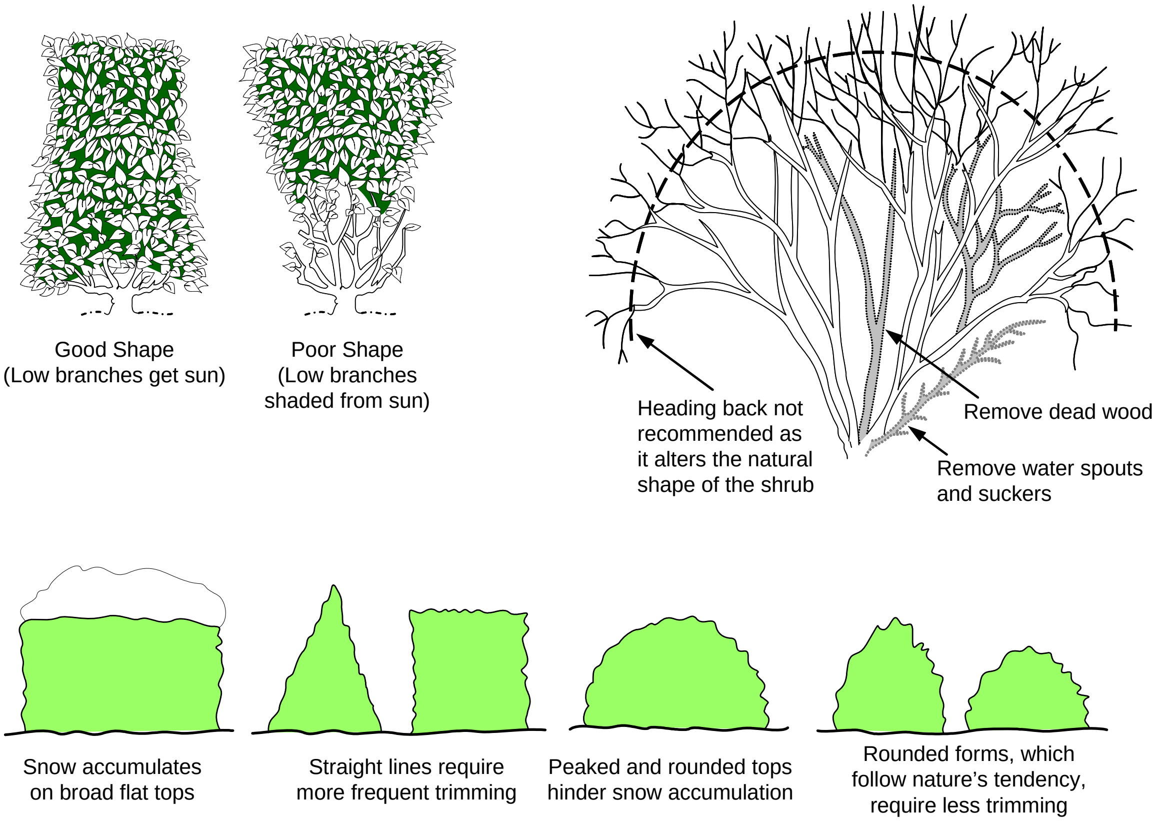

Good Shape
(Low branches get sun)
Poor Shape
(Low branches shaded from sun)
Heading back not recommended as it alters the natural shape of the shrub
Remove dead wood
Remove water spouts and suckers
Snow accumulates on broad flat tops
Straight lines require more frequent trimming
Peaked and rounded tops hinder snow accumulation
Rounded forms, which follow nature's tendency, require less trimming

Figure 1-15: Quality metrics for hedge pruning.

but this probably will not be necessary, because after the first few increments you will obtain an accurate measurement of your pruning velocity. The advantage of incremental work is that you can quickly gain accurate estimate of the entire effort and will not need to rush it later to complete on time, while sacrificing product quality.

Speaking of product quality, next we will see how *iterative* work helps improve product quality. You may be surprised to find that hedge pruning involves more than simply trimming the shrub. Some of parameters that characterize the quality of hedge pruning are illustrated in Figure 1-15. Suppose that after the first iteration (sections ⑧ and ⑦), Mr. McMansion can examine the work and decide if the quality is satisfactory or needs to be adjusted for future iterations.

It is much more likely that Mr. McMansion will be satisfied with your work if he is continuously consulted then if he simply disappeared to vacation after describing the job requirements. Regardless of how detailed the requirements description, you will inevitably face unanticipated situations and your criteria of hedge esthetics may not match those of Mr. McMansion. Everyone sees things differently, and frequent interactions with your customer will help you better understand his viewpoint and preferences. Early feedback will allow you to focus on things that matter most to the customer, rather than facing a disappointment when the work is completed. This is why it is important that the customer remains engaged throughout the duration of the project, and participates in all important decisions and inspects the quality of work any time a visible progress is made.

In summary, we use *incremental* staging and scheduling strategy to quickly arrive at an effort estimate and to improve the development *process quality*. We use the *iterative*, rework-scheduling strategy to improve the *product quality*. Of course, for both of these strategies it is essential to have good metrics. Project and product metrics are described in Chapter 4. We will also see in Section 2.2.3 how user-story points work similar to hedge-pruning points, and how they can be used to estimate development effort and plan software releases.

1.3 Case Studies

Two case studies will be used in examples throughout the text to illustrate software development techniques. In addition, several more projects are designed for student teams later in Section 1.5.

Both case studies (as well as student projects) address relatively complex problems. I favor complex projects, threading throughout the book, rather than simple, unconnected examples, because I feel that the former illustrate better the difficulties and merits of the solutions. Both projects are open-ended and without a clear objective, so that we can consider different features and better understand the requirements derivation process. My hope is that by seeing software engineering applied on complex (and realistic) scenarios, the reader will better grasp compromises that must be made both in terms of accuracy and richness of our abstractions. This should become particularly evident in Chapter 3, which deals with modeling of the problem domain and the system that will be developed.

Before we discuss the case studies, I briefly introduce a simple diagrammatic technique for representing knowledge about problem domains. **Concept maps**[4] are expressed in terms of *concepts* and *propositions*, and are used to represent knowledge, beliefs, feelings, etc. **Concepts** are defined as apperceived regularities in objects, events, and ideas, designated by a *label*, such as "green," "high," "acceleration," and "confused." A **proposition** is a basic unit of meaning or expression, which is an expression of the *relation* among *concepts*. Here are some example propositions:

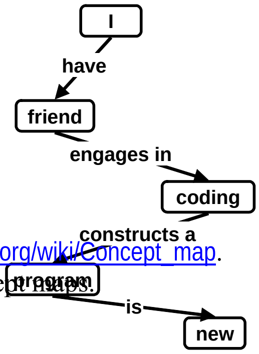
- Living things are composed of cells
- The program was flaky
- Ice cube is cold

We can decompose arbitrary sentences into propositions. For example, the sentence

"*My friend is coding a new program*"

can be written as the following propositions

Proposition	Concept	Relation	Concept
1.	I	have	friend



[4] A good introduction about concept maps can be found here: http://en.wikipedia.org/wiki/Concept_map. CmapTools (http://cmap.ihmc.us/) is free software that facilitates construction of concept maps.

2.	friend	engages in	coding
3.	coding	constructs a	program
4.	program	is	new

How to construct a concept map? A common strategy starts with listing all the concepts that you can identify in a given problem domain. Next, create the table as above, initially leaving the "Relation" column empty. Then come up with (or consult a domain expert for) the relations among pairs of concepts. Note that, unlike the simple case shown in the above table, in general case some concepts may be related to several other concepts. Finally, drawing the concept map is easy when the table is completed. We will learn more about propositions and Boolean algebra in Chapter 3.

Concept maps are designed for capturing static knowledge and relationships, not sequential procedures. A concept map provides a semiformal way to represent knowledge about a problem domain. It has reduced ambiguity compared to free-form text, and visual illustration of relationships between the concepts is easier to understand. I will use concepts maps in describing the case study problems and they can be a helpful tool is software engineering in general. But obviously we need other types of diagrammatic representations and our main tool will be UML.

1.3.1 Case Study 1: From Home Access Control to Adaptive Homes

Figure 1-16 illustrates our case-study system that is used in the rest of the text to illustrate the software engineering methods. In a basic version, the system offers house access control. The system could be required to *authenticate* ("Are you who you claim to be?") and *validate* ("Are you supposed to be entering this building?") people attempting to enter a building. Along with controlling the locks, the system may also control other household devices, such as the lighting, air conditioning, heating, alarms, etc.

As typical of most software engineering projects, a seemingly innocuous problem actually hides many complexities, which will be revealed as we progress through the development cycle. Figure 1-16 already indicates some of those—for example, houses usually have more than one lock. Shown are two locks, but there could be additional ones, say for a garage entrance, etc. Additional features, such as intrusion detection further complicate the system. For example, the house could provide you with an email report on security status while you are away on vacation. Police will also attend when they receive notification from a central monitoring station that a monitored system has been activated. False alarms require at least two officers to check on and this is a waste of police resources. Many cities now fine residents for excessive false alarms.

Here are some additional features to think about. You could program the system to use timers to turn lights, televisions and sound systems on and off at different times to give your home a "lived-in look" when you are away. Install motion-detecting outdoor floodlights around your home or automatic announcing of visitors with a chime sound. More gadgets include garage door openers, active badges, and radio-frequency identification (RFID) tags, to detect and track the tenants. Also, an outside motion sensor may turn on the outdoors light even *before* the user unlocks the door. We could dream up all sorts of services; for example, you may want to be able to open the door for a pizza-deliveryman remotely, as you are watching television, by point-and-

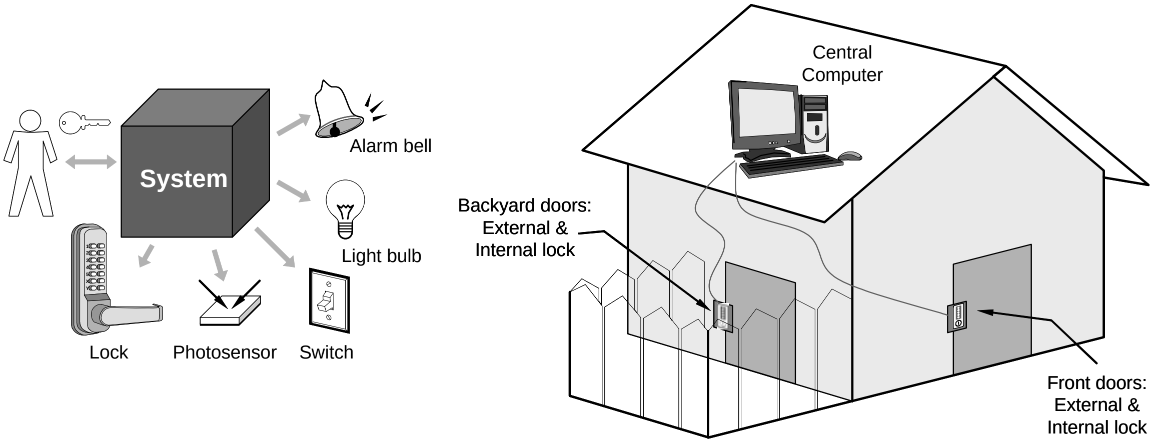


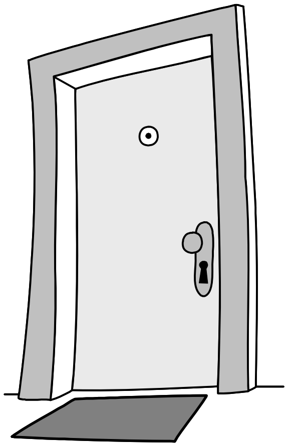
Figure 1-16: Our first case-study system provides several functions for controlling the home access, such as door lock control, lighting control, and intrusion detection and warning.

click remote controls. Moreover, the system may bring up the live video on your TV set from a surveillance camera at the doors.

Looking at the problem in a broader business context, it is unlikely that all or even the majority of households targeted as potential customers of this system will be computer-savvy enough to maintain the system. Hence, in the age of outsourcing, what better idea than to contract a security company to manage all systems in a given area. This brings a whole new set of problems, because we need to deal with potentially thousands of distributed systems, and at any moment many new users may need to be registered or unregistered with the (centralized?) system.

There are problems maintaining a *centralized* database of people's access privileges. A key problem is having a permanent, hard-wired connection to the central computer. This sort of network is very expensive, mainly due to the cost of human labor involved in network wiring and maintenance. This is why, even in the most secure settings, a very tiny fraction of locks tend to be connected. The reader should check for an interesting decentralized solution proposed by a software company formerly known as CoreStreet (http://www.actividentity.com/). In their proposed solution, the freshest list of access privileges spreads by "viral propagation" [Economist, 2004].

First Iteration: Home Access Control



Our initial goal is only to support the basic door unlocking and locking functions. Although at the first sight these actions appear simple, there are difficulties with both.

Figure 1-16 shows the locks connected by wire-lines to a central personal computer (PC). This is not necessarily how we want to solve the problem; rather, the PC just illustrates the problem. We need it to manage the users (adding/removing valid users) and any other voluminous data entry, which may be cumbersome from a lock's keypad—using a regular computer keyboard and monitor would be much more user friendly. The connections could be wireless,

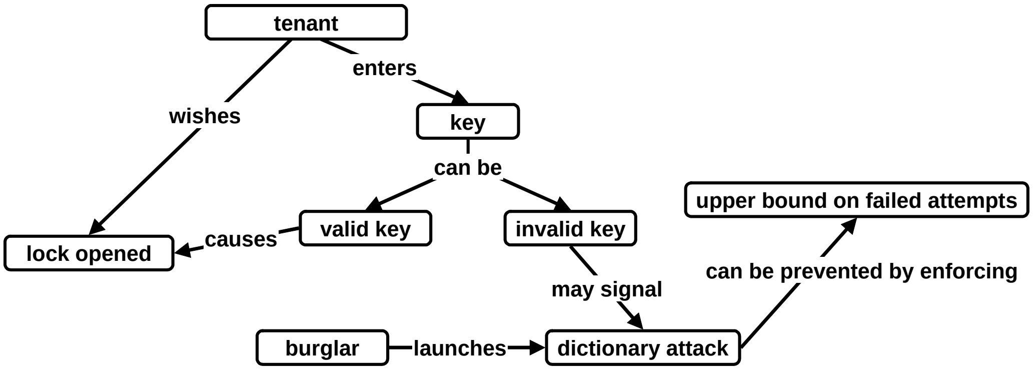


Figure 1-17: Concept map representing home access control.

and moreover, the PC may not even reside in the house. In case of an apartment complex, the PC may be located in the renting office.[5]

The first choice is about the user identification. Generally, a person can be identified by one of the following:

- What you carry on you (physical key or another gadget)
- What you know (password)
- Who you are (biometric feature, such as fingerprint, voice, face, or iris)

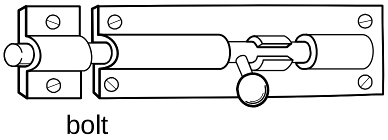
I start with two constraints set for this specific system: (1) user should not need to carry any gadgets for identification; and, (2) the identification mechanism should be cheap. The constraint (1) rules out a door-mounted reader for magnetic strip ID cards or RFID tags—it imposes that the user should either memorize the key (i.e., "password") or we should use biometric identification mechanism(s). The constraint (2) rules out expensive biometric identifiers, such as face recognition (see, e.g., http://www.identix.com/ and http://passfaces.com/) or voice recognition (see, e.g., http://www.nuance.com/prodserv/prodverifier.html). There are relatively cheap fingerprint readers (see, e.g., http://www.biometrics-101.com/) and this is an option, but to avoid being led astray by technology details, for now we assume that the user memorizes the key. In other words, at present we do not check the person's true identity (hence, no authentication)—as long as she knows a valid key, she will be allowed to enter (i.e., validation only).

For unlocking, a difficulty is with handling the failed attempts (Figure 1-17). The system must withstand "dictionary attacks" (i.e., burglars attempting to discover an identification key by systematic trial). Yet it must allow the legitimate user to make mistakes.

For locking coupled with light controls, a difficulty is with detecting the daylight: What with a dark and gloomy day, or if the photo sensor ends up in a shade. We could instead use the wall-clock time, so the light is always turned on between 7:30 P.M. and 7:30 A.M. In this case, the limits should be adjusted for the seasons, assuming that the clock is automatically adjusted for daylight saving time shift. Note

[5] This is an architectural decision (see Section 2.3 about software architecture).

also that we must specify which light should be turned on/off: the one most adjacent to the doors? The one in the hallway? The kitchen light? … Or, all lights in the house?



Interdependency question: What if the door needs to be locked after the tenant enters the house—should the light stay on or should different lights turn on as the tenant moves to different rooms?

Also, what if the user is not happy with the system's decision and does opposite of what the system did, e.g., the user turns off the light when the system turned it on? How do these events affect the system functioning, i.e., how to avoid that the system becomes "confused" after such an event?

Figure 1-18 illustrates some of the difficulties in specifying exactly what the user may want from the system. If all we care about is whether the door is unlocked or locked, identify two possible *states*: "unlocked" and "locked." The system should normally be in the "locked" state and unlocked only in the event the user supplies a valid key. To lock, the user should press a button labeled "Lock," but to accommodate forgetful users, the system should lock automatically autoLockInterval seconds after being unlocked. If the user needs the door open longer for some reason, she may specify the holdOpenInterval. As seen, even with only two clearly identified states, the rules for transitioning between them can become very complex.

I cannot overstate the importance of clearly stating the user's goals. The goal state can be articulated as ⟨unlocked AND light_on⟩. This state is of necessity temporary, because the door should be locked once the user enters the house and the user may choose to turn off the hallway light and turn on the one in the kitchen, so the end state ends up being ⟨lockeded AND light_off⟩. Moreover, this definition of the goal state appears to be utterly incomplete.

Due to the above issues, there are difficulties with unambiguously establishing the action preconditions. Therefore, the execution of the "algorithm" turns out to be quite complex and eventually we have to rely only on heuristics. Although each individual activity is simple, the combination of all is overwhelming and cannot be entirely solved even with an extremely complex system! Big software systems have too many moving parts to conform to any set of simple percepts. What appeared a simple problem turns out not to have an algorithmic solution, and on the other hand we cannot guarantee that the heuristics will always work, which means that we may end up with an unhappy customer.

Note that we only scratched the surface of what appeared a simple problem, and any of the above

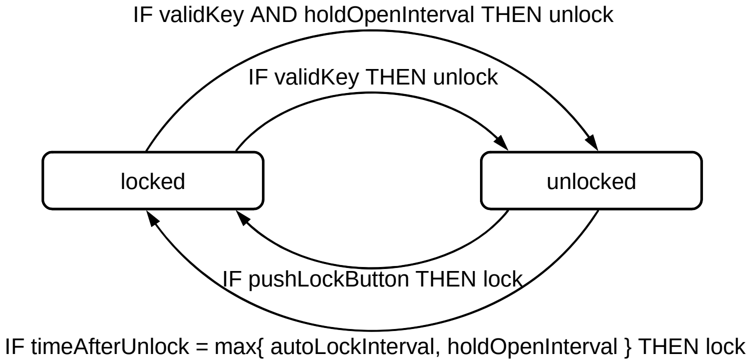

IF validKey AND holdOpenInterval THEN unlock
IF validKey THEN unlock
locked
unlocked
IF pushLockButton THEN lock
IF timeAfterUnlock = max{ autoLockInterval, holdOpenInterval } THEN lock

Figure 1-18: System states and transition rules.

issues can be further elaborated. The designer may be simply unable to explicitly represent or foresee every detail of the problem. This illustrates the real problem of heuristics: at a certain point the designer/programmer must stop discerning further details and related issues. But, of course, this does not mean that they will not arise sooner or later and cause the program to fail. And we have not mentioned program bugs, which are easy to sneak-in in a complex program. Anything can happen (and often does).

1.3.2 Case Study 2: Personal Investment Assistant

> "The way to make money is to buy stock at a low price, then when the price goes up, sell it.
> If the price doesn't go up, don't buy it." —Will Rogers

Financial speculation, ranging from straight gambling and betting to modern trading of financial securities, has always attracted people. For many, the attraction is in what appears to be a promise of wealth without much effort; for most, it is in the promise of a steady retirement income as well as preserving their wealth against worldly uncertainties. Investing in company equities (stocks) has carried the stigma of speculation through much of history. Only relatively recently stocks have been treated as reliable investment vehicles (Figure 1-19). Nowadays, more than 50% of the US households own stocks, either directly, or indirectly through mutual funds, retirement accounts or other managed assets. There are over 600 securities exchanges around the world. Many people have experience with financial securities via pension funds, which today are the largest investor in the stock market. Quite often, these pension funds serve as the "interface" to the financial markets for individual investors. Since early 1990s the innovations in personal computers and the Internet made possible for the individual investor to enter the stock markets without the help from pension funds and brokerage firms. The Internet also made it possible to do all kinds of researches and comparisons about various companies, in a quick and cheap fashion—an arena to which brokerage firms and institutional investors had almost exclusive access owing to their sheer size and money-might.

Computers have, in the eyes of some, further reduced the amount of effort needed for participation in financial markets, which will be our key motivation for our second case study: how to increase automation of trading in financial markets for individual investors. Opportunities for automation range from automating the mechanics of trading to analysis of how wise the particular trades appear to be and when risky positions should be abandoned.

There are many different financial securities available to investors. Most investment advisers would suggest hedging the risk of investment loss by maintaining a diversified investment portfolio. In addition to stocks, the investor should buy less-risky fixed income securities such as bonds, mutual funds, treasuries bills and notes or simply certificate of deposits. To simplify our case study, I will ignore such prudent advice and assume that our investor wants to invest in stocks only.

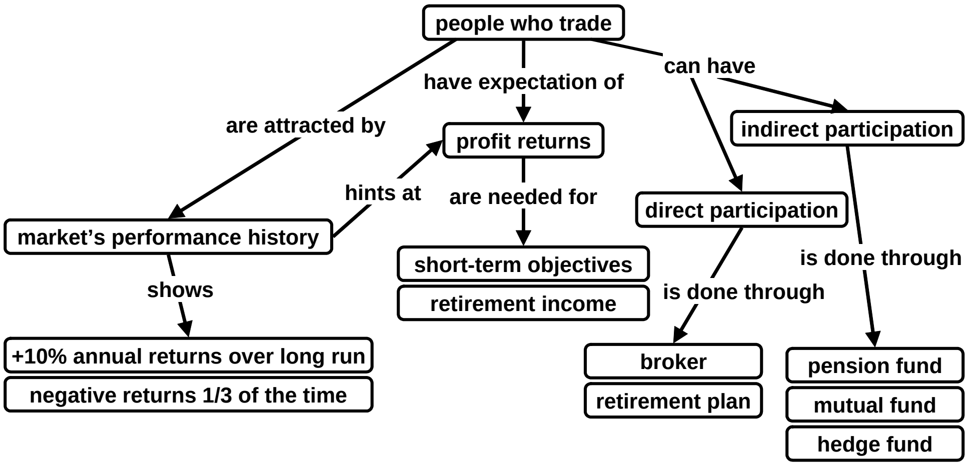


Figure 1-19: Concept map of why people trade and how they do it.

Why People Trade and How Financial Markets Work?

Anyone who trades does so with the expectation of making profits. People take risks to gain rewards. Naturally, this immediately begets questions about the kind of return the investor expects to make and the kind of risk he is willing to take. Investors enter into the market with varied objectives. Broadly, the investor objectives could be classified into short-term-gain and long-term-gain. The investors are also of varied types. There are institutional investors working for pension funds or mutual funds, and then there are day-traders and hedge-fund traders who mainly capitalize on the anomalies or the arbitrages that exist in the markets. Usually the institutional investors have a "long" outlook while the day-traders and the hedge-funds are more prone to have a "short" take on the market.

Here I use the terms "trader" and "investor" and synonymous. Some people use these terms to distinguish market participants with varied objectives and investment styles. Hence, an "investor" is a person with a long outlook, who invests in the company future by buying shares and holds onto them expecting to profit in long term. Conversely, a "trader" is a person with a short outlook, who has no long-term interest in the company but only looks to profit from short-term price variations and sells the shares at first such opportunity.

As shown in Figure 1-20(a), traders cannot exchange financial securities directly among themselves. The trader only *places orders* for trading with his broker and only accredited financial brokers are allowed to execute transactions. Before the Internet brokers played a more significant role, often provided investment advice in addition to executing transactions, and charged significant commission fees. Nowadays, the "discount brokers" mostly provide the transaction service at a relatively small commission fee.

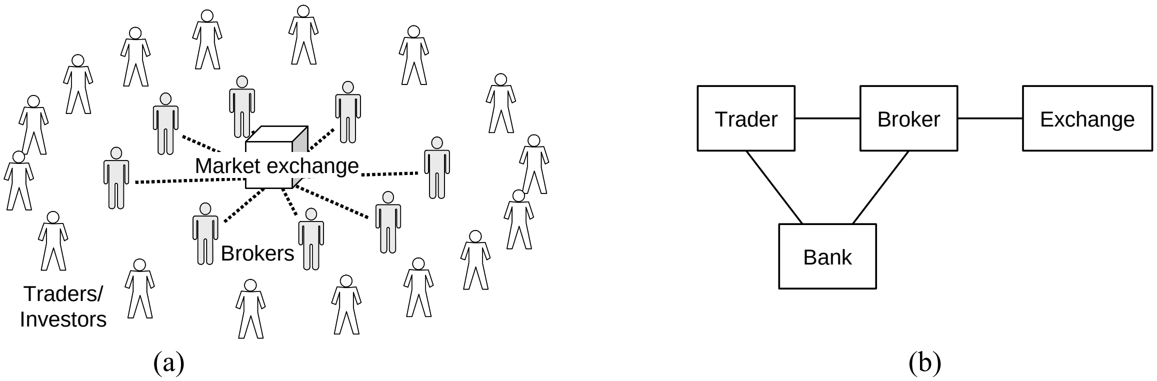


Figure 1-20: Structure of securities market. (a) Trading transactions can be executed only via brokers. (b) "Block diagram" of market interactions.

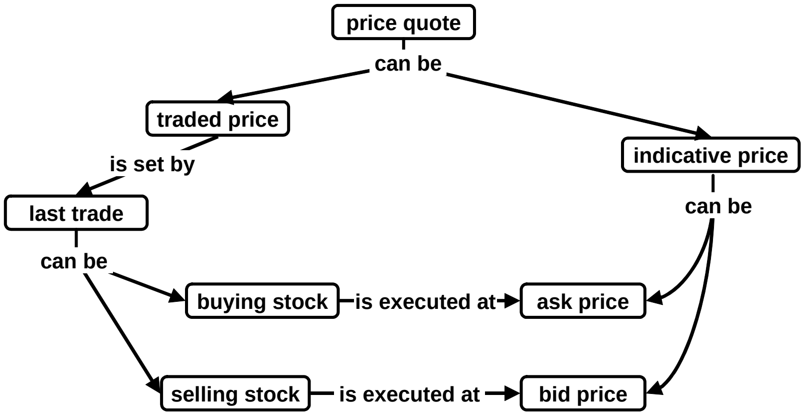


Figure 1-21: Concept map explaining how quoted stock prices are set.

Mechanics of Trading in Financial Markets

A market provides a forum where people always sell to the highest bidder. For a market to exist there must be a supply and demand side. As all markets, financial markets operate on a bid-offer basis: every stock has a quoted *bid* and a quoted *ask* (or offer). The concept map in Figure 1-21 summarizes the functioning of stock prices. The trader buys at the current ask and sells at the current bid. The bid is always lower than the ask. The difference between the bid and the ask is referred to as the *spread*. For example, assume there is a price quote of 100/105. That means the highest price someone is willing to pay to buy is 100 (bid), and the lowest price there is selling interest at 105 (offer or ask). Remember that there are volumes (number of shares) associated with each of those rates as well.

Using the bid side for the sake of illustration, assume that the buyer at 100 is willing to purchase 1,000 units. If someone comes in to sell 2,000 units, he would execute the first 1,000 at 100, the bid rate. That leaves 1,000 units still to be sold. The price the seller can get will depend on the depth of the market. It may be that there are other willing buyers at 100, enough to cover the reminder of the order. In an active (or liquid) market this is often the case.

What happens in a thin market, though? In such a situation, there may not be a willing buyer at 100. Let us assume a situation illustrated in the table below where the next best bid is by buyer B3 at 99 for 500 units. It is followed by B4 at 98 for 100 units, B1 for 300 units at 97, and B2 for 200 at 95. The trader looking to sell those 1,000 remaining units would have to fill part of the order at 99, more at 98, another bit at 97, and the last 100 at 95. In doing so, that one 2,000 unit trade lowered the bid down five points (because there would be 100 units left on the bid by B2). More than likely, the offer rate would move lower in a corresponding fashion.

Trader	Seller S1	Buyer B1	Buyer B2	Buyer B3	Buyer B4
Bid	market				
Ask		$97	$95	$99	$98
Num. of shares	1000	300	200	500	100

The above example is somewhat exaggerated but it illustrates the point. In markets with low volume it is possible for one or more large transactions to have significant impact on prices. This can happen around holidays and other vacation kinds of periods when fewer traders are active, and it can happen in the markets that are thinly traded (lack liquidity) in the first place.

When a trader wishes to arrange a trade, he places an *order*, which is a request for a trade yet to be executed. An **order** is an instruction to a broker/dealer to buy, sell, deliver, or receive securities that commits the issuer of the "order" to the terms specified. An **order ticket** is a form detailing the parameters of an Order instruction. Buy or sell orders differ in terms of the time limit, price limit, discretion of the broker handling the order, and nature of the stock-ownership position (explained below). Four types of orders are most common and frequently used:

1. **Market order:** An order from a trader to a broker to buy or sell a stock at the best available price. The broker should execute the order immediately, but may wait for a favorable price improvement. A market order to buy 10 shares of Google means buy the stock at whatever the lowest ask (offer) price is at the time the trade is executed. The broker could pay more (or less) than the price quoted to the trader, because in the meantime the market may have shifted (also recall the above example). Market orders are the quickest but not necessarily the optimal way to buy or sell a security.

2. **Limit order:** An order to buy or sell at a specific price, or better. The trader using a limit order specifies the maximum buy price or the minimum sale price at which the transaction shall be executed. That means when buying it would be at the limit price or below, while the reverse is true for a sell order. For example, a limit order to sell 100 Google shares at 600 means the trade will be executed at or above 600. A limit order can only be filled if the stock's market price reaches the limit price.

3. **Stop order:** (also referred to as a *stop-loss order*) A delayed market order to buy or sell a security when a certain price is reached or passed. A stop order is set at a point above (for a buy) or below (for a sell) the current price. When the current price reaches or passes through the specified level, the stop order is converted into an active market order (defined above in item 1). For example, a sell stop at 105 would be triggered if the market price touches or falls below 105. A buy stop order is entered at a stop price above the current market price. Investors generally use a buy stop order to limit a loss or to protect a profit on a stock that they have sold short. A sell stop order is entered at a stop

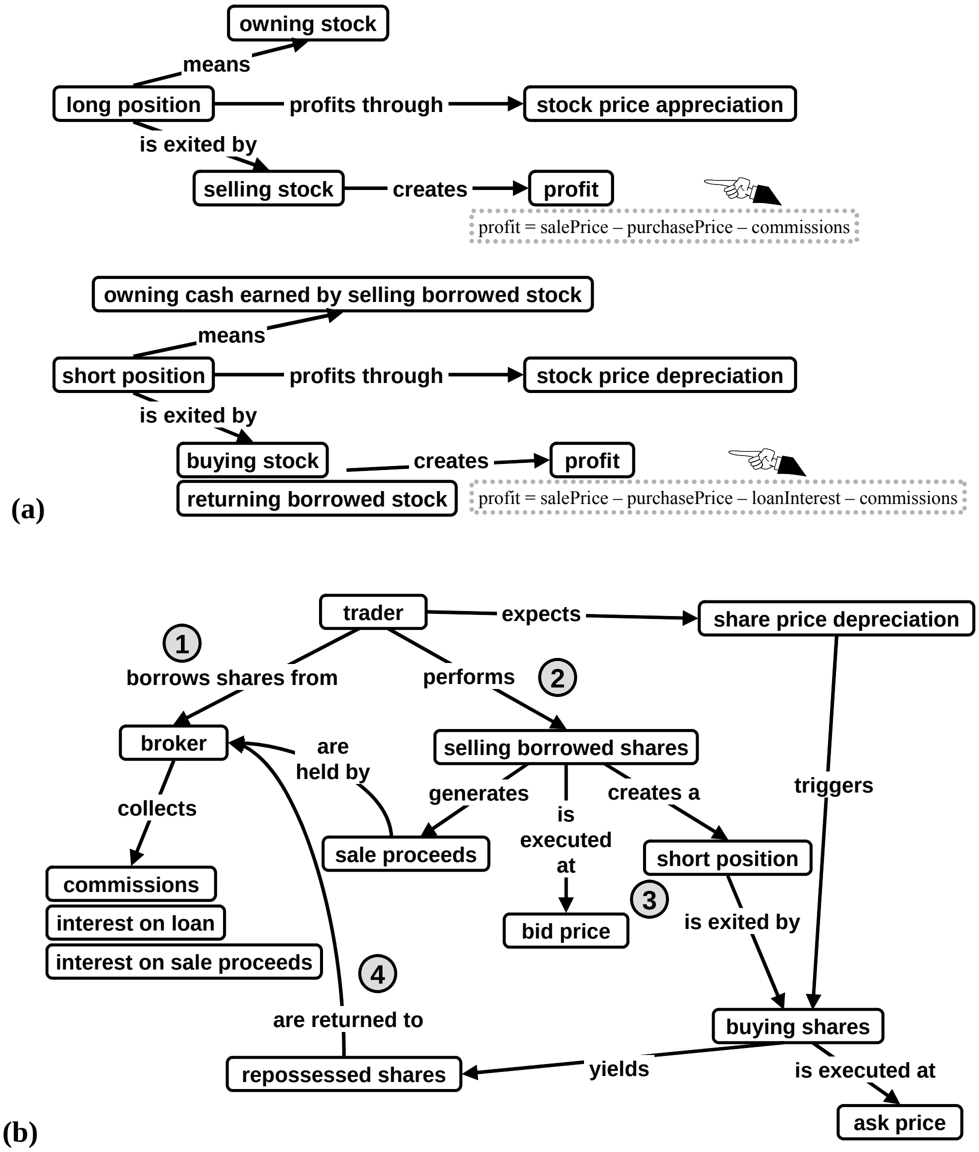


Figure 1-22: (a) Concept map of two types of stock-ownership positions: long and short. (b) Concept map explaining how short position functions.

price below the current market price. Investors generally use a sell stop order to limit a loss or to protect a profit on a stock that they own.

4. **Stop Limit Order:** A combination of the stop and limit orders. Unlike the simple stop order, which is converted into a market order when a certain price is reached, the stop limit order is converted into a limit order. Hence, the trader can control the price at which the order can be executed and will get a fill at or better than the limit order price.

For information on other, more advanced order types, the reader should search the Web. There are two types of security-ownership positions: long and short, see Figure 1-22(a). A *long position* represents actual ownership of the security regardless of whether personal funds, financial leverage (borrowed funds), or both are used in its purchase. Profits are realized if the price of the security increases.

A *short position* involves first a sale of the stock, followed by a purchase at, it is hoped, a lower price, Figure 1-22(b). The trader is "short" (does not own the stock) and begins by borrowing a stock from the investment broker, who ordinarily holds a substantial number of shares and/or has access to the desired stock from other investment brokers. The trader then sells the borrowed stock at the market price. The short position holder owes the shares to the broker; the short position can be covered by buying back the shares and returning the purchased shares to the broker to settle the loan of shares. This sequence of steps is labeled by numbers in Figure 1-22(b). The trader hopes that the stock price will drop and the difference between the sale price and the purchase price will result in a positive profit.

One can argue that there is no such thing as a "bad market," there is only the wrong position in the market. If the trader believes that a particular stock will move upwards, he should establish a long position. Conversely, if he believes that the stock will slide, he should establish a short position[6]. The trader can also hedge his bets by holding simultaneously both long and short positions on the same stock.

Computerized Support for Individual Investor Trading

We need to consider several choices and constraints for the system-to-be. First, we need to decide whether the system-to-be will provide brokerage services, or will just provide trading advice. Online brokerage firms already offer front-end systems for traders, so it will be difficult to insert our system-to-be between a trader and a broker. Offering our system-to-be as tool for on-a-side analysis (out of the trading loop) would have limited appeal. The other option is to include brokerage services, which will introduce significant complexity into the system. An important constraint on applicability of our system is that real-time price quotations currently are not available for free. We choose to consider both options in this book. The first five chapters will consider a case study of a system that includes a trader/broker services. Chapter 8 on Web services will consider stock analyst services. Both versions are described in Section 1.5.

Knowing how to place a trading order does not qualify one as a trader. It would be equivalent of saying that one knows how to drive a car just after learning how to use the steering wheel or the brake. There is much more to driving a car than just using the steering wheel or the brake. Similarly, there is much more to trading than just executing trades. To continue with the analogy, we need to have a "road map," a "travel plan," and we also need to know how to read the "road signs," and so on.

In general, the trader would care to know if a trading opportunity arose and, once he places a trading order, to track the status of the order. The help of computer technology has always been sought by traders for number crunching and scenario analysis. The basic desire is to be able to tell the future based on the knowledge of the past. Some financial economists view price movements on stock markets as a purely "random walk," and believe that the past prices cannot tell us anything useful about future behavior of the price. Others, citing chaos theory, believe that useful

[6] This is the idea of the so called *inverse funds*, see more here: B. Steverman: "Shorting for the 21st century: Inverse funds allow investors to place bets on predictions of a drop in stocks," *Business Week*, no. 4065, p. 78, December 31, 2007.

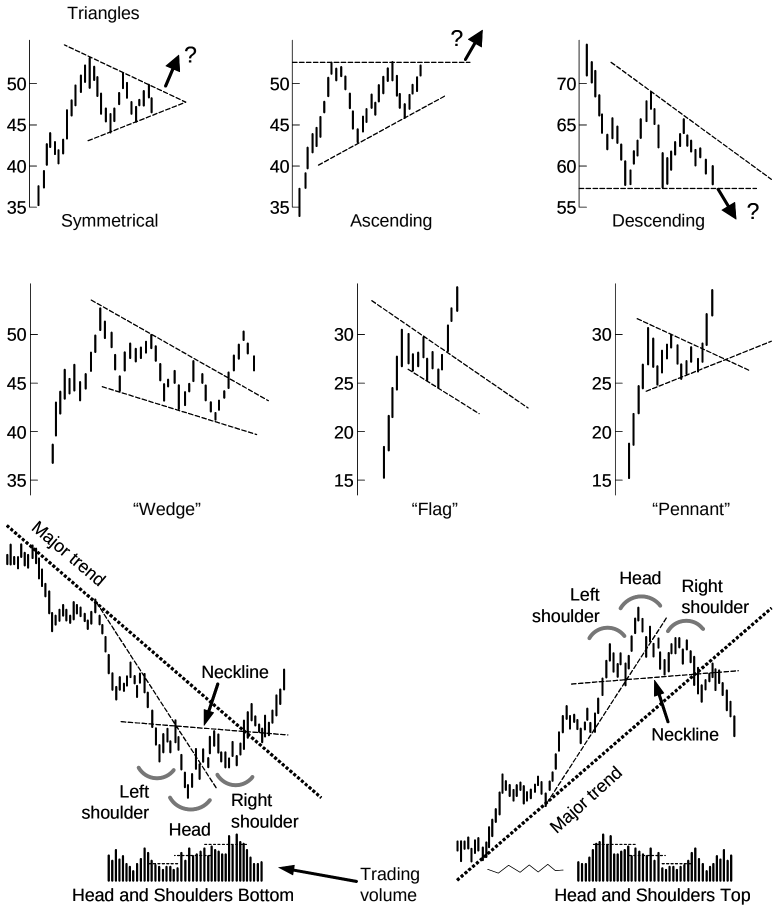


Figure 1-23: Technical analysis of stock price trends: Some example types of trend patterns. In all charts the horizontal axis represents time and the vertical axis stock price range. Each vertical bar portrays the high and low prices of a particular stock for a chosen time unit. Source: Alan R. Shaw, "Market timing and technical analysis," in Sumner N. Levine (Editor), *The Financial Analyst's Handbook, Second Edition*, pp. 312-372, Dow Jones-Irwin, Inc., Homewood, IL, 1988.

regularities can be observed and exploited. Chaos theory states that seemingly random processes may in fact have been generated by a deterministic function that is not random [Bao, et al., 2004].

Bao, Yukun, Yansheng Lu, Jinlong Zhang. "Forecasting stock prices by SVMs regression," Artificial Intelligence: Methodology, Systems, and Applications, vol. 3192, 2004.

A simple approach is to observe prices $price_i(t)$ of a given stock i over a window of time $t_{current} - Window$, …, $t_{current} - 2$, $t_{current} - 1$, $t_{current}$. We could fit a regression line through the observed points and devise a rule that a positive line slope represents a buying opportunity, negative slope a need to sell, and zero slope calls for no action. Obviously, it is not most profitable to buy when the stock already is gaining nor it is to sell when the stock is already sliding. The worst-case scenario is to buy at a market top or to sell when markets hit bottom. Ideally, we would like to detect the turning points and buy when the price is just about to start rising or sell when the price is just about to start falling. Detecting an ongoing trend is relatively easy; detecting an imminent onset of a new trend is difficult but most desirable.

This is where *technical analysis* comes into picture. Technical analysts believe that market prices exhibit identifiable regularities (or patterns or indicators) that are bound to be repeated. Using technical analysis, various trends could be "unearthed" from the historical prices of a particular stock and potentially those could be "projected into future" to have some estimation around where that stock price is heading. Technical analysts believe that graphs give them the ability to form an opinion about any security without following it in real time. They have come up with many types of indicators that can be observed in stock-price time series and various interpretations of the meaning of those indicators. Some chart formations are shown in Figure 1-23. For example, the triangles and flags represent consolidations or corrective moves in market trends. A *flag* is a well-defined movement contrary to the main trend. The *head-and-shoulder* formations are used as indicators of trend reversals and can be either top or bottom. In the "bottom" case, for example, a major market low is flanked on both sides (shoulders) by two higher lows. Cutting across the shoulders is some resistance level called the neckline. (*Resistance* represents price levels beyond which the market has failed to advance.) It is important to observe the trading volume to confirm the price movements. The increasing volume, as you progress through the pattern from left to right, tells you that more and more traders see the shifting improvement in the company's fortunes. A "breakout" (a price advance) in this situation signals the end of a downtrend and a new direction in the price trend. Technical analysts usually provide behavioral explanations for the price action and formation of trends and patterns.

However, one may wonder if just looking at a sequence of price numbers can tell us everything we need to know about the viability of an investment?! Should we not look for actual causes of price movements? Is the company in bad financial shape? Unable to keep up with competition? Or, is it growing rapidly? There is ample material available to the investor, both, in electronic and in print media, for doing a sound research before making the investment decision. This kind of research is called *fundamental analysis*, which includes analysis of important characteristics of the company under review, such as:

1. Market share: What is the market standing of the company under review? How much share of the market does it hold? How does that compare against the competitors?

2. Innovations: How is the company fairing in terms of innovations? For example in 3M company no less than 25% of the revenues come from the innovative products of last 5 years. There is even an index for innovations available for review and comparison (8th Jan'2007 issue of Business Week could be referred to).

3. Productivity: This relates the input of all the major factors of production – money, materials and people to the (inflation adjusted) value of total output of goods and services from the outside

4. Liquidity and Cash-flow: A company can run without profits for long years provided it has enough cash flows, but hardly the reverse is true. A company, if it has a profitable unit, but not enough cash flows, ends of "putting that on sale" or "spinning that unit out."

In addition to the above indicators, number crunching is also a useful way to fine-tune the decision. Various financial numbers are readily available online, such as

- Sales

- EPS: Earning per Share

- P/E – ttm: Trailing 12 months' ratio of Price per Share to that of Earning per Share

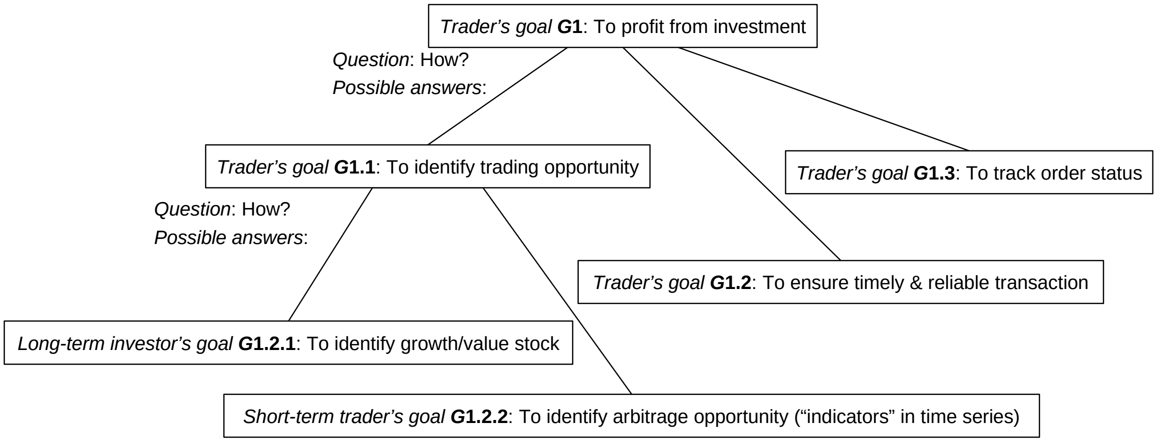

Trader's goal ***G*1**: To profit from investment
Question: How?
Possible answers:
Trader's goal ***G*1.1**: To identify trading opportunity
Trader's goal ***G*1.3**: To track order status
Question: How?
Possible answers:
Trader's goal ***G*1.2**: To ensure timely & reliable transaction
Long-term investor's goal ***G*1.2.1**: To identify growth/value stock
Short-term trader's goal ***G*1.2.2**: To identify arbitrage opportunity ("indicators" in time series)

Figure 1-24: Example of refining the representation of user's goals.

- P/E – forward: Ratio of Estimated Price per Share for coming 12 months to that of Estimated Earning of coming 12 months

- ROI: Return on Investment

The key barometer of stock market volatility is the Chicago Board Options Exchange's Volatility Index, or VIX, which measures the fluctuations of options contracts based on the S&P 100-stock index.

In fact, one could argue that the single most important decision an investor can make is to get out of the way of a collapsing market[7].

Where the investor is usually found to be handicapped is when she enters into the market with the objective of short term gains. The stock market, with its inherent volatility offers ample opportunities to exploit that volatility but what the investor lacks is an appropriate tool to assist in this "decision-making" process.

The investor would ideally like to "enter" the market after it is open and would "exit" the market before it is closed, by the end of that day. The investor would seek a particular stock, the price of which she is convinced would rise by the end of the day, would buy it at a "lower" price and would sell it at a higher price. If she gets inkling, somehow, that a particular stock is going to go up, it will be far easier for her to invest in that stock. Usually time is of essence here and this is where *technical analysis* comes into picture.

Again, we must clearly state what the user needs: the user's goals. It is not very helpful to state that the user's goal is "to make money." We must be as specific as possible, which can be achieved by keeping asking questions "How?" An example of goal refinement is shown in Figure 1-24. Note that in answering how to identify a trading opportunity, we also need to know whether our trader has a short-term or long-term outlook to investment. In addition, different trader types may compose differently the same sub-goals (low-level goals) into high-level goals. For example, the long-term investor would primarily consider the company's prospects (*G*1.2.1), but may employ time-series indicators (*G*1.2.2) to decide the timing of their investments. Just because one

[7] Michael Mandel, "Bubble, bubble, who's in trouble?" *Business Week*, p. 34, June 26, 2006.

anticipates that an investment will be held for several years because of its underlying fundamentals, that does not mean that he should overlook the opportunity for buying at a lower price (near the bottom of an uptrend).

It is important to understand the larger context of the problem that we are trying to solve. There are already many people who are trying to forecast financial markets. Companies and governments spend vast quantities of resources attempting to predict financial markets. We have to be realistic of what we can achieve with relatively minuscule resources and time period of one academic semester.

From the universe of possible market data, we have access only to a subset, which is both due to economic (real-time data are available with paid subscription only) and computational (gathering and processing large data quantities requires great computing power) reasons. Assuming we will use freely available data and a modest computing power, the resulting data subset is suitable only for certain purposes. By implication, this limits our target customer and what he can do with the software-to-be.

In conclusion, our planned tool is not for a "professional trader." This tool is not for institutional investor or large brokerage/financial firm. This tool is for an ordinary single investor who does not have acumen of financial concepts, yet would like to trade smartly. This tool is for an investor who does not have too much time to do a thorough research on all aspects of a particular company, neither does he have understanding and mastery over financial number crunching. It is unlikely to be used for "frequency trading," because we lack computing power and domain knowledge needed for such sophisticated uses.

1.4 The Object Model

"You cannot teach beginners top-down programming, because they don't know which end is up."
—C.A.R. Hoare

An **object** is a software packaging of data and code together into a unit within a running computer program. Objects can interact by calling other objects for their services. In Figure 1-25, object `Stu` calls the object `Elmer` to find out if 905 and 1988 are coprimes. Two integers are said to be *coprime* or *relatively prime* if they have no common factor other than 1 or, equivalently, if their greatest common divisor is 1. `Elmer` performs computation and answers positively. Objects do not accept arbitrary calls. Instead, acceptable calls are defined as a set of object "methods." This fact is indicated by the method `areCoprimes()` in Figure 1-25. A **method** is a function (also known as operation, procedure, or subroutine) associated with an object so that other objects can call on its services. Every software object supports a limited number of methods. Example methods for an ATM machine object are illustrated in Figure 1-26. The set of methods along with the exact format for calling each method (known as the method "signature") represents the object's **interface** (Figure 1-27). The interface specifies object's *behavior*—what kind of calls it accepts and what it does in response to each call.

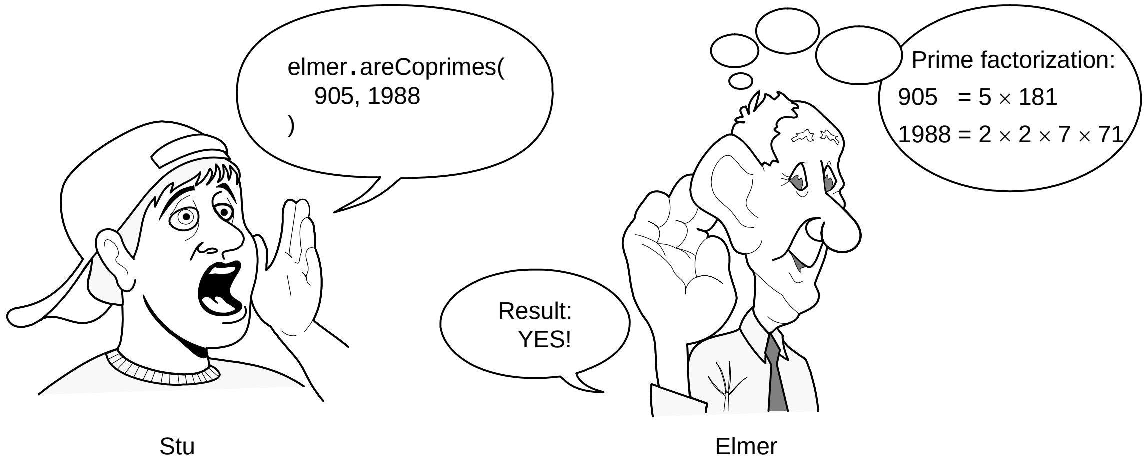


Figure 1-25: Client object sends a message to a server object by invoking a method on it. Server object is the method receiver.

Software objects work together to carry out the tasks required by the program's business logic. In object-oriented terminology, objects communicate with each other by sending **messages**. In the world of software objects, when an object *A* calls a method on an object *B* we say, "*A* sends a message to *B*." In other words, a *client* object requests the execution of a method from a *server* object by sending it a message. The message is matched up with a method defined by the software class to which the receiving object belongs. Objects can alternate between a client role and a server role. An object is in a client role when it is the originator of an object invocation, no matter whether the objects are located in the same memory space or on different computers. Most objects play both client and server roles.

In addition to methods, software objects have attributes or properties. An **attribute** is an item of data named by an identifier that represents some information about the object. For example, a person's attribute is the age, or height, or weight. The attributes contain the information that differentiates between the various objects. The currently assigned *values* for object attributes describe the object's internal **state** or its current condition of existence. Everything that a software object knows (state) and can do (behavior) is expressed by the attributes and the methods within that object. A **class** is a collection of objects that share the same set of attributes and methods (i.e., the interface). Think of a class as a template or blueprint from which objects are made. When an instance object is created, we say that the objects are *instantiated*. Each instance object has a distinct identity and its own copy of attributes and methods. Because objects are created from classes, you must design a class and write its program code before you can create an object.

Objects also have special methods called **constructors**, which are called at the creation of an object to "construct" the values of object's data members (attributes). A constructor prepares the new object for use, often accepting parameters which the constructor uses to set the attributes. Unlike other methods, a constructor never has a return value. A constructor should put an object in its initial, valid, safe state, by initializing the attributes with meaningful values. Calling a constructor is different from calling other methods because the caller needs to know what values are appropriate to pass as parameters for initialization.

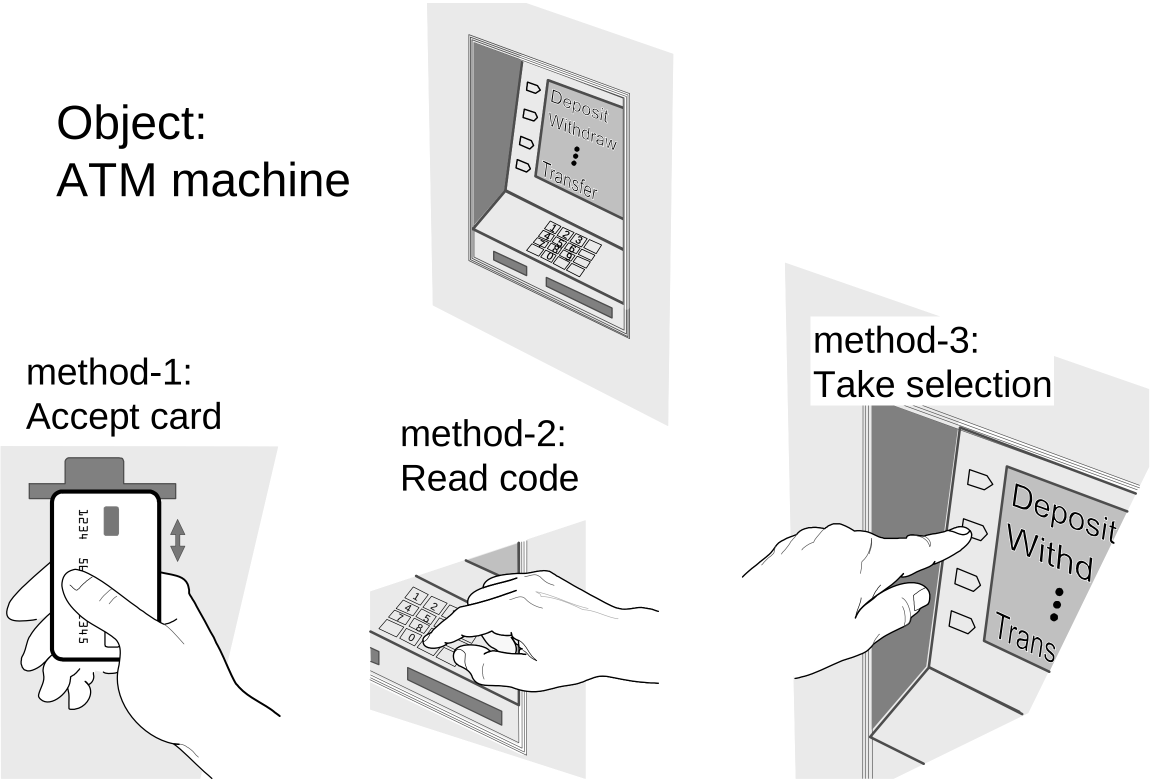


Figure 1-26: Acceptable calls are defined by object "methods," as shown here by example methods for an ATM machine object.

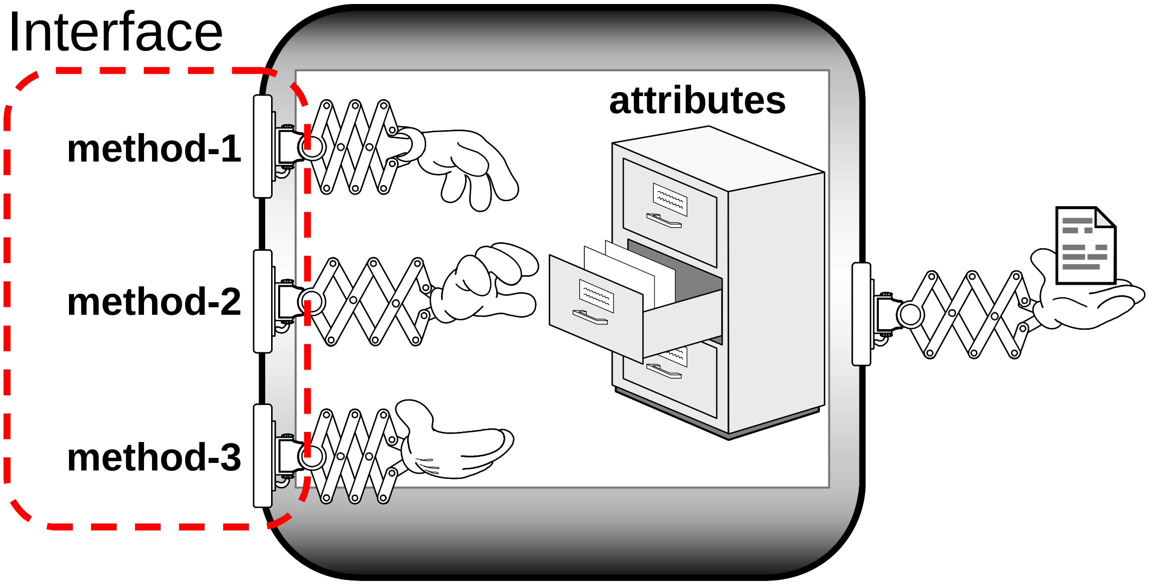


Figure 1-27: Software object interface is a set of object's methods with the format for calling each method.

Traditional approach to program development, known as ***procedural approach***, is process oriented in that the solution is represented as a *sequence of steps* to be followed when the program is executed. The processor receives certain input data and first does this, then that, and so on, until the result is outputted. The ***object-oriented approach*** starts by breaking up the whole program into software objects with specialized roles and creating a *division of labor*. Object-oriented programming then, is describing what messages get exchanged between the objects in the system. This contrast is illustrated on the safe home access system case study (Section 1.3.1).

Example 1.1 Procedural approach versus Object-oriented approach

The process-based or procedural approach represents solution as a *sequence of steps* to be followed when the program is executed, Figure 1-28(a). It is a *global* view of the problem as seen by the single agent advancing in a stepwise fashion towards the solution. The step-by-step approach is easier to understand when the whole problem is relatively simple and there are few alternative choices along the path. The problem with this approach is when the number of steps and alternatives becomes overwhelming.

Object-oriented (OO) approach adopts a *local* view of the problem. Each object specializes only in a relatively small subproblem and performs its task upon receiving a message from another object, Figure 1-28(b). Unlike a single agent travelling over the entire process, we can think of OO approach as organizing many tiny agents into a "bucket brigade," each carrying its task when called upon, Figure 1-28(c). When an object completes its task, it sends a message to another object saying "that does it for me; over to you—here's what I did; now it's your turn!" Here are pseudo-Java code snippets for two objects, KeyChecker and LockCtrl:

Listing 1-1: Object-oriented code for classes KeyChecker (left) and LockCtrl (right).

```
public class KeyChecker {
    protected LockCtrl lock_;
    protected java.util.Hashtable
        validKeys_;
    ...

 /** Constructor */
    public KeyChecker(
        LockCtrl lc, ...) {
        lock_ = lc;
        ...
    }

 /** This method waits for and
  * validates the user-supplied key
  */
    public keyEntered(
        String key
    ) {
        if (
         validKeys.containsKey(key)
        ) {
            lock_.unlock(id);
        }
    } else {
            // deny access
            // & sound alarm bell?
        }
```

```
public class LockCtrl {
    protected boolean
        locked_ = true; // start locked
    protected LightCtrl switch_;
    ...

 /** Constructor */
    public LockCtrl(
        LightCtrl sw, ...) {
        switch_ = sw;
        ...
    }

 /** This method sets the lock state
  * and hands over control to the switch
  */
    public unlock() {
        ... operate the physical lock device
        locked_ = false;
        switch_.turnOn();
    }

    public lock(boolean light) {
        ... operate the physical lock device
        locked_ = true;
        if (light) {
            switch_.turnOff();
        }
```

} }	} }

Two important observations:

Object roles/responsibilities are focused (each object is focused on one task, as its name says); later, we will see that there are more responsibilities, like calling other objects

Object's level of abstraction must be carefully chosen: here, we chose key checker and its method keyEntered(), instead of specifying the method of key entry (type in code vs. acquire biometric identifier), and LockCtrl does not specify how exactly the lock device functions. Too low level specifies such details (which could be specified in a derived class), or too high abstraction level just says control-the-access().

The key developer skill in object-oriented software development is *performing the division of labor* for software objects. Preferably, each object should have only one clearly defined task (or, responsibility) and that is relatively easy to achieve. The main difficulty in assigning responsibilities arises when an object needs to communicate with other objects in accomplishing a task.

When something goes wrong, you want to know where to look or whom to single out. This is particularly important for a complex system, with many functions and interactions. Object-oriented approach is helpful because the responsibilities tend to be known. However, the responsibilities must be assigned adequately in the first place. That is why assigning responsibilities to software objects is probably the most important skill in software development. Some responsibilities are obvious. For example, in Figure 1-28 it is natural to assign the control of the light switch to the LightCtrl object.

However, assigning the responsibility of communicating messages is harder. For example, who should send the message to the LightCtrl object to turn the switch on? In Figure 1-28, LockCtrl is charged with this responsibility. Another logical choice is KeyChecker, perhaps even more suitable, because it is the KeyChecker who ascertains the validity of a key and knows whether or not unlocking and lighting actions should be initiated. More details about assigning responsibilities are presented in Section 2.6.

The concept of objects allows us to divide software into smaller pieces to make it manageable. The divide-and-conquer approach goes under different names: reductionism, modularity, and structuralism. The "object orientation" is along the lines of the reductionism paradigm: "the

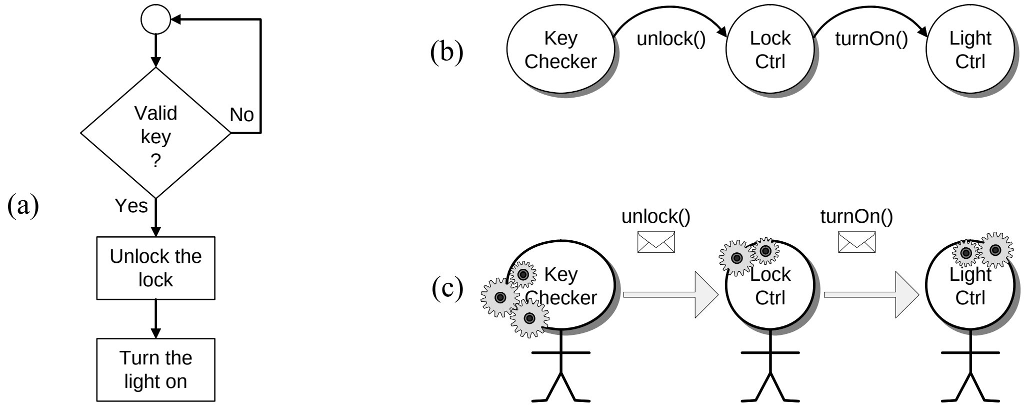


Figure 1-28: Comparison of process-oriented (procedural) and object-oriented methods on the safe home access case study. (a) A flowchart for a procedural solution; (b) An object-oriented solution. (c) An object can be thought of as a person with expertise and responsibilities.

tendency to or principle of analysing complex things into simple constituents; the view that a system can be fully understood in terms of its isolated parts, or an idea in terms of simple concepts" [Concise Oxford Dictionary, 8th Ed., 1991]. If your car does not work, the mechanic looks for a problem in one of the parts—a dead battery, a broken fan belt, or a damaged fuel pump. A design is **modular** when each activity of the system is performed by exactly one unit, and when inputs and outputs of each unit are well defined. Reductionism is the idea that the best way to understand any complicated thing is to investigate the nature and workings of each of its parts. This approach is how humans solve problems, and it comprises the very basis of science.

SIDEBAR 1.1: Object Orientation

♦ Object orientation is a worldview that emerged in response to real-world problems faced by software developers. Although it has had many successes and is achieving wide adoption, as with any other worldview, you may question its soundness in the changing landscape of software development. OO stipulates that data and processing be packaged together, data being *encapsulated* and unreachable for external manipulation other than through object's methods. It may be likened to disposable cameras where film roll (data) is encapsulated within the camera mechanics (processing), or early digital gadgets with a built-in memory. People have not really liked this model, and most devices now come with a replaceable memory card. This would speak against the data hiding and for separation of data and processing. As we will see in Chapter 8, web services are challenging the object-oriented worldview in this sense.

There are three important aspects of object orientation that will be covered next:

- Controlling access to object elements, known as encapsulation
- Object responsibilities and relationships
- Reuse and extension by inheritance and composition

1.4.1 Controlling Access to Object Elements

Modular software design provides means for breaking software into meaningful components, Figure 1-29. However, modules are only loose groupings of subprograms and data. Because there is no strict ownership of data, subprograms can infringe on each other's data and make it difficult to track who did what and when. Object oriented approach goes a step further by emphasizing state *encapsulation*, which means *hiding* the object state, so that it can be observed or modified only via object's methods. This approach enables better control over interactions among the modules of an application. Traditional software modules, unlike software objects, are more "porous;" encapsulation helps prevent "leaking" of the object state and responsibilities.

In object-orientation, object data are more than just program data—they are object's *attributes*, representing its individual characteristics or properties. When we design a class, we decide what internal state it has and how that state is to appear on the outside (to other objects). The internal state is held in the attributes, also known as class *instance variables*. UML notation for software class is shown in Figure 1-30. Many programming languages allow making the internal state directly accessible through a variable manipulation, which is a bad practice. Instead, the access to object's data should be controlled. The external state should be exposed through method calls,

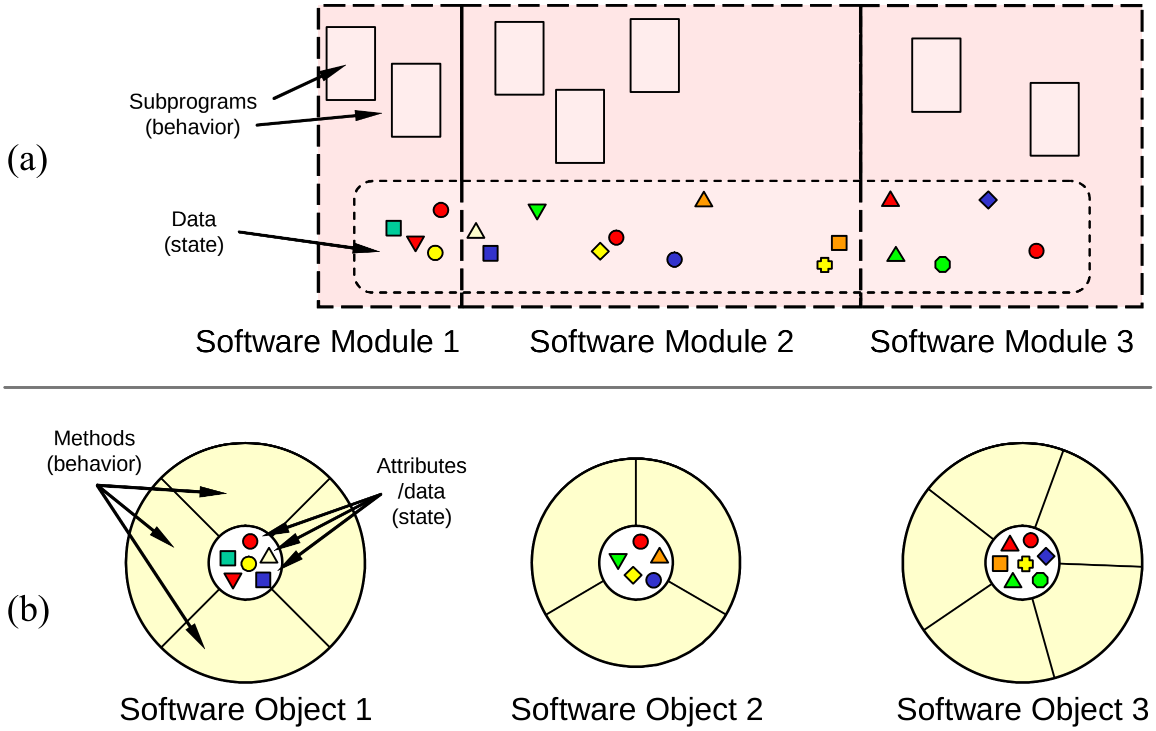


Figure 1-29: Software modules (a) vs. software objects (b).

called *getters* and *setters*, to get or set the instance variables. Getters and setters are sometimes called *accessor* and *mutator* methods, respectively. For example, for the class LightController in Figure 1-31 the getter and setter methods for the attribute `lightIntensity` are `getLightIntensity()` and `setLightIntensity()`, respectively. Getter and setter methods are considered part of object's *interface*. In this way, the interface exposes object's behavior, as well as its attributes via getters and setters.

Access to object attributes and methods is controlled using **access designations**, also known as *visibility* of attributes and methods. When an object attribute or method is defined as `public`, other objects can directly access it. When an attribute or method is defined as `private`, only that specific object can access it (not even the descendant objects that inherit from this class). Another access modifier, `protected`, allows access by related objects, as described in the next section. The UML symbols for access designations in class diagrams are as follows (Figure 1-30): **+** for `public`, global visibility; **#** for `protected` visibility; and, **−** for `private` within-the-class-only visibility.

We separate object design into three parts: its public *interface*, the terms and conditions of use (*contracts*), and the private details of how it conducts its business (known as *implementation*).

The services presented to a client object comprise the **interface**. The interface is the fundamental means of communication between objects. Any behavior that an object provides must be invoked by a message sent using one of the provided interface methods. The interface should precisely describe how client objects of the class interact with the class. Only the methods that are designated as `public` comprise the class interface ("+" symbol in UML class diagrams). For example, in Figure 1-31 the class HouseholdDeviceController has three public methods that constitute its interface. The private method `sendCommandToUSBport()` is not part of the

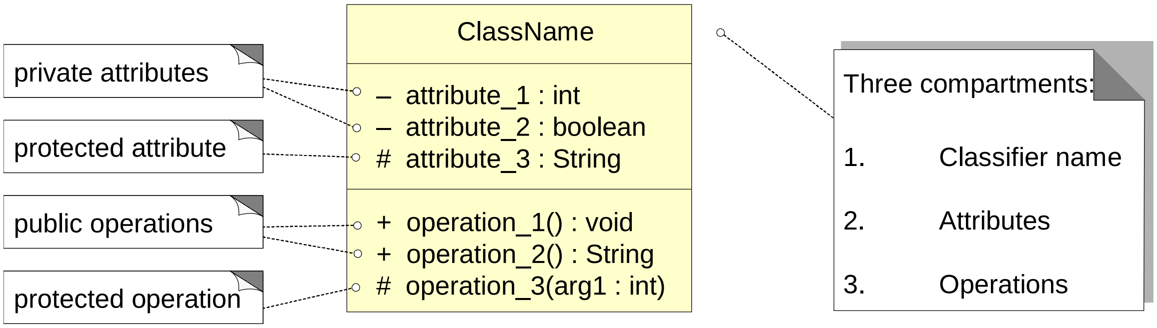
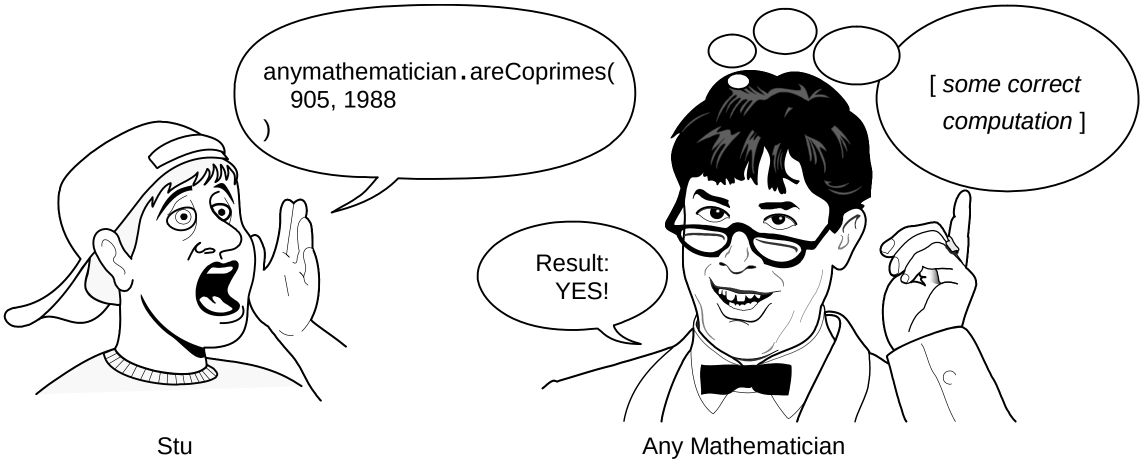


Figure 1-30: UML notation for software class.

interface. Note that interfaces do not normally include attributes—only methods. If a client needs to access an attribute, it should use the getter and setter methods.

Encapsulation is fundamental to object orientation. Encapsulation is the process of packaging your program, dividing its classes into the public interface and the private implementation. The basic question is, what in a class (which elements) should be exposed and what should be hidden. This question pertains equally to attributes and behavior. (Recall that attributes should never be exposed directly, but instead by using getter and setter methods.) Encapsulation hides everything that is not necessary for other classes to know about. By localizing attributes and behaviors and preventing logically unconnected functions from manipulating object elements, we ensure that a change in a class will not cause a rippling effect around the system. This property makes for easier maintaining, testing, and extending the classes.

Object orientation continues with the black-box approach of focusing on interface. In Section 1.2.2, the whole system was considered as a black box, and here we focus on the micro-level of individual objects. When specifying an interface, we are only interested in *what* an object does, not *how* it does it. The "how" part is considered in implementation. Class **implementation** is the program code that specifies how the class conducts its business, i.e., performs the computation. Normally, the client object does not care how the computation is performed as long as it produces the correct answer. Thus, the implementation can change and it will not affect the client's code. For example, in Figure 1-25, object `Stu` does not care that the object `Elmer` answers if numbers are coprimes. Instead, it may use any other object that provides the method `areCoprimes()` as part of its interface.



Contracts can specify different terms and conditions of object. Contract may apply at design time or at run time. Programming languages such as Java and C# have two language constructs for specifying design-time contracts.

Run time contracts specify the conditions under which an object methods can be called upon (conditions-of-use guarantees), and what outcome methods achieve when they are finished (aftereffect guarantees).

It must be stressed that the interchangeable objects must be identical in every way—as far as the client object's perceptions go.

1.4.2 Object Responsibilities and Relationships

The key characteristic of object-orientation is the concept of *responsibility* that an object has towards other objects. Careful assignment of responsibilities to objects makes possible the division of labor, so that each object is focused on its specialty. Other characteristics of object orientation, such as polymorphism, encapsulation, etc., are characteristics local to the object itself. Responsibilities characterize the *whole system* design. To understand how, you need to read Chapters 2, 4, and 5. Because objects work together, as with any organization you would expect that the entities have defined roles and responsibilities. The process of determining what the object should know (state) and what it should do (behavior) is known as *assigning the responsibilities*. What are object's responsibilities? The key object responsibilities are:

1. Knowing something (*memorization* of data or object attributes)
2. Doing something on its own (*computation* programmed in a "method")
3. Calling methods of other objects (*communication* by sending messages)

We will additionally distinguish a special type of doing/computation responsibilities:

2.a) Business rules for implementing business policies and procedures

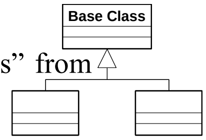
Business rules are important to distinguish because, unlike algorithms for data processing and calculating functions, they require knowledge of customer's business context and they often change. We will also distinguish communication responsibilities:

3.a) Calling constructor methods; this is special because the caller must know the appropriate parameters for initialization of the new object.

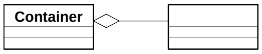
Assigning responsibilities essentially means deciding what methods an object gets and who invokes those methods. Large part of this book deals with assigning object responsibilities, particularly Section 2.6 and Chapter 5.

The basic types of class relationships are *inheritance*, where a class inherits elements of a base class, and *composition*, where a class contains a reference to another class. These relationships can be further refined as:

- *Is-a* relationship (hollow triangle symbol Δ in UML diagrams): A class "inherits" from another class, known as base class, or parent class, or superclass



- *Has-a* relationship: A class "contains" another class



  - *Composition* relationship (filled diamond symbol ♦ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk
  - *Aggregation* relationship (hollow diamond symbol ◊): The contained item is an element of a collection but it can also exist on its own, such as a desk in an office

- *Uses-a* relationship (arrow symbol ↓ in UML diagrams): A class "uses" another class
- *Creates* relationship: A class "creates" another class (calls a constructor method)

Has-a and Uses-a relationships can be seen as types of composition.

1.4.3 Reuse and Extension by Inheritance and Composition

One of the most powerful characteristics of object-orientation is code reuse. Procedural programming provides code reuse to a certain degree—you can write a procedure and then reuse it many times. However, object-oriented programming goes an important step further, allowing you to define relationships between classes that facilitate not only code reuse, but also better overall design, by organizing classes and factoring in commonalities of various classes.

Two important types of relationships in the object model enable reuse and extension: *inheritance* and *composition*. Inheritance relations are static—they are defined at the compile time and cannot change for the object's lifetime. Composition is dynamic, it is defined at run time, during the participating objects' lifetimes, and it can change.

When a message is sent to an object, the object must have a method defined to respond to that message. The object may have its own method defined as part of its interface, or it may inherit a method from its parent class. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, because each subclass is a separate entity, each might require a separate response to the same message. For example, in Figure 1-31 subclasses Lock Controller and Light Controller inherit the three public methods that constitute the interface of the superclass Household Device Controller. The private method is private to the superclass and not available to the derived subclasses. Light Controller *overrides* the method `activate()` that it inherits from its superclass, because it needs to adjust the light intensity after turning on the light. The method `deactivate()` is adopted unmodified. On the other hand, Lock Controller overrides both methods `activate()` and `deactivate()` because it requires additional behavior. For example, in addition to disarming the lock, Lock Controller's method `deactivate()` needs to start the timer that counts down how long time the lock has remained unlocked, so it can be automatically locked. The method `activate()` needs to clear the timer, in addition to arming the lock. This property that the same method behaves differently on different subclasses of the same class is called **polymorphism**.

Inheritance applies if several objects have some responsibilities in common. The key idea is to place the generic algorithms in a *base class* and inherit them into different detailed contexts of *derived classes*. With inheritance, we can program by difference. Inheritance is a strong relationship, in that the derivatives are inextricably bound to their base classes. Methods from the base class can be used only in its own hierarchy and cannot be reused in other hierarchies.

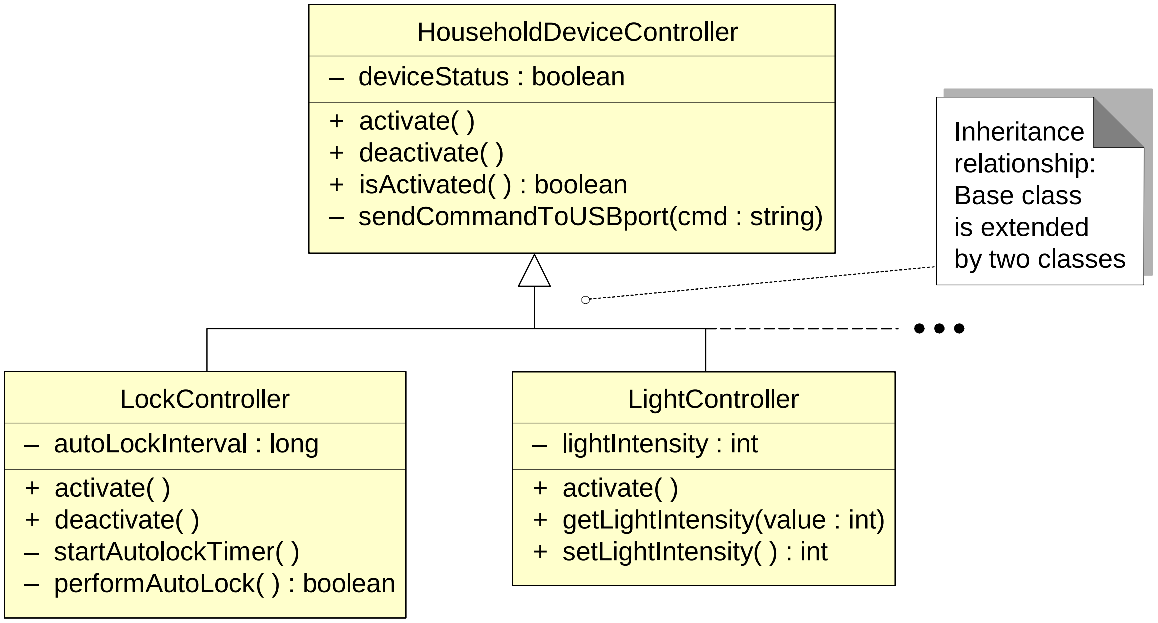

HouseholdDeviceController
– deviceStatus : boolean
+ activate( )
+ deactivate( )
+ isActivated( ) : boolean
– sendCommandToUSBport(cmd : string)
Inheritance relationship: Base class is extended by two classes
LockController
– autoLockInterval : long
+ activate( )
+ deactivate( )
– startAutolockTimer( )
– performAutoLock( ) : boolean
LightController
– lightIntensity : int
+ activate( )
+ getLightIntensity(value : int)
+ setLightIntensity( ) : int
• • •

Figure 1-31: Example of object inheritance.

1.5 Student Team Projects

> "Knowledge must come through action; you can have no test which is not fanciful, save by trial."
> —Sophocles

> "I have been impressed with the urgency of doing. Knowing is not enough; we must apply.
> Being willing is not enough; we must do." —Leonardo da Vinci

The book website, given in Preface, describes several student team projects. These projects are selected so each can be accomplished by a team of undergraduate students in the course of one semester. At the same time, the basic version can be extended so to be suitable for graduate courses in software engineering and some of the projects can be extended even to graduate theses. Here I describe only two projects and more projects along with additional information about the projects described here is available at the book's website, given in Preface.

Each project requires the student to learn one or more technologies specific for that project. In addition, all student teams should obtain a UML diagramming tool.

1.5.1 Stock Market Investment Fantasy League

This project is fashioned after major sports fantasy leagues, but in the stock investment domain. You are to build a **website** which will allow investor players to make virtual investments in real-world stocks using fantasy money. The system and its context are illustrated in Figure 1-32. Each

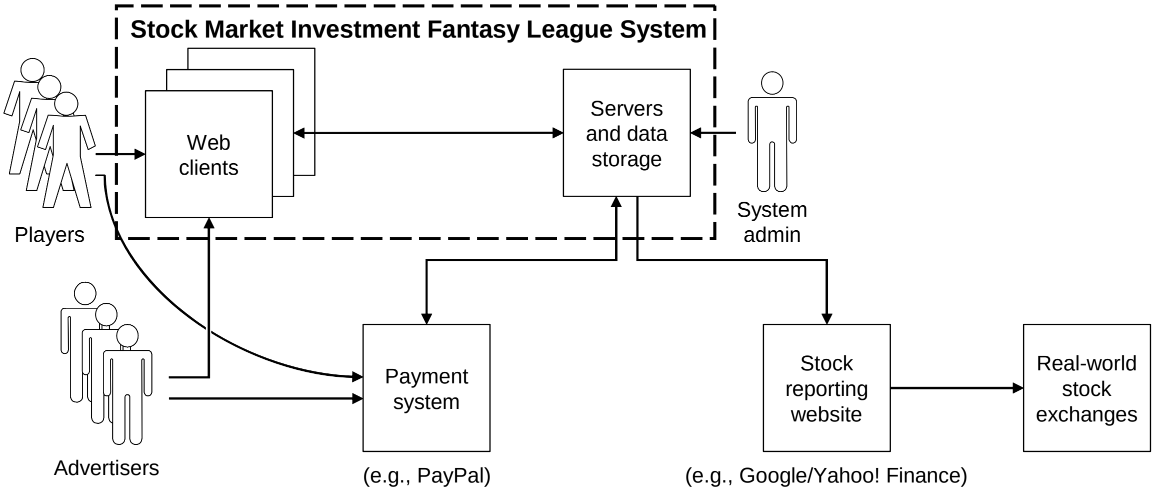


Figure 1-32: Stock market fantasy league system and the context within which it operates.

player has a personal account with fantasy money in it. Initially, the player is given a fixed amount of startup funds. The player uses these funds to virtually buy the stocks. The system then tracks the actual stock movement on real-world exchanges and periodically adjusts the value of players' investments. The actual stock prices are retrieved from a third-party source, such as Yahoo! Finance, that monitors stock exchanges and maintains up-to-date stock prices. Given a stock in a player's portfolio, if the corresponding actual stock loses value on a real-world stock exchange, the player's virtual investment loses value equally. Likewise, if the corresponding actual stock gains value, the player's virtual investment grows in the same way.

The player can sell the existing stocks or buy new ones at any time. This system does not provide any investment advice. When player sells a stock, his/her account is credited with fantasy money in the amount that corresponds to the current stock price on a stock exchange. A small commission fee is charged on all trading transactions (deducted from the player's account).

Your business model calls for advertisement revenues to support financially your website. Advertisers who wish to display their products on your website can sign-up at any time and create their account. They can upload/cancel advertisements, check balance due, and make payments (via a third party, e.g., a credit card company or PayPal.com). Every time a player navigates to a new window (within this website), the system randomly selects an advertisement and displays the advertisement banner in the window. At the same time, a small advertisement fee is charged on the advertiser's account. A more ambitious version of the system would fetch an advertisement dynamically from the advertiser's website, just prior to displaying it.

To motivate the players, we consider two mechanisms. One is to remunerate the best players, to increase the incentive to win. For example, once a month you will award 10 % of advertisement profits to the player of the month. The remuneration is conducted via a third party, such as PayPal.com. In addition, the system may support learning by analyzing successful traders and extracting information about their trading strategies. The simplest service may be in the form stock buying recommendations: "players who bought this stock also bought these five others." More complex strategy analysis may be devised.

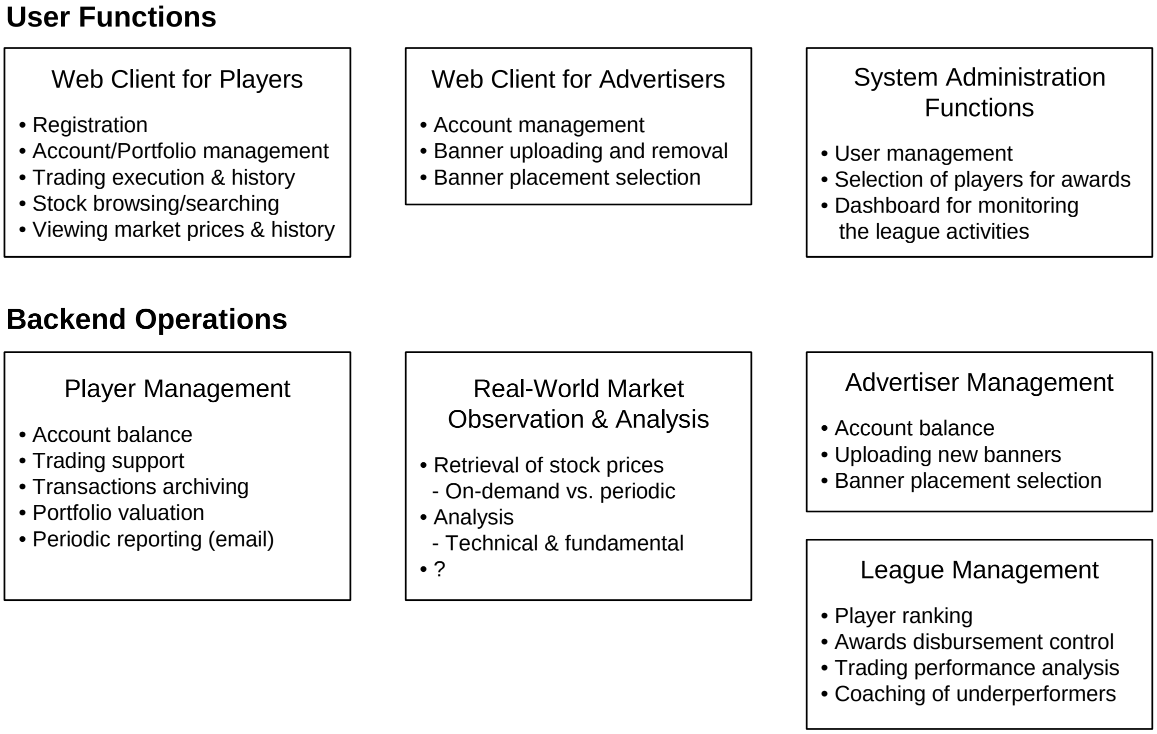

User Functions
Web Client for Players
• Registration
• Account/Portfolio management
• Trading execution & history
• Stock browsing/searching
• Viewing market prices & history
Web Client for Advertisers
• Account management
• Banner uploading and removal
• Banner placement selection
System Administration Functions
• User management
• Selection of players for awards
• Dashboard for monitoring the league activities
Backend Operations
Player Management
• Account balance
• Trading support
• Transactions archiving
• Portfolio valuation
• Periodic reporting (email)
Real-World Market Observation & Analysis
• Retrieval of stock prices
- On-demand vs. periodic
• Analysis
- Technical & fundamental
• ?
Advertiser Management
• Account balance
• Uploading new banners
• Banner placement selection
League Management
• Player ranking
• Awards disbursement control
• Trading performance analysis
• Coaching of underperformers

Figure 1-33: Logical grouping of required functions for Stock Market Fantasy League.

Statement of Requirements

Figure 1-33 shows logical grouping of functions requested from our system-to-be.

Player portfolio consists of *positions*—individual stocks owned by the player. Each position should include company name, ticker symbol, the number of shares owned by this player, and date and price when purchased. Player should be able to specify stocks to be tracked without owning any of those stocks. Player should also be able to specify buy- and sell thresholds for various stocks; the system should alert (via email) the player if the current price exceeds any of these thresholds.

Stock prices should be retrieved periodically to valuate the portfolios and at the moment when the user wishes to trade. Because price retrieval can be highly resource demanding, the developer should consider smart strategies for retrieval. For example, cache management strategies could be employed to prioritize the stocks based on the number of players that own it, the total number of shares owned, etc.

Additional Information

I would strongly encourage the reader to look at Section 1.3.2 for an overview of financial investment. Additional information about this project can be found at the book website, given in Preface.

http://finance.yahoo.com/

http://www.marketwatch.com/

See also Problem 2.29 and Problem 2.32 at the end of Chapter 2, the solutions of which can be found at the back of the text.

1.5.2 Web-based Stock Forecasters

> "Business prophets tell what is going to happen, business profits tell what has happened." —Anonymous

There are many tools available to investors but none of them removes entirely the element of chance from investment decisions. Large trading organizations can employ sophisticated computer systems and armies of analysts. Our goal is to help the individual investor make better investment decisions. Our system will use the Delphi method,[8] which is a systematic interactive forecasting method for obtaining consensus expectation from a panel of independent experts.

The goal of this project is to have multiple student teams implement Web services (Chapter 8) for stock-prediction. Each Web service (WS) will track different stocks and, when queried, issue a forecast about the price movement for a given stock. The client module acts as a "facilitator" which gathers information from multiple Web services ("independent experts") and combines their answers into a single recommendation. If different Web services offer conflicting answers, the client may repeat the process of querying and combining the answers until it converges towards the "correct" answer.

There are three aspects of this project that we need to decide on:

- What kind of information should be considered by each forecaster? (e.g., stock prices, trading volumes, fundamental indicators, general economic indicators, latest news, etc. Stock prices and trading volumes are fast-changing so must be sampled frequently and the fundamental and general-economy indicators are slow-moving so could be sampled at a low frequency.)
- Who is the target customer? Organization or individual, their time horizon (day trader vs. long-term investor)
- How the application will be architected? The user will run a client program which will poll the WS-forecasters and present their predictions. Should the client be entirely Web-based vs. locally-run application? A Web-based application would be downloaded over the Web every time the user runs the client; it could be developed using AJAX or a similar technology.

As a start, here are some suggested answers:

- Our target customers are individuals who are trading moderately frequently (up to several times per week), but not very frequently (several times per day).
- The following data should be gathered and stored locally. Given a list of about 50–100 companies, record their quoted prices and volumes at the maximum available sampling

[8] An introductory description is available here: http://en.wikipedia.org/wiki/Delphi_method . An in-depth review is available here: http://web.njit.edu/~turoff/Papers/delphi3.html (M. Turoff and S. R. Hiltz: "Computer Based Delphi Processes," in M. Adler and E. Ziglio (Editors), *Gazing Into the Oracle: The Delphi Method and Its Application to Social Policy and Public Health*, London, UK: Kingsley Publishers, 1995.)

density (check http://finance.yahoo.com/); also record some broad market indices, such as DJIA or S&P500.

- The gathered data should be used for developing the *prediction model*, which can be a simple regression-curve fitting, artificial neural network, or some other statistical method. The model should consider both the individual company's data as well as the broad market data. Once ready for use, the prediction model should be activated to look for trends and patterns in stock prices as they are collected in real time.

Potential services that will be provided by the forecaster service include:

- Given a stock *x*, suggest an action, such as "buy," "sell," "hold," or "sit-out;" we will assume that the forecaster provides recommendation for one stock at a time
- Recommend a stock to buy, from all stocks that are being tracked, or from all in a given industry/sector

A key step in specifying the forecaster service is to determine its *Web service interface*: what will go in and what will come out of your planned Web service? Below I list all the possible parameters that I could think of, which the client and the service could exchange. The development team should use their judgment to decide what is reasonable and realistic for their own team to achieve within the course of an academic semester, and select only some of these parameters for their Web service interface.

Parameters sent by the facilitator to a forecaster (from the client to a Web service) in the inquiry include:

- Stock(s) to consider: *individual* (specified by ticker symbol), *select-one-for-sector* (sector specified by a standard category), *any* (select the best candidate)
- Trade to consider: *buy*, *sell*, *hold*, *sit-out* OR Position to consider: *long*, *short*, *any*
- Time horizon for the investment: integer number
- Funds available: integer number for the capital amount/range
- Current portfolio (if any) or current position for the specified symbol

Some of these parameters may not be necessary, particularly in the first instantiation of the system. Also, there are privacy issues, particularly with the last two items above, that must be taken into account. The forecaster Web-services are run by third parties and the trader may not wish to disclose such information to third parties.

Results returned by a forecaster to the facilitator (for a single stock per inquiry):

- Selected stock (if the inquiry requested selection from "sector" or "any")
- Prediction: price trend or numeric value at time *t* in the future
- Recommended action and position: buy, sell, hold, sit-out, go-short
- Recommended horizon for the recommended action: time duration
- Recommendation about placing a protective sell or buy Stop Order.
- Confidence level (how confident is the forecaster about the prediction): range 0 – 100 %

The performance of each prediction service should be evaluated as follows. Once activated, each predicted price value should be stored in a local database. At a future time when the actual value becomes known, it should be recorded along with the previously predicted value. A large number of samples should be collected, say over the period of tens of days. We use absolute mean error and average relative error as indices for performance evaluation. The *average relative error* is defined as $\left(\sum_i |y_i - \hat{y}_i|\right) / \sum_i y_i$, where y_i and $\hat{y}_i$ are the actual and predicted prices at time i, respectively.

Statement of Requirements

Extensions

Risk analysis to analyze “what if” scenarios.

Additional information about this project can be found at the book website, given in Preface.

1.5.3 Remarks about the Projects

My criteria in the selection of these projects was that they are sufficiently complex so to urge the students to enrich their essential skills (creativity, teamwork, communication) and professional skills (administration, leadership, decision, and management abilities when facing risk or uncertainty). In addition, they expose the students to at least one discipline or problem domain in addition to software engineering, as demanded by a labor market of growing complexity, change, and interdisciplinarity.

The reader should observe that each project requires some knowledge of the problem domain. Each of the domains has myriads of details and selecting the few that are relevant requires a major effort. Creating a good model of any domain requires skills and expertise and this is characteristic of almost all software engineering projects—in addition to software development skills, you always must learn something else in order to build a software product.

The above projects are somewhat deceptive insofar as the reader may get impression that all software engineering projects are well defined and the discovery of what needs to be developed is done by someone else so the developer’s job is just software development. Unfortunately, that is rarely the case. In most cases the customer has a very vague idea of what they would like to be developed and the discovery process requires a major effort. That was the case for all of the above projects—it took me a great deal of fieldwork and help from many people to arrive at the project descriptions presented above. In the worst case you may not even know who will be your customer, as is the case for traffic monitoring (described at the book website, given in Preface) and the investment fantasy league (Section 1.5.1). In such cases, you need to invent your own customers—you need to identify who might benefit from your product and try and interest them in participating in the development.

Frederick Brooks, a pioneer of software engineering, wrote that “the hardest single part of building a software system is deciding precisely what to build” [Brooks, 1995: p. 199]. By this token, the hardest work on these projects is already done. The reader should not feel short-changed, though, because difficulties in deriving system requirements will be illustrated.

Example 1.2 RFID tags in retail

The following example illustrates of how a typical idea for a software engineering project might evolve. The management of a grocery supermarket (our customer) contacted us with an idea for a more effective product promotion. Their plan is to use a computer system to track and influence people’s buying habits. A set of logical rules would define the conditions for generating promotional offers for customers, based on the products the customer has already chosen. For example, if customer removed a product *A* from a shelf, then she may be offered a discount coupon on product *B*. Alternatively, the customer may be asked if she may also need product *C*. This last feature serves as a reminder, rather than for offering discount coupons. For example, if a customer removes a soda bottle from a shelf, she may be prompted to buy potato chips, as well.

To implement this idea, the store will use Radio Frequency Identification (RFID) tags on all store items. Each tag carries a 96-bit EPC (Electronic Product Code). The RFID tag readers will be installed on each shelf on the sales floor, as well as in the cashier registers at the sales point. When a tag is removed from the region of a reader’s coverage, the reader will notify the computer system that the given tag disappeared from its coverage area. In turn, the system will apply the logical rules and show a promotional offer on a nearest display. We assume that each shelf will have an “offers display” that will show promotional offers or reminders related to the last item that was removed from this shelf.

As we consider the details of the idea, we realize that the system will not be able to identify individual customers and tailor promotional offers based on the customer identity. In addition to privacy concerns, identifying individual customers is a difficult technological problem and the store management ruled out potential solutions as too expensive. We do not care as much to know who the customer is; rather, we want to know the historic information about other items that this customer placed in her cart previously during the current shopping episode to customize the offer. Otherwise, the current offer must be based exclusively on the currently removed item and *not* on prior shopping history. Next, we come up with an idea of installing RFID tag readers in the shopping carts, so we can track the current items in each shopping cart. However, the supermarket management decides against this approach, because of a high price of the readers and concerns about their robustness to weather and handling or vandalism.

As a result, we conclude that logical IF-THEN-ELSE rules for deciding about special offers will take as input only a single *product identity*, based on the RFID tag of the item the customer has just removed from the shelf. The discount coupon will be a "virtual coupon," which means that the customer is told about the discounted product, and the discount amount will be processed at the cashier's register during the checkout. The display will persist for a specified amount of time and then automatically vanish. The next question is whether each display will be dedicated to a single product or shared among several adjacently shelved products? If the display will be shared, we have a problem if other items associated with this display are removed (nearly) simultaneously. How do we show multiple offers, and how to target each to the appropriate customer? A simple but difficult question is, when the displayed coupon should vanish? What if the next customer arrives and sees it before it vanishes? Perhaps there is nothing bad with that, but now we realize that we have a difficulty *targeting* the coupons. In addition, because the system does not know what is in the customer's cart, it may be that the customer already took the product that the system is suggesting. After doing some market research, we determine that small displays are relatively cheap and an individual display can be assigned to each product. We give up targeting customers, and just show a virtual coupon as specified by the logical rules.

Given that the store already operates in the same way with physical, paper-based coupons, the question is if it is worth to install electronic displays or use RFID tags? Is there any advantage of upgrading the current system? If the RFID system input is used, then the coupon will appear when an item is removed. We realize that this makes no sense and just show the product coupon all the time, same as with paper-based coupons. An advantage of electronic displays is that they preclude having the store staff go around and place new coupons or remove expired ones.

We started with the idea of introducing RFID tags and ended up with a solution that renders them useless. An argument can be made that tags can be used to track product popularity and generate promotional offers based on the current demand or lack thereof. A variation of this project, with a different goal, will be considered in Problem 2.15 at the end of Chapter 2.

There are several lessons to be learned about software engineering from the above example:

- One cannot propose a solution without a deep understanding of the problem domain and working closely with the customer
- Requirements change dynamically because of new insights that were not obvious initially
- Final solution may be quite different from the initial idea.

The project descriptions presented earlier in this chapter are relatively precise and include more information than what is usually known as the *customer statement of work*, which is an expression, from a potential customer, of what they require of a new software system. I expressed the requirements more precisely to make them suitable for one-semester (undergraduate) student projects. Our focus here will be on what could be called "core software engineering."

On the other hand, the methods commonly found in software engineering textbooks would not help you to arrive at the above descriptions. Software engineering usually takes from here—it assumes a defined problem and focuses on finding a solution. Having defined a problem sets the constraints within which to seek for the solution. If you want to broaden the problem or reframe it, you must go back and do some fieldwork. Suppose you doubt my understanding of financial markets or ability to extract the key aspects of the security trading process (Section 1.3.2) and you want to redefine the problem statement. For that, software engineering methods (to be described in Chapter 2) are not very useful. Rather, you need to employ ethnography or, as an engineer you may prefer Jackson's "problem frames" [Jackson 2001], see Chapter 3. Do I need to mention that you better become informed about the subject domain? For example, in the case of the financial assistant, the subject domain is finance.

1.6 Summary and Bibliographical Notes

Because software is pure invention, it does not have physical reality to keep it in check. That is, we can build more and more complex systems, and pretend that they simply need a little added debugging. Simple models are important that let us understand the main issues. The search for simplicity is the search for a structure within which the complex becomes transparent. It is important to constantly simplify the structure. Detail must be abstracted away and the underlying structure exposed.

Although this text is meant as an introduction to software engineering, I focus on critical thinking rather than prescriptions about structured development process. Software development can by no means be successfully mastered from a single source of instruction. I expect that the reader is already familiar with programming, algorithms, and basic computer architecture. The reader may also wish to start with an introductory book on software engineering, such as [Larman, 2005; Sommerville, 2004]. The Unified Modeling Language (UML) is used extensively in the diagrams, and the reader unfamiliar with UML should consult a text such as [Fowler, 2004]. I also assume solid knowledge of the Java programming language. I do offer a brief introduction-to/refresher-of the Java programming language in Appendix A, but the reader lacking in Java knowledge should consult an excellent source by Eckel [2003].

The problem of scheduling construction tasks (Section 1.1) is described in [Goodaire & Parmenter, 2006], in Section 11.5, p. 361. One solution involves first setting posts, then cutting, then nailing, and finally painting. This sequence is shown in Figure 1-2 and completes the job in 11 units of time. There is a second solution that also completes the job in 11 units of time: first cut, then set posts, then nail, and finally paint.

Although I emphasized that complex software systems defy simple models, there is an interesting view advocated by Stephen Wolfram in his NKS (New Kind of Science): http://www.wolframscience.com/ , whereby some systems that appear extremely complex can be captured by very simple models.

In a way, software development parallels the problem-solving strategies in the field of artificial intelligence or means-ends analysis. First we need to determine *what* are our goals ("ends"); next, represent the current state; then, consider *how* ("means" to employ) to minimize the difference between the current state and the goal state. As with any design, software design can be seen as a difference-reduction activity, formulated in terms of a symbolic description of differences. Finally, in autonomic computing, the goals are represented explicitly in the program that implements the system.

There are many reasons why some systems succeed (e.g., the Web, the Internet, personal computer) and others fail, including:

- They meet a real need
- They were first of their kind
- They coevolved as part of package with other successful technologies and were more convenient or cheaper (think MS Word versus WordPerfect)
- Because of their technical excellence

Engineering excellence alone is not guarantee for success but a clear lack of it is a guarantee for failure.

There are many excellent and/or curious websites related to software engineering, such as:

Teaching Software Engineering – Lessons from MIT, by Hal Abelson and Philip Greenspun: http://philip.greenspun.com/teaching/teaching-software-engineering

Software Architecture – by Dewayne E. Perry: http://www.ece.utexas.edu/~perry/work/swa/

Software Engineering Academic Genealogy – by Tao Xie:
http://www.csc.ncsu.edu/faculty/xie/sefamily.htm

Section 1.2.1: Symbol Language

Most people agree that symbols are useful, even if some authors invent their own favorite symbols. UML is the most widely accepted graphical notation for software design, although it is sometimes criticized for not being consistent. Even in mathematics, the ultimate language of symbols, there are controversies about symbols even for such established subjects as calculus (cf., Newton's vs. Leibnitz's symbols for calculus), lest to bring up more recent subjects. To sum up, you can invent your own symbols if you feel it absolutely necessary, but before using them, explain their meaning/semantics and ensure that it is always easy to look-up the meanings of your symbols. UML is not ideal, but it is the best currently available and most widely adopted.

Arguably, symbol language has a greater importance than just being a way of describing one's designs. Every language comes with a theory behind it, and every theory comes with a language. Symbol language (and its theory) helps you articulate your thoughts. Einstein knew about the general relativity theory for a long time, but only when he employed tensors was he able to articulate the theory (http://en.wikipedia.org/wiki/History_of_general_relativity).

Section 1.2.3: Object-Oriented Analysis and the Domain Model

I feel that this is a more gradual and intuitive approach than some existing approaches to domain analysis. However, I want to emphasize that it is hard to sort out software engineering approaches into right or wrong ones—the developer should settle on the approach that produces best results for him or her. On the downside of this freedom of choice, choices ranging from the dumbest to the smartest options can be defended on the basis of a number of situation-dependent considerations.

Some authors consider object-oriented analysis (OOA) to be primarily the analysis of the existing practice and object-oriented design (OOD) to be concerned with designing a new solution (the system-to-be).

Modular design was first introduced by David Parnas in 1960s.

A brief history of object orientation [from Technomanifestos] and of UML, how it came together from 3 amigos. A nice introduction to programming is available in [Boden, 1977, Ch. 1], including the insightful parallels with knitting which demonstrates surprising complexity.

Also, from [Petzold] about ALGOL, LISP, PL/I.

Objects: http://java.sun.com/docs/books/tutorial/java/concepts/object.html

N. Wirth, "Good ideas, through the looking glass," *IEEE Computer*, vol. 39, no. 1, pp. 28-39, January 2006.

H. van Vliet, "Reflections on software engineering education," *IEEE Software*, vol. 23, no. 3, pp. 55-61, May-June 2006.

[Ince, 1988] provides a popular account of the state-of-the-art of software engineering in mid 1980s. It is worth reading if only for the insight that not much has changed in the last 20 years. The jargon is certainly different and the scale of the programs is significantly larger, but the issues remain the same and the solutions are very similar. Then, the central issues were reuse, end-user programming, promises and perils of formal methods, harnessing the power of hobbyist programmers (today known as open source), and prototyping and unit testing (today's equivalent: agile methods).

Section 1.3.1: Case Study 1: From Home Access Control to Adaptive Homes

The Case Study #1 Project (Section 1.3.1) – Literature about the home access problem domain:

A path to the future may lead this project to an "adaptive house" [Mozer, 2004]. See also:

Intel: Home sensors could monitor seniors, aid diagnosis (ComputerWorld)
http://www.computerworld.com/networkingtopics/networking/story/0,10801,98801,00.html

Another place to look is: University of Florida's Gator Tech Smart House [Helal *et al.*, 2005], online at: http://www.harris.cise.ufl.edu/gt.htm

For the reader who would like to know more about home access control, a comprehensive, 1400-pages two-volume set [Tobias, 2000] discusses all aspects of locks, protective devices, and the methods used to overcome them. For those who like to tinker with electronic gadgets, a great companion is [O'Sullivan & T. Igoe, 2004].

Biometrics:

Wired START: "Keystroke biometrics: That doesn't even look like my typing," *Wired*, p. 42, June 2005. Online at: http://www.wired.com/wired/archive/13.06/start.html?pg=9

Researchers snoop on keyboard sounds; Computer eavesdropping yields 96 percent accuracy rate. Doug Tygar, a Berkeley computer science professor and the study's principal investigator http://www.cnn.com/2005/TECH/internet/09/21/keyboard.sniffing.ap/index.html

Keystroke Biometric Password; Wednesday, March 28, 2007 2:27 PM/EST

BioPassword purchased the rights to keystroke biometric technology held by the Stanford Research Institute. On March 26, 2007, the company announced BioPassword Enterprise Edition 3.0 now with optional knowledge-based authentication factors, integration with Citrix Access Gateway Advanced Edition, OWA (Microsoft Outlook Web Access) and Windows XP embedded thin clients.

http://blogs.eweek.com/permit_deny/content001/seen_and_heard/keystroke_biometric_password.html?kc=EWPRDEMNL040407EOAD

See also [Chellappa *et al*., 2006] for a recent review on the state-of-the-art in biometrics.

Section 1.4: The Object Model

The concept of information hiding originates from David Parnas [1972].

D. Coppit, “Implementing large projects in software engineering courses,” *Computer Science Education*, vol. 16, no. 1, pp. 53-73, March 2006. Publisher: Routledge, part of the Taylor & Francis Group

J. S. Prichard, L. A. Bizo, and R. J. Stratford, “The educational impact of team-skills training: Preparing students to work in groups,” *British Journal of Educational Psychology*, vol. 76, no. 1, pp. 119-140, March 2006.

(downloaded: NIH/Randall/MATERIALS2/)

M. Murray and B. Lonne, “An innovative use of the web to build graduate team skills,” *Teaching in Higher Education*, vol. 11, no. 1, pp. 63-77, January 2006. Publisher: Routledge, part of the Taylor & Francis Group