

Part I

**An Analysis of Product  
Development**



# Chapter 2

# Systems

## 2.1 Introduction

The concept of a system is a crucial tool in understanding system development methods. Unfortunately, different authors use different definitions of this concept. Some view a system as an organized collection of components, others view the concept of a state space as central to the system concept and still others take purposive behavior as the defining characteristic of a system. In section 2.2, we start from a minimal concept of a system that is common to all different approaches, viz. that of a system as an observable part of the world. In section 2.3, we introduce the familiar concept of a system as an organized collection of related elements. In section 2.4, we add the perspective of observable system behavior, which allows us to define the important concept of state space. In section 2.5, we look at systems from the perspective of their function for their environment, which allows us to introduce the concept of system objective. This falls short of the idea of purposive behavior mentioned above, but it suffices for our purpose. The distinctions between function, behavior and structure are summarized in section 2.6. In section 2.7, we look at a concept that is characteristic for data-manipulating systems such as computers, the universe of discourse.

## 2.2 System Boundary

### 2.2.1 The observability of systems

It is possible to speak of the system of natural numbers, a system of law, a software system, a system of logical inference rules and the solar system. For our purpose, a system concept that would encompass all these different uses of the word “system” would be too general to be useful. We will therefore restrict the use of the term to **observable** systems, where the concept of observation is left unexplained. If pressed for an explanation, I would say that an observation is always an interaction of a system with its environment, where the interaction may be initiated by the system or by its environment. Conversely, any interaction of a system with its environment is viewed in this book as an observation of the system by its environment. (Depending upon one’s point of view, it can also be viewed as an observation of the environment by the system). This reduces one unexplained concept (observation) to

another (interaction) and vice versa. Perhaps this is typical for starting points.

In what follows, I treat **observation** as synonymous with **interaction** between systems. To make the situation more vivid, I will often treat the environment of a system as an *observer* who observes the system by interacting with it. Each interaction can be viewed as an experiment in which the observer learns something about the system.

Given this, a **system** is defined as any actual or possible part of reality that, if it exists, can be observed. We illustrate this definition with some examples, non-examples and borderline cases of systems:

- Physical objects like cars, stones, trees, elevators and airplanes are systems. Observations of these systems include observations of their color, weight, speed, and location. To make observations, we perform an interaction with the system, and when we interact with these systems, we make observations.
- Intangible objects like operating systems, database management systems, information systems and organizations are systems. From the point of view of physics, these objects are not observable. Nevertheless, they can interact with other systems and from our point of view therefore, they are observable. If we were to restrict ourselves to observations allowed in physics, organizations would be invisible, but since we allow observations allowed in social science and psychology, organizations are observable. Observations of these systems include observations of responses to commands, queries, requests, statements, of the time it takes to produce these responses, of resource usage during the production of the responses, etc.
- Abstract entities like numbers, truth values and letters are not systems. The number 3 cannot interact with other numbers or with anything else. It does stand in mathematical relations to other numbers, but this is different from engaging in interactions with those numbers. The mathematical relations are not events occurring in time. Similarly, the letter denoted by the symbol “a” cannot interact with other systems. By contrast, a *symbol* that represents the number 3 or the letter “a” can interact with other systems. It can be written, read and erased, for example.
- The “system” of Peano axioms for the natural numbers is not a system in our sense, for it cannot interact with other systems. It just has some logical relations to other axiom “systems” and to propositions about the natural numbers.
- Physicists define a “closed system” as something that cannot interact with its environment. For example, a closed container of gas is an idealized body that has no interaction with its environment. “Closed systems” are useful fictions for doing thought experiments and for approximating the behavior of real systems, but they do not exist in reality. They are not systems as we define the term here.
- A system of law is a system in our sense. It has a period of existence and may interact with other systems of law as well as with events occurring in the society ruled by the system of law. Nevertheless, it also shares some properties with the “system” of natural numbers. A system of law has for example logical relations to other systems of law and to propositions about the real world. It is therefore a borderline case of our concept of a system.

As pointed out above, many entities that we talk about are social constructs that have no physical existence. In one way or another, employees, committees, organizations, bank accounts, and budgets are socially constructed entities that are physically invisible. All we can observe physically are human bodies, buildings, symbols written on pieces of paper, symbols printed on screens, and sounds produced by people. Nevertheless, these socially constructed entities always include observation procedures in their definition. Observable properties of an employee include his or her employee number, role in the organization, and salary; observable properties of a committee are its name, function and composition; observable properties of a bank account are its number, owner and balance. These social constructs exist because we agreed upon ways to observe them. If no observation procedures were agreed upon, the salary of an employee, the composition of a committee and the balance of a bank account would not exist.

From a physical point of view, making these observations always consists of interacting with *something else*. Often, this something else is a system that *is* physically observable, such as a written or printed record. For example, we observe the balance of a bank account by observing a paper-based or computer-based administration. In this book, we view these physical interactions as *implementations* of abstract interactions with these social constructs. In the physical interactions, we observe properties of these social constructs.

Of course, occasionally, there may be conflicts about the observable properties of social constructs; but then there are procedures agreed upon to resolve those conflicts. For example, when there is disagreement about the actual balance on of a bank account, we turn to recorded statements to resolve the disagreement; if there is disagreement about what is recorded, we resort to procedures to reach an irrefutable verdict about what the balance is; and if this verdict contradicts some written records, then the verdict states the fact of the matter and the written records are overruled. This means that observation of the balance is not the same thing as observation of what is recorded. Reading what is recorded is a way to *implement* the observation, but this implementation may be wrong and we may turn to other implementations.

Our definition of systems has three important consequences. First, define the **environment** of a system as that part of the world with which it can interact. It then follows from the definition that each system has an environment; and that the environment of a system is a system itself too. The choice to call it an environment merely indicates our focus.

A second consequence is that systems may be actual or possible. We define a system to **exist** if it is capable of interacting with other existing systems. This is a circular definition, for the concept of an existing system is used to define the concept of an existing system. However, this circularity is harmless. The definition just says that to exist is to be able to interact, i.e. to be able to initiate or suffer interactions. For example, a symbol stored on disk exists, because it can be operated upon: it can be read or erased. According to this concept of existence, abstract entities like numbers and truth values do not exist, because they cannot interact with existing systems. By contrast, a symbol written on paper that represents a number exists, because it can be manipulated.

The third consequence of the definition is that systems are **dynamic**. This is because systems can *interact* with other systems and interactions occur in time. Systems therefore exist in time. Going through the list of examples and nonexamples of systems above, we see that each of the examples can be said to exist in time and each of the nonexamples stands outside time.

- Each system should have an underlying **system idea** that describes its coherence.
- Interaction among system components (**cohesion**) is higher than interaction between the system and its environment (**coupling**).
- Changes within a system should cause minimal changes outside the system.
- There are more relations between system components than between system components and the environment.
- More energy is needed to transfer something across the system boundary than to transfer something within the system boundary.
- Each system boundary should “divide nature at its joints”.
- The system boundary should be chosen in such a way that the number of regularities in the behavior of the system is higher than with any other choice of system boundary.
- The system boundary should be chosen in such a way that system behavior is simpler than with any other choice of system boundary.

Figure 2.1: Modularity guidelines.

### 2.2.2 System boundary and modularity

We call the set of *all possible* interactions of a system its **interface** or **boundary**. Some interactions in a system’s interface may never occur, others may occur frequently. The interface of a candy store contains interactions like *sell chocolate bar*, which may occur frequently, and *sell 100 bars of chocolate*, which may never actually occur.

The choice of where to put a system boundary is up to us, the observers of the system. Of course, some choices are better than others. Suppose that we define a system  $S$  as consisting of a coffee machine *excluding* the buttons to operate it. Since  $S$  is an observable part of the world, it is a system according to our definition. However, the observable behavior of  $S$  is harder to understand than it would be if we had included the buttons in the system. In both cases, we can observe interactions like *insert coin* and *emit cup*, but without the buttons, we cannot observe interactions like *push coffee button*. This makes system behavior unnecessarily hard to understand, because the machine seems to emit cups without reason. If we had included the behavior of the buttons in the system boundary, then observable behavior would have been easy to understand. Apparently, there are good and bad choices of a system boundary.

There is in general one guideline for defining a system boundary: define it such that the system is **modular**. This means, vaguely, that the system must act as a more or less independent unit and that the separation between the system and its environment is “larger” than the separation of the components of the system. Figure 2.1 lists some criteria for modularity. These are still vague, but nevertheless should convey a message. The underlying **idea** of a system is its concept of operation, the rationale of its behavior. When a system has a single underlying idea, it is likely to be coherent and should therefore be modular. For example, the idea of a solar system is that a number of bodies revolve around the sun, and the idea of a coffee machine is that it dispenses coffee upon request. From this idea of a coffee machine, it follows that there should be some device that allows the user to

make a request for coffee. Consequently, the system boundary should be chosen in such a way that this device, say a button, is included.

## 2.3 System Structure

### 2.3.1 Subsystems and aspect systems

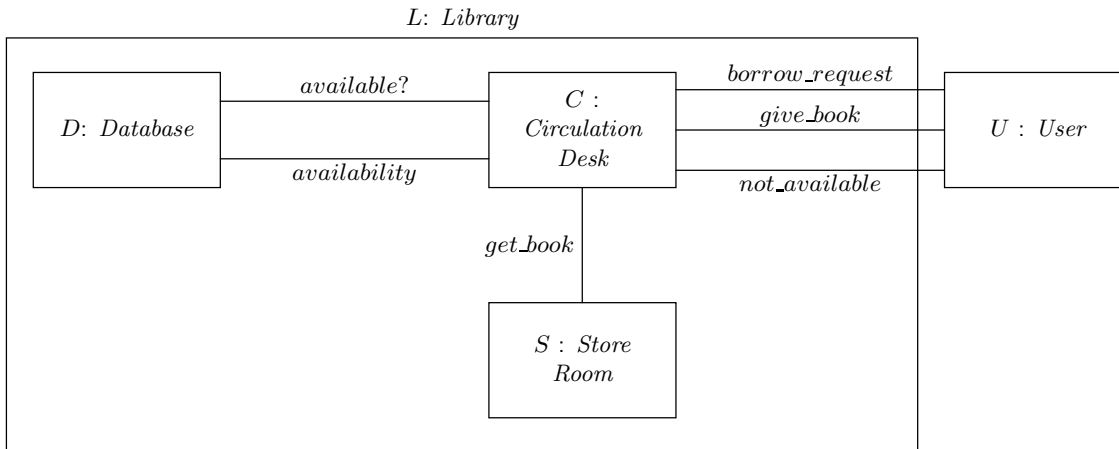
Any observable part of the world, except the smallest particle (if it exists), consists of components that themselves are observable and hence are systems. These components are called **subsystems**. For example, an organization consists of departments, a software system consists of modules, a house consists of rooms, etc. We will say that a system is **implemented** in its subsystems. The behavior of a system is realized by means of the behavior of its subsystems, including their interactions with each other and with the environment. This means that a system is a collection of subsystems *acting as a whole*. An arbitrary collection of mechanical parts is not usefully regarded as a system; a collection of parts put together to form a car is usefully regarded as a system. The system behaves as it does, not only because each of its parts behaves as it does, but because the parts interact in a way such that the total system behavior is realized. A system is thus an *organized* collection of interacting subsystems. As expressed by the modularity heuristics, there is a cohesion between the subsystems.

Subsystem boundaries should be chosen in such a way that the subsystems are modular. The entire system is then said to have a **modular architecture**.

We define an **aspect** of a system as a subset of all possible interactions of the system. If we observe all subsystems of a system  $S$  but restrict our observations to some of the interactions between them, then we observe an **aspect system** of  $S$ . For example, the financial aspect system of an organization consists of all departments of the organization, together with their financial interactions. An information system may be viewed as a *subsystem* of an organization, consisting of hardware, software, users, user procedures and the data manipulated by them. Alternatively, we may view it as an *aspect system* of the organization, consisting of all departments of the organization and the flows of information among them.

We call interactions in which only subsystems of a system  $S$  participate **internal** to  $S$ , and interactions in which at least one external system of  $S$  participates **external**. Figure 2.2 shows a library  $L$  and an environment consisting of a single user  $U$ , and some subsystems of  $L$ , viz. a database  $D$ , a circulation desk  $C$  and a store room  $S$ . Interactions are represented by lines. The interactions *available?*, *availability* and *get\_book* are invisible for the environment  $U$  and hence internal to  $L$ . The interactions *borrow\_request*, *give\_book* and *not\_available* are observable by  $U$ , and hence external. Note that in real life, the clerk may search the database and go to the store room in full sight of the user. However, these observations are not relevant interactions with the user and they have not been modeled in figure 2.2. As far as this model is concerned, they are invisible to the user.

Engineers usually try to hide internal interactions from view by the users. Thus, a protective cover is placed around the internals of a coffee machine to hide implementation behavior that is irrelevant for the user of the coffee machine. In a business viewed as a system, such a protective cover may take the form of procedures for interacting with customers, layout of offices, rules for restricting the disseminating information, etc., all of



**Figure 2.2:** Some subsystems and interactions of a library.

which serve to hide internal interactions from external view. Interestingly, there are also internal interactions in a business that need no protective cover to be hidden: gossip at the coffee machine, secret agreements, etc. Indeed, the protective cover of these internal interactions is itself invisible. People can hide some of their interactions by pretending that they never took place. Unlike the metal plate covering the internals of a coffee machine, the cover-up of secret interactions between people can itself be made invisible. This is something the information analyst will have to deal with when he or she tries to model some of the internal business interactions.

### 2.3.2 A hierarchy of system levels

The world can be divided into levels such that systems that exist at any level are decomposed into systems at the next lower level. We call this the **aggregation** or **implementation hierarchy** in this book. At each level of aggregation, we view the world as a collection of interacting systems. The behavior of a system is the result of the interactions among its subsystems and between its subsystems and its environment. For example, figure 2.2 shows the interactions between the subsystems of a library and its environment that realize the *borrow* interaction between the library and a user. The behavior of the system is sometimes said to **emerge** from the behavior and interaction of the subsystems. The concept of emergence indicates that the behavior of the subsystems itself is not sufficient to explain the behavior of the compound system. To explain the compound behavior, one must additionally take into account the way in which the subsystems interact.

Any partitioning of the world into hierarchical levels is a simplification. For example, the library database may be part of a national network of library databases in which the clerk can search through the normal database interface. Are these other databases subsystems of the library? Is the library a subsystem of this network? Any answer to these questions leaves out an aspect of reality. Nevertheless, hierarchical decomposition is the major tool of the human mind for complexity reduction. We use it precisely with the purpose of simplifying

Level	Examples
Social system	An organization, a company division, a set of organizations
Computer-based system	An information system, an elevator system, an EDI network, a flight simulator
Software system	A database system, an elevator control system, a network communication software system
Software subsystem	An error recovery module, an authentication subsystem, a scheduler

**Figure 2.3:** A useful aggregation hierarchy from the software engineer’s point of view.

our model of reality. As long as we do not forget that we are dealing with a simplification, hierarchical decomposition allows us to focus on the systems of interest within an otherwise bewildering complexity of systems within systems.

In this book, we will use the system hierarchy shown in figure 2.3. We explain the hierarchy level by level.

- Any system that we build for human use interacts with a human environment. The top level of the hierarchy is therefore that of **social systems**. The social system into which our product is embedded may be a human-machine system consisting of one user and one machine, or it may be an organization, or a group of organizations. For example, a word processing package is used by an individual, an information system is used by an organization, and an EDI system is used by a group of organizations.
- Taking the archetypical case of a system developed for use in an organization, the next lower level of aggregation is that of a **computer-based system**. Examples of computer-based systems are automated information systems, EDI systems, elevator systems and flight simulators. Of course, many of these systems may be implemented without using computers — paper-based information systems, electromagnetic elevator control systems, etc. Figure 2.3 takes a software engineer’s point of view and focuses on computer-based systems.
- In the cases of interest for us, the computer-based system is composed, at the next lower level, of hardware and **software** systems (and possibly other kinds of systems). For example, an information system, viewed as a subsystem of an organization, is composed of hardware, software, users, procedures followed by users, and data manipulated by the people and the software. If the computer-based system is composed of hardware and software only, such as an elevator system, then it is customary to call it simply *the system* and to call the software *embedded*. Figure 2.3 takes a software engineer’s point of view and shows only software systems below the computer-based system level. There are other relevant kinds of systems at this level of aggregation, such as hardware systems, users, and operators. The software engineer must interact with developers specialized in these other subsystems.
- Software systems are in turn composed of **software subsystems**, which may be called *packages, modules, classes*, or whatever.

If we develop a system at some level in this hierarchy, then we must determine *what* that system must do and *how* the system is going to do this. A specification of *what* the system must do at that level of aggregation is usually called a *requirements specification*. This is contrasted with *how* the system is realized internally at the next lower level of aggregation, which is called its *implementation*. The hierarchy of figure 2.3 shows that these distinctions are meaningless if we do not indicate to which level of the hierarchy we refer. A specification of the *implementation* of a computer-based system contains a specification of the *requirements* for its software subsystems, for example. This is not different from the fact that in an apartment building, one person's floor is another person's ceiling [76] — but it is more confusing, because the distinctions we make in system engineering are cast in concepts rather than in concrete.

To resolve the ambiguity between the *what* and the *how*, we should replace the distinction between *what* a system does and *how* it does it with the distinction between

- an indication of the level of aggregation we want to refer to, and
- an indication of the observer looking at that level of aggregation.

For example, at the level of libraries, a library user can observe such interactions as *borrow* a document (figure 2.2). At the same level of aggregation, a publisher can observe other interactions, such as *buy document*. These two observers (user and publisher) see different observable behavior of the library but they observe the library at the same level of aggregation. Descending one level of aggregation, the very same observers will see different behavior too. At the level of library subsystems, the library user can observe interactions such as *borrow\_request*, *give\_book* and *not\_available* and the publisher observes such interactions as *order document*, *send invoice* and *pay for invoice*.

## 2.4 System Behavior

### 2.4.1 System state

Systems engage in observable behavior by engaging in interactions with their environment. Any system with interesting behavior has a *memory* of past interactions such that its behavior in a current interaction depends upon this memory. The memory of a system is called the system **state**, and the set of all its possible states is called its **state space**. A system is called **discrete** when its state space is a discrete set. Skipping the formal definitions, a state space is discrete when all states in it can be numbered using the natural numbers. For many purposes, organizations can be modeled as discrete systems. Information systems and many communication systems are also usually specified as discrete systems.

A system is called **continuous** when state information cannot be encoded by the natural numbers but we need real numbers to represent the possible states. In a **hybrid** system, part, but not all of a state can be encoded by the natural numbers. An example of a hybrid system is a computer-controlled audio system. The computer control itself is discrete, but it must interface with continuous systems such as potentiometers and speakers, so that the total system is hybrid.

The concept of a state is linked to those of the aggregation level and the observer of a system. For example, the state of the library is the sum of the states of each of its subsystems; in turn, each of these subsystem states is the sum of the states of *its* subsystems,

etc. We can decompose the state of the library down to any level of decomposition we want. However, for the library user, it is only useful to observe some of the states at the highest level of aggregation. A library user can observe whether or not a document is available for borrowing — e.g. by trying to borrow the document. What this means for the state of the store room and the database system is not important to the user. For the circulation desk clerk, on the other hand, these internal states *are* important. In other words, at any level of aggregation, and with respect to an observer or class of observers at that level of aggregation, part of the state space of a system is *observable*.

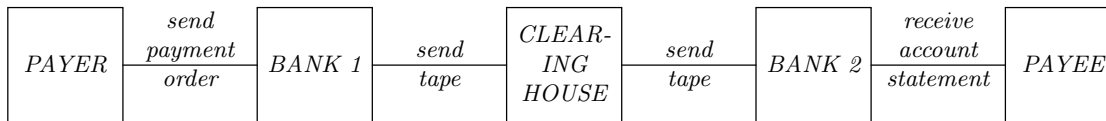
As a consequence, there is no such thing as “the” state space of a system. We must define a state space with respect to a level of aggregation and with respect to an observer of system states at that level of aggregation. This determines which states are observable and at what level of detail they are observable.

### 2.4.2 System transactions

In this subsection, we restrict the discussion to observable system states. At the start of an interaction, a system is in a certain (observable) state and at the end of the interaction (assuming the interaction has an end), it is again in in a state. Let us call these states the *initial* and *final* states, respectively. Any state of the system after the initial state of the interaction and before the final state of the interaction is called an *intermediary* state of the interaction. An interaction is called a **transaction** if it is has no observable intermediary states. We say that transactions are **atomic**. Transactions have the atomicity property that at any moment in time, either the transaction has occurred or it has not occurred; there is no observable intermediary state in which the transaction has started to occur but has not yet finished. For example, at the library level of aggregation and with respect to library users, *borrow* is a transaction that either occurs or does not occur. At the next lower level of aggregation, and again for library users, transactions like *borrow\_request* and *give\_book* can be observed. These lower level transactions implement the higher level *borrow* transaction (figure 2.2).

According to Gray [126], the concept of transaction has its roots in contract law. In making a contract, two or more parties negotiate for a while and then make a deal. The deal is made binding by some ceremony, like signing a contract or even a simple handshake or nod. The function of this ceremony is to mark a discrete point in time, before which the contract did not exist and after which the contract exists; as far as the observers of the negotiations are concerned, there are no intermediary states between non-existence and existence of the contract. Important observers of the negotiations are the parties to the contract themselves and, possibly, the judge if there is disagreement about the contract.

Just like the concept of observable state, a transaction is defined at a particular level of aggregation and with respect to a particular observer, or class of observers. The crucial characteristic of a transaction is that, at that level of aggregation and with respect to those observers, it has no observable intermediary states. To implement this, the next lower level of aggregation must offer a **rollback** mechanism to undo an unsuccessful transaction attempt, and a **commitment** mechanism by which a successful transaction attempt is turned into a transaction. For example, during the negotiations for the sale of a house, either party can withdraw, after which the transaction attempt has been rolled back and no transaction is said to have taken place. However, after the sales contract is signed, the transaction has been committed and rollback is not possible.



**Figure 2.4:** Payment by giro as a complex transaction. The role of postal services has been ignored here.

Deciding when a transaction has been committed may be a case to be ruled in court. For example, making a payment by giro involves a process involving in general two banks, a clearing house and two customers (figure 2.4). High court in the Netherlands has ruled that the payment transaction takes place at the moment when the receiver of the payment gets the legal power over the transferred amount [125, page 270]. For a payment by giro, this happens at the moment that the receiver’s bank credits the receiver’s bank account, i.e. when *BANK 2* credits the account of the payee.

After a transaction has been committed, we may try to undo some of its results, by means of a **compensating transaction**. For example, after we bought a house, we may want to undo this transaction by reselling the house to the original owner. After borrowing a document, we may change our mind and return the document. These compensating transactions do *not* roll back the original transaction, but try to undo some of its effects. The difference with a rollback is that once a transaction has been committed, it is part of the observable history of the system. By contrast, when a transaction attempt is rolled back, the transaction never took place and it is not part of the observable history of the system. This is often expressed by saying that the result of a committed transaction is available to other transactions. This means that later transactions can use results of earlier transactions that occurred in the observable history of the system to that point.

We may compose transactions synchronously with other transactions and preserve atomicity. For example, suppose you reserve a document with the library and borrow it later. This borrowing transaction may be viewed as a synchronous occurrence of two transactions: one in which a reservation is terminated, and another one, occurring at the same time, in which a document loan is started. The resulting transaction is composed of two other transactions but is still atomic, because it has no observable intermediary state.

What is atomic at one level of aggregation may be a process at the next lower level. For example, the *borrow* transaction is atomic at the level of social systems, since it is an atomic interaction between a library and a customer, but it is a process at the level of computer-based systems, since it involves a dialog between a circulation desk employee, the customer and the circulation desk information system.

Due to the atomicity requirement, some would-be “transactions” are not transactions according to the above definition. A famous example in this respect is a holiday booking [126]. At the highest level of aggregation this is a single transaction between a customer and a travel agency in which the agency books a holiday for the customer and the customer pays for this service. The trouble is that this “transaction” would take place *before* it is known whether it can be performed. The holiday booking “transaction” consists of other transactions like *reserve flight*, *book hotel room*, *take out a travel insurance*, etc. Some of these transactions occur after the contract between the customer and the travel agency is signed and all of them must occur if the holiday booking is to occur. But some of these

transactions may fail and any failure causes an attempt to roll back the holiday booking “transaction”. However, other transactions may already have been committed, so that the holiday booking “transaction” cannot be rolled back. But this leaves the *holiday booking* “transaction” in a state where it cannot be committed and cannot be rolled back, and so it does not satisfy the criteria for atomicity of a transaction.

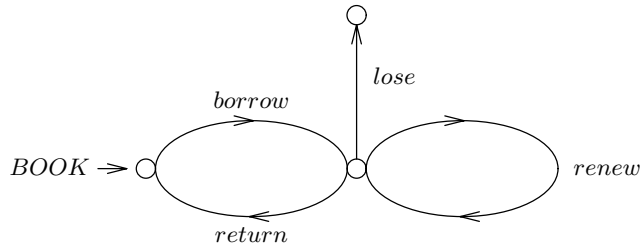
This dilemma can be resolved if we realize that the holiday booking contract is a *promise* by the travel agency to book a holiday and a *promise* by the customer to pay for the holiday once the preconditions for the booking are satisfied. This mutual promise is a transaction in our sense of the term that can be rolled back as long as it is not committed, and that, once committed, cannot be rolled back. After this mutual promise is made, a number of other transactions are performed in fulfillment of this promise. When all required transactions have taken place, the actual holiday booking transaction can take place. All these transactions take place at the same aggregation level.

### 2.4.3 System behavior

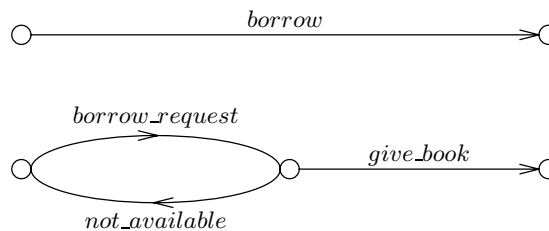
The **behavior** of a system is the way in which the system interacts with its environment over time. Here are some examples of system behavior of a discrete system, a hybrid system, and a continuous system:

- When you buy furniture in a furniture store, a certain protocol is executed, which consists of viewing different pieces of furniture, requesting information, selecting a piece, ordering it, making a down payment, etc. There is a certain temporal structure to this process that goes beyond a mere listing of the set of interactions that take place. It is for example impossible to make a down payment before having selected a piece. The detailed structure of organization behavior depends among others on company policy.
- A system that controls barriers at the entrance of a parking garage interacts with the systems that it controls following a certain protocol. First a car arrival is sensed, then it is checked whether the garage is full, then whether this car is permitted to enter, then the barrier is opened or the car is given a message that it cannot enter, etc. Again, there is a protocol in this behavior that is not visible from merely giving the set of interactions.
- When a sailing boat with its sails up catches wind, it responds by moving forward. The direction of movement depends upon the position of the rudder, the direction and force of the flow in the water, and certain properties of the boat.

Part of the behavior of discrete systems can conveniently be represented by transition diagrams such as the one shown in figure 2.5. A small arrow points to the initial state of the behavior, the arrows represent transactions and the nodes represent states. Transition diagrams can be used to represent **properties** of system behavior. For example, the book transition diagram in figure 2.5 represents regularities that can be observed in the occurrence of system transactions. Some properties represented by figure 2.5 are that a *lose* can only occur after at least one *borrow* occurred, and that after *lose*, nothing else can happen to the book.



**Figure 2.5:** A simple book transition diagram.



**Figure 2.6:** Implementation of the *borrow* transaction.

Transition diagrams can also be used to represent the implementation of a transaction at the next lower level of aggregation. For example, figure 2.6 shows the implementation of the *borrow* transaction into the three lower level transactions of figure 2.2.

Transition diagrams are not always the best kind of notation for behavior representation of discrete systems. For continuous systems they are totally inadequate and we should use other kinds of notation systems. For example, in order to represent the observable continuous behavior of a ship, we may draw a graph of the speed of the ship as a function of the speed and direction of wind, or we may draw a stylized diagram of its rudder and include a vector diagram of the effect of the forces exerted on it, etc. It is one of the challenges of engineering to find ways to communicate system behavior to other engineers, to designers, to users and to customers.

#### 2.4.4 System properties

A **system property** is an aspect of system behavior. Stated more abstractly, a property of a system is the contents of any true proposition about observable system behavior. In software engineering, software product properties are often called *attributes* or *quality attributes*. We will not use this terminology, because the term “attribute” is used in Entity-Relationship models to indicate properties of a restricted class of entities, viz. entities represented by a software system, rather than properties of the software system itself. Note that system properties are always aspects of system *behavior*. Here are two examples.

- An industrial product may be required to have a certain size, color, strength, stiffness etc. All these properties summarize an aspect of product behavior. They tell us something about the behavior of the product in certain circumstances. When the size is measured, we should observe a certain value, when light is applied to the product,

we should observe a certain color, etc.

- A tree can be tall, old, wide, straight, etc. Again, the meaning of these properties is that, when certain operations are applied, certain observations will follow.

This view of properties implies that all properties of a system are, at least in principle, observable. A “property” that does not make an observational difference is not, in our terminology, a property.

Often, it is difficult to indicate an experiment in which we can observe a property. For example, software products may be required, among others, to be user-friendly, reliable, portable, flexible and maintainable. All of these properties summarize some aspect of software product behavior, but which aspects do they summarize exactly? Is a product user-friendly if novices learn to use it in a short time, or, alternatively, if there are few help requests? Is it portable if it can be easily ported to many kinds of systems? What is the appropriate measure of easiness here? Is a software product maintainable if it is well-documented or if it is modular? Are there unambiguous measures of modularity?

We will say that a property is **specified behaviorally** if an experiment has been specified that will tell us unambiguously whether a system has the property. If no such experiment has been specified, then the property is specified **nonbehaviorally**. If a property is specified behaviorally, then it is always possible to define a *transaction* in which the property is observed. This transaction is just the successful completion of the experiment in which the property is observed. But then each behavioral property specification indicates some kind of transaction that the system should be able to perform. For example, if we want a system to be fast, we can specify that the average response time per day should be 0.1 seconds and never exceed 2 seconds, etc. Often, the experiment/transaction in which a property  $P$  is observed requires the observation of a set of other transactions. Observation of these other transactions is then a *precondition* of the observation of  $P$ .

In software engineering, behaviorally specified properties are often called **functional properties**, and properties for which no observation procedure has been specified are called **nonfunctional properties**. This terminology is misleading, for it transfers a distinction of *specifications* of properties to the properties themselves. We have just seen that all *properties* are behavioral, for they all summarize an aspect of system behavior. However, a property *specification* may be nonbehavioral, because we cannot define an observation procedure for the property.

Nonbehavioral property specifications give information about a system, because groups of people may agree on the presence or absence of these properties without using experimental evidence that unambiguously settles the question whether a system has the property. There is agreement among large groups of people about the beauty of Beethoven’s sonatas, the quality of Rembrandt’s paintings, or the user-friendliness of some software product interfaces. Typically, there are other groups of people that vehemently disagree about the presence of these qualities in these products. There is nothing wrong about this. It just shows that the market for these products is not homogeneous. During the development of a product, nonbehavioral specification of some important properties gives important marketing information, and it can motivate developers to build certain useful properties into the product.

If a behavioral specification of a property cannot be found, it may be useful to give behavioral specifications of *different* properties, whose presence indicates the presence of

the nonbehaviorally specified property, but which are not equal to it. These different properties are then treated as **proxies** of the intended property. For example, proxies of user-friendliness are “easy to learn” and “easy to use”, which can be specified behaviorally in terms of, say, average time needed for training and average number of help requests. The term *proxy* is used to indicate that the behaviorally specified properties are a substitute for the intended property, and that they approximate the intended property but are not equal to it.

## 2.5 System Function

So far, we have said that any system has a boundary that separates its internal structure from its externally observable behavior. In this section we turn to an aspect of system behavior present in some but not all systems: the function that a system can have for its environment. A system **function** is here defined as a *service* provided by the system to its environment. Here are some examples of systems that have a function for another system.

- The function of the liver for a vertebrate is to transform certain substances in the blood; the organism needs this for its sustained existence.
- The function of an elevator for its users is to transport them from floor to floor; the reason for the existence of elevators is that people do not like walking the stairs.
- The function of an information system for an organization is to provide members of an organization with useful data; they need this data to be able to communicate with each other effectively. The reason that information systems exist is that organizations cannot continue to exist for long without a properly functioning (manual or automatic) information system.

We will use the term “function” also in the derivative sense of *useful behavior*. The determination of required product functions is then the determination of useful product behavior.

A system that has a function for its environment provides for a *need* or *desire* in its environment. The liver answers a need of its environment, and the elevator answers a desire of its users. The needs of people may differ from the desires of people: managers may desire an information system, but perhaps what is really needed is a change in organizational procedures. In product development, the developer and the client should reach agreement about the client’s needs such that what the developer perceives to be the client’s needs is indeed what the client desires. In this book, we therefore define a **need** as the lack of something *desirable*.

If a system has a function, then there is always some system in its environment whose needs (desires) are answered by interacting with the system. We call this system the **user** of the system. The function of a system is thus always relative to a user and a need is for us always a **user need**. The same system can have different functions for different users because these have different needs. If we take away all users, then the system does not have any function.

### 2.5.1 Products

Outside the realm of biology, most natural systems have no function in the above sense. For example there is no environment that has a need which is fulfilled by the planet Jupiter.

In the universe of physics, things just exist and behave. In the universe of social systems, however, some natural systems occur in contexts where they have a certain function for man. For example, the sun provides light and warmth to living beings on earth, and a mountain can provide aesthetic pleasure to tourists. In the realm of biology, we find many cases in which one system (e.g. an organ) has a function for another (e.g. an organism).

In system development, we are not interested in natural systems with a function, but in artificial systems with a function. We call these systems **products**. A product has two properties:

- it is manufactured, built, developed, or otherwise created by man, and
- it is created in order to provide a function to its user.

Thus, like the liver, the reason for existence of a product is the provision of a service to its environment; but unlike the liver, it is manufactured by man in order to provide this service. According to this definition, a coffee machine is a product built to provide its users with coffee, and an information system is a product built to provide an organization with useful data.

We call the most general function of the product the **product idea**. For example, the product idea of an elevator is that

upon request, it should transport people from floor to floor.

The statement of a product idea should be succinct and clear. Generally, a clear product idea leads to well-defined system boundaries and hence to modular systems (figure 2.1).

If any system created by man to provide a service to its environment is a product, then a business is a product, too. The users of the business are called *customers* and the function of the product is called the **mission** of the business. The mission statement of a business says “who we are and what business we are in”. For example, the mission of a hospital is

the care and treatment of the ill.

Mission statements, like product ideas, should be succinct. Like products, implementing an organization according to a simple coherent underlying idea enhances the modularity of the organization and increases the regularity and simplicity of its behavior.

A product idea or business mission can be decomposed into **objectives** that the product or business must realize. Each business is managed in such a way that these objectives are reached (or so one hopes). Each business therefore displays the purposive behavior that some people take to be characteristic of systems in general. In this book, we take a more general view of systems. Thus, products like information systems or coffee machines do not engage in purposive behavior but nevertheless are viewed as systems. Note that the realization of product objectives is the task of designers and maintenance personnel.

## 2.5.2 Functions of computer-based systems

Because computers are data manipulation systems, computer-based systems provide functions in which data manipulation plays an essential role:

- **Computation.** Computer-based systems can perform complex computations faster and with less errors than we do. This is useful for us.

- **Registration.** Computer-based systems often register data. This is useful for us because we need an external memory to supplement our own fallible memories. Moreover, law often prescribes that we keep external records of business transactions.
- **Communication.** Because a computer-based system allows us to read data that has been written earlier, it allows asynchronous communication between people at the same location at different times (information systems). This can be extended to synchronous or asynchronous communication between nodes at geographically different locations (EDI systems).
- **Control.** Embedded systems often send commands to machines in their environment, to make them behave in a certain way that is useful for us.

Traditionally, computer-based systems are classified as oriented towards one of these functions. In **computing-intensive** systems, computation outweighs registration and control. An example is a simple pocket calculator. Another example is a decision-support system that performs linear programming or other optimization tasks, or that contains an economic model to be used for computing what-if scenarios. In **data-intensive** systems, large amounts of data are registered and relatively simple computation and control functions are performed. Classical examples are database systems and transaction-processing systems. In **communication-intensive systems**, a set of nodes performing relatively simple computations and having little memory is connected by a network with heavy traffic. An example is a telephone system. In **control-intensive** systems, part of the environment is controlled but relatively simple computations are performed and few data are stored. Examples are control software for an elevator system, a cruise control system, or a chemical processing plant.

With the advent of telematics systems, EDI systems and integrated manufacturing systems, the borders between these systems disappear. All these systems have more control functions than classical data-intensive systems, manipulate more data than classical control-intensive systems, and contain more communication than the other kinds of systems.

## 2.6 The Why, the What and the How

A specification of observable product behavior is a specification of *what* the product does, a specification of the product function is a specification of *why* it does this, and a specification of its implementation is a specification of *how* it does this. The major concern in these three specifications is that of utility. **Utility** is a relation between something that has a use and something else that benefits from this use. There are two utility relations to be considered (figure 2.7).

- The **service** that a product delivers to its environment is the benefit that the environment has from the product. Service is a relation between the behavior of the product and the needs of the user.
- The **correctness** of the implementation is the degree to which the implementation actually realizes the required behavior of the product. Correctness is a relation between a specification of observable behavior and an implementation.

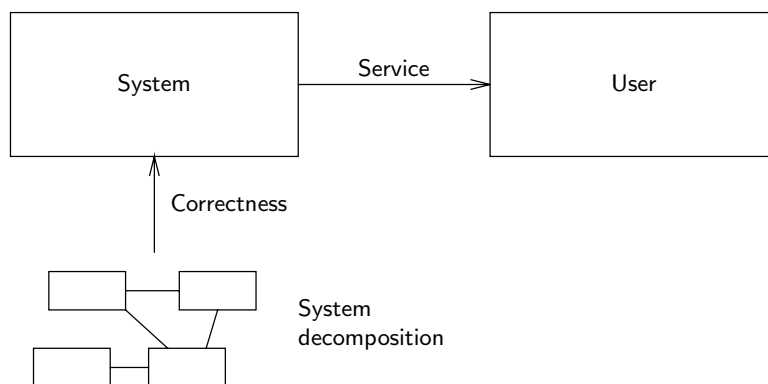


Figure 2.7: Correctness and service.

If we consider the function, behavior and implementation of a product, then behavior is central in this triad. Looking at figure 2.3, if we start at any level of aggregation and ask *what* the product does at that level, we ask for the behavior of the product and if we ask *why* the product behaves as it does, then we move upwards in the hierarchy. If we ask *how* the product behaves as it does, then we move downwards in the hierarchy. For example, the mission of a business may be to provide its customers with a certain product. The function of an information system in this business is then to provide the data required to fulfill this business mission, and the function of a database system within this information system is then to accept updates and queries required to fulfill this information system function. Moving further downwards, the function of database programs is to fetch and store data on disk as needed to fulfill the database system function, etc.

A consequence of this is that the *reason* for certain behavior becomes visible when we move up and becomes invisible when we move down in the aggregation hierarchy, and that the *implementation* of a behavior becomes invisible when we move up and becomes visible when we move down. Of course, invisibility does not mean absence. The highest level function to which the behavior of a low-level system component contributes is still very much present. This presence makes itself felt when someone asks whether the utility of a low-level component is still justified by the cost of maintaining the component. If in our daily work, we are concerned mostly with building and repairing these low-level components, then such a question disrupts our unquestioning acceptance of what we are familiar with and forces us to travel in the *why*-direction of the aggregation hierarchy, i.e. upwards. Conversely, the lowest-level components that implement a high-level function are present even though they are invisible to system users. This presence makes itself felt when a low-level component breaks down and our high-level user is confronted with low-level error messages and other incomprehensible system behavior. Such behavior disrupts the unquestioning acceptance of the normal flow of events for the user. Fixing erroneous system behavior always requires traveling in the *how* direction of the aggregation hierarchy, i.e. downwards.

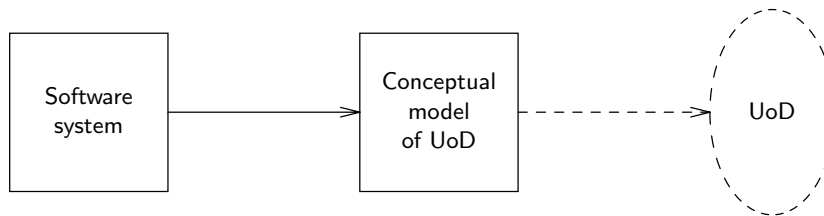
To show that an implemented product is useful, we must show that the two utility relations are present. Thus, we should show that the behavior of the product is useful for its users and that the implementation of this behavior is correct. If an argument for the quality of the product ignores one of these two relations, then this argument does not prove

its claim. In order to produce such an argument, it is important that links are maintained between related specifications at all levels in the hierarchy such that each specification is linked to specifications (at higher levels) that justify it and to specifications of components (at lower levels) that implement it. A set of specifications for which these links are maintained is called **traceable**. (Although *traced* would be more appropriate, we follow accepted terminology here.) Behavior specifications should be traceable to specifications of user needs and system objectives on the one hand and to specifications of subsystems on the other. **Forward traceability** of a specification is the property that each part of the specification can be traced to specifications of lower-level components that implement it. Forward traceability requires maintenance of links in the *how*-direction, which is down the aggregation hierarchy. **Backward traceability** of a specification is the property that for each part of the specification, it is clear why it is included. This is the maintenance of links in the *why*-direction, which is up the aggregation hierarchy. Backward traceability ensures that the impact of changing a component of a specification can be traced to changes in product behavior and in the service that the product delivers to its user. The realization of traceability is particularly important when user needs, product objectives, behavior specifications or system decompositions are changed and the impact of such a change must be estimated.

## 2.7 The Universe of Discourse of Computer-based Systems

Computer-based systems contain software systems, and software manipulates *data*. Now, the essence of data is that they *mean* something. Data items are symbols that, for their readers and writers, refer to something. The part of the world to which the manipulated data refers is called the **universe of discourse** (UoD) of the system. The UoD of a system is sometimes called the *domain* of the system, but we will not use that term in this way. We give some examples of computer-based systems and their UoD.

- The universe of discourse of an order administration system contains suppliers, ordered material, orders, deliveries, etc.
- The universe of discourse of an elevator control system consists of buttons, requests for service, elevators and their movement, lights that can be told to switch on and off, doors that can be told to open and close, etc.
- The universe of discourse of an EDI system connecting different businesses in a harbor consists of cargo to be moved, container shipments, insurances, payment obligations, etc.
- The UoD of a pocket calculator is the abstract, mathematical world of arithmetic. The pocket calculator registers no events that occur in its UoD, nor does it send control signals to its UoD, because nothing happens in its UoD. Its function is to indicate relations between elements of its UoD.
- A decision support system is an information system that registers events in its UoD, and that contains an econometric model of the UoD. It registers no events but com-



**Figure 2.8:** The UoD and its conceptual model.

putes a number of relations that exist in this abstract, mathematical model. For example, it can compute possible futures according to this model,

In order to achieve a shared understanding of the meaning of data manipulated by a computer-based system, users, developers and builders of the system must have a shared model of the UoD of that system. The function of this model is to act as the semantic structure in which the manipulated data are interpreted by all relevant people. The model must be understood by the users of the system, because they would otherwise misunderstand the meaning of the data, and it must be understood by the developers and builders of the system, because they must know what are the meaningful data manipulations that they have to make the system perform. In addition, to avoid misunderstandings about the meaning of the data manipulations of the system, users and developers must have the *same* understanding of the model of the UoD. To emphasize the important role of the model in the understanding of the behavior of a data manipulating system, we call it a **conceptual model** of the UoD of the system. A conceptual model of the UoD is an abstract representation of the behavior of the UoD, understandable and understood in the same way by the users and developers of the system. The term “conceptual” refers to the fact that the model consists of concepts by means of which users and developers interpret and observe the UoD. Put more prosaically, a conceptual model of a UoD is a pair of spectacles by which people can observe the UoD. Figure 2.8 illustrates the situation.

Later, we will also look at conceptual models of the system under development (SuD). The hallmark of conceptual models is that they are conceptual structures used as framework for communication between people. The communication may be about a UoD, a computer-based system, a software system or any other kind of system.

## 2.8 Summary

In the systems approach, the world consists of building blocks called *systems*. Any actual or possible observable part of the world is a system. Any observation of the system is an interaction between the system and its environment, and conversely any interaction with the system is an observation of that system. The system *interface* or *boundary* is the set of all possible interactions of the system and the *environment* of the system is the part of the world it interacts with. Each system has a *state space*, where a *state* is the memory the system has of past behavior. A system is *discrete* if its state space is a discrete set and *continuous* if its state space is a continuous set. It is *hybrid* if a projection of its state space is a discrete set and another projection is a continuous set. A *transaction* is a smallest interaction between a system and its environment, i.e. an interaction that has

no intermediary states. Whether or not an interaction is considered to have intermediary states is a matter of choosing a level in the aggregation hierarchy at which to observe the system.

A system is called *modular* if its boundary has been chosen in such a way that there would be less regularities in its behavior, and more complexity, with any other choice of boundary. In addition, there should be much cohesion within the system but loose coupling between systems.

The *structure* of a system is the way it is composed of subsystems and the *behavior* of a system is the way it interacts with its environment. Often, some of this can be represented by transition diagrams, but in most cases, other kinds of notations must be used additionally. A *property* summarizes an aspect of system behavior. All properties are observable, but some properties may be hard to specify behaviorally. For these properties, it may be useful to specify *proxies* for which behavioral specifications can be given.

A *function* of a system for another system is a service that the first system delivers to the second. System developers develop *products*, which are systems that are built to provide a service to their environment. The reason for existence of a product is (or should be) that it has a function for another system. The basic function of a product should be describable as a *product idea*.

Computer-based systems have at least one of the functions of computation, registration, communication, and control. The traditional division of computation-intensive, data-intensive, communication-intensive and control-intensive systems tends to be blurred when we move to EDI, telematics and manufacturing control systems.

System structure, behavior and function cover the *how*, *what* and *why* perspectives from which we can view a system. There are two utility relations between these views: system structure must be *correct* with respect to required system behavior, and the behavior itself must deliver a *service* to the users. A collection of system specifications at different levels of aggregation is called *traceable* if each part of each specification is linked to specifications of the components that implement it (*forward traceability*) and to the justification for including this part in the specification (*backward traceability*).

Each computer-based system manipulates data that have meaning in a *universe of discourse* (UoD). A *conceptual model* of the UoD represents the shared interpretation that users and developers should have of this data.

## 2.9 Exercises

1. A coffee grinder is connected to a power plug, has a reservoir for coffee beans, an on/off switch, and a light that indicates its on/of status.
  - (a) What is the function of the grinder?
  - (b) Describe the interface of the grinder, including the systems with which it interfaces.
  - (c) Is the grinder a discrete, continuous or hybrid system?
2. The interface between a grocery store and its customers can be described at several levels of aggregation. At the highest level, there is a simple interface with only one transaction, *sell grocery*.

- (a) Decompose this high level transaction into lower level transactions and decompose one of lower level transactions again into still lower level transactions.
  - (b) At each of the three levels of aggregation, the grocery store has another interface. Some observers are interested only in the highest level, some in the next lower level, etc. For each level, give an observer who is interested in observing the system at that level.
3. An observable system state may be an equivalence class of internal system states whose difference is not observable. For each of the following systems and observers, give an example of an internal system state that is not observable by the observer. State the reason why you think that the internal state is not observable.
  - (a) A coffee machine and its user.
  - (b) A text editor and its user.
  - (c) A fashion store and a potential buyer.
  - (d) A database system for the library circulation desk, and its user.
  - (e) A database system for the library circulation desk, and a library user.
4. Figure 2.4 shows a series of communications between systems that occurs when a giro payment is made.
  - (a) the communication between the banks and their customers takes place through the postal services, and the communications between the banks and the clearing house takes place through a special transport company specialized in armed transports. Add these systems to the communication diagram.
  - (b) Describe the process that a payment goes through by means of a transition diagram. First, draw the diagram of the normal process and next, at each stage, add abnormal events by which the *pay by giro* transactions may go wrong. For example, the payment order may get lost in the post, etc. Finally, take care that the whole payment process implements the *pay by giro* transaction by adding, where necessary, rollback actions.
5. Suppose an artist uses a hammer as part of a piece of art. Is the hammer then still a product?
6. We can represent finite state machines by a transition diagram, but also by a regular expression. For example,  $a.(c.b+c.d)$  is a regular expression in which the dot represents sequence and the plus represents choice.
  - (a) Draw the transition diagram corresponding to this expression.
  - (b) Draw the transition diagram of  $a.c.(b + d)$ .
  - (c) Do both diagrams represent the same behavior? Explain your answer.

## 2.10 Bibliographical Remarks

**The systems approach.** Two important roots of the systems approach lie in biology and in engineering. In the 1920s, the biologist Von Bertalanffy argued that an explanation of the functioning of biological organisms required a level of explanation above that of physical and chemical processes because biological organisms persist for some time under sometimes adverse circumstances [32]. According to Von Bertalanffy, this goes counter to the second principle of thermodynamics, which says that any state change in a closed system decreases the energy available for work in the system. He maintained that biological organisms must be seen as systems that interact with their environment with the purpose of extracting energy from the environment to sustain themselves. The concept of a system is thus connected to the idea of purposive behavior.

In the 1940s, the mathematician Norbert Wiener studied goal-seeking mechanisms that could be used to improve automatic radar, and this led to the idea of feed-back loops to implement purposive behavior in machines [144]. Wiener termed the study of goal-directed systems *cybernetics* and published a book about this in 1948 [361]. In engineering, too, the concept of a system has been connected with the idea of purposive behavior. The connection of systems theory with feedback loops and purposive behavior has been ignored in this chapter because it is not necessary to understand the structure of product development. Checkland [63] gives a grand vision of the history of the systems approach.

**The systems approach in computer science.** An important application of the systems approach in computer science, especially of the idea of modularity, is the method of *structured analysis and structured design* [84, 378]. Structured analysis is discussed in chapter 10. General systems ideas in this approach go back to work by G.M. Weinberg [357], C.W. Churchman [67] and Ross and Schoman [294]. Another important application of the system approach is Checkland's *soft systems methodology* [63, 62].

**Automata theory.** The classical definition of the concept of a state of a deterministic finite-state system is given by Minsky [233, pp. 13 ff.]. A state concept that is useful in the context of communicating systems is given by Milner [231, 232]. Milner defines various notions of equivalence on transition diagrams, such that two equivalent diagrams represent the same observable behavior. Another good survey of equivalence notions can be found in Baeten and Weijland [18].

**Modularity and complexity reduction.** The building blocks view of the world which pervades systems theory is primarily a method of complexity reduction. Descartes had already advocated this method in the seventeenth century. To understand an object, he said, we must decompose it into its parts and repeat this process until we reach a clear understanding of the simplest building blocks. Then we recompose to achieve an understanding of the whole [85, pages 15–20].

In addition to being indispensable as a method to *understand* a complex system, reduction to simpler subsystems is advocated by Simon [314] as the only way we can *build* a complex system. By reducing a complex system to simpler subsystems, we can define intermediary products that have a stability of their own and can be used in several ways at higher levels. *Structured programming* uses the same idea [75] to reduce the complexity of a

program to manageable chunks, and, as said before, *structured design* [378] and *structured analysis* [84] took these ideas to earlier stages in the system development process. The same idea is central in structured computer organization [337] and the leveling of computer systems in general.

**Modularity heuristics.** The requirement that a system must have an underlying idea that defines its coherence is common in industrial product development [289]. Lawson [190] emphasizes the importance of an underlying product idea for successful software system design. Tully [348, pages 57–60] expresses the same idea. The close cohesion and loose coupling heuristics are well-known from Structured Design [251, 378]. The heuristic that a change inside a system should cause a minimal change outside the system is given by Page-Jones [252]. Parnas' [253] heuristic that each module should hide a design decision (so that changing the decision will not affect other modules) is special case of this.

**System behavior and properties.** The concept of a transaction is well-known in database research. A readable introduction, which puts the concept in historical perspective, is given by Gray [126]. A practical introduction to the specification of required system properties is given by Gilb [115], who stresses observability of the properties in experiments. Gause and Weinberg [111] is a treasure island full of insights and heuristics in the specification of required system properties. “Nonfunctional properties” are treated extensively by Vincent, Waters and Sinclair [350]. A brief introduction is given by Ould [250]. Other useful sources are Davis [77] and Yeh [374].

**System levels.** The idea of system leveling is central in general systems theory. Boulding [49] and Checkland [63] published two influential hierarchies. The idea of leveling is coupled with complexity reduction and modularity, as well as with the concept of emergent behavior [346]. According to this idea, system behavior cannot be explained merely by the behavior of its components, but must be explained by referring to the way in which these components are put together. It is also the key to the disentanglement of the confusion about the difference between requirements and implementation. Davis [77, pages 17–18] gives a very clear explanation of this. Hatley and Pirbhai [141] use it as a basis for their structured requirements specification and system decomposition method.

**System function.** The definition of system function as service for the environment is taken from In't Veld [349]. A very good introduction to system concepts, including that of function, is given in the first few chapters of Katz and Kahn [170].

**The UoD.** The concept of UoD was introduced by the 19th century logician De Morgan to stand for the whole of some definite category of things under discussion [181]. It is central in the ISO report *Concepts and Terminology for the Conceptual Schema and the Information Base* [127] and to the NIAM method [244].

**Conceptual models.** The term “conceptual model” is often defined vaguely as a set of concepts from a given reality that can be used to describe a set of data and manipulations to operate on those data [25, pages 6, 26]. In this meaning, the Entity-Relationship (ER)

approach to data modeling is a conceptual model. Sometimes, conceptual models are identified with ER-like models. We will not follow this usage in this book. Loucopoulos and Karakostas [198] define conceptual models as cognitive structures used for the purposes of understanding and communicating aspects of the physical and social world around us. This usage of the term arose in the field of data modeling but has a wider applicability than just ER modeling [51]. In this book, we use the term in this sense.

## Chapter 3

# Product Development

### 3.1 Introduction

In this chapter, we define product development as a rational problem solving process. The problem to be solved by product development is that there is a need to be met; the solution delivered is a specification of a product that would meet this need. Section 3.2 distinguishes client-oriented development from market-oriented development and identifies the tasks in these two kinds of development. Once a product is used, the needs of its users evolve and hence a point may be reached where the product must be adapted to meet these changed needs. In section 3.3, we identify product evolution as a nonterminating alternation between product (re)development and product use. An important characteristic of product evolution is that the effects of a product on the satisfaction of the client or consumer are evaluated *after* the product is built and delivered. In section 3.4, we contrast this with product engineering, in which design choices are made based on an evaluation of likely effects of alternative product designs *before* any of the specified designs is implemented.

In chapter 2, we saw that computer-based systems manipulate data that has a meaning in a UoD. During the development of a computer-based system, developers must make a descriptive model of the UoD. The method to find such model is the empirical cycle, discussed in section 3.5. The empirical cycle is an application of the rational problem solving method to find an appropriate model of the world. This is the mirror image of the engineering cycle, which is an application of the rational problem solving method to find a proper intervention in the world.

In section 3.6, we combine the engineering cycle with the hierarchy of system levels identified in the previous chapter to yield a simple framework within which we can place the requirements engineering methods reviewed in part II.

### 3.2 Client-oriented and Market-oriented Development

**Product development** is a process in which a specification is delivered of a product that would satisfy an identified need (figure 3.1). For example, the development of a new kind of car ends in a specification of a car design that would satisfy the needs of a certain niche in the market. *Implementation* of the product specification lies outside the development



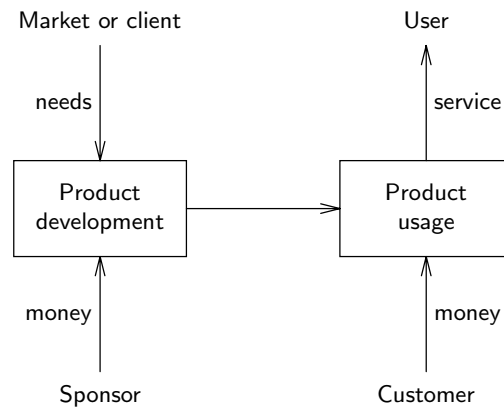
**Figure 3.1:** The essence of development is the transformation of a need into a solution that is expected to satisfy this need.

process so in a way, product development would more appropriately be called product *specification* development. However, we will stick to the shorter expression. We will use the terms “system development” and “product development” as synonyms, because in this book, all system specifications are product specifications.

The starting point of a development process is a need in the environment of the future product. This need can exist in an individual user or in a market. In the first case, we call the development process **client-oriented** and in the second case, we call it **market-oriented**. The **client** has a desire for a product and as we will see below, the client may also pay for development, or pay for the product, or use the product, or all of these at the same time. Examples of client-oriented development are designing a suit for a client, designing a kitchen to be installed in a house on assignment of the inhabitant, specifying an information system for a business, and specifying control software for a nuclear reactor system. Examples of market-oriented development are designing a new type of vacuum cleaner, developing a new type of printing equipment, developing a new version of a word processing package and developing control software for a type of elevator system. Client-oriented and market-oriented development are distinguishable from each other in a number of ways.

- In client-oriented development, the need that starts the development process is *experienced by the client*. Because the client experiences it to exist, it exists. In market-oriented development the need is perceived by marketing specialists. There is always room for debate whether the perceived need really exists in the market.
- In market-oriented development, a specification of a product *type* is delivered, such as a type of vacuum cleaner, a type of printer, or a software package. Product development results in a description of a mass production process by which a series of instances of the type is produced. Client-oriented development on the other hand delivers an individual solution, such as the drawings for a tailor-made suit, the plans for a house, or a specification of an information system. Only one instance of an implementation is produced.
- In client-oriented development, not only the product, but also the client itself is an individual. This creates room for the client to change his or her mind during development and to demand a high degree of satisfaction of his or her needs. In market-oriented development, on the other hand, the variability of needs is much less, because these are not subject to individual whims. Consumers do not expect as high a degree of satisfaction as clients do.

There are borderline cases that can be classified in both classes. A project in which a block of houses is built is client-oriented in as far as there is a client who commissions the project, but market-oriented in as far as the client intends to sell the houses in the block on the



**Figure 3.2:** Different stakeholders in product development and usage.

market. Consequently, the need for the block is felt by an individual (the client) and is also perceived to exist in the market. Similarly, the delivered product is an individual (a block of houses) as well as a type (a blueprint used to build several houses).

There are a number of **stakeholders** that have an interest in the processes of product development and usage. We identify a number of important stakeholders in figure 3.2. Note that one person or organization can play the role of several stakeholders at once.

- The **development organization** is the organization (or person) that performs the development process. The development organization of a new car model would contain, among others, the engineers, industrial designers, ergonomists, project managers and supporting personnel who participate in the design of the car. The development organization of an information system would consist of the information analysts, software engineers, system designers, programmers, project managers and supporting personnel.
- The **sponsor** of the development process is the person or organization that pays for the development organization and its process. In the development of a new car model, the sponsor would be the company that commissions the development process. In information system development, the sponsor is the organization that pays the developers to produce an information system.
- The **customer** of the developed product is the person or organization that pays for the delivered product. In the development of a new type of car, the customer is the person who buys an individual car. In information system development, the customer is the sponsor that pays for the development process, because the sponsor owns the result of development. In general, in client-oriented development, the customer and sponsor are identical.
- The **user** of the product is the person or organization that actually uses the product. In the development of a new type of car, the users of the individual cars probably include the customer who bought the car, but most probably his or her spouse and

possibly some of their children as well, if they all drive the car. In the development of an information system, the users of the system are the employees who actually initiate the transactions of the information system.

If we include more stages of a product life cycle, we can identify more stakeholders. Examples are marketing, manufacturing, sales, distribution, and maintenance personnel, and of course operational, tactical and strategic management.

In software development, the sponsor is often called the *user* and the people actually using the software are called *end-users*, but we will not follow this terminology.

Note that in client-oriented development, the roles of sponsor and customer are identical: he or she that pays for the development (sponsor) also pays for the delivered product (customer). In turn, both are often identical to the person or organization that experiences the need for the product (client). However, there may be client-oriented developments where the customer/sponsor differs from the client who feels the need for development. An example is the development of a scheduling system for a government-owned public transport system. The client is the public that uses of the transport system, the customer/sponsor of the scheduling project is the government.

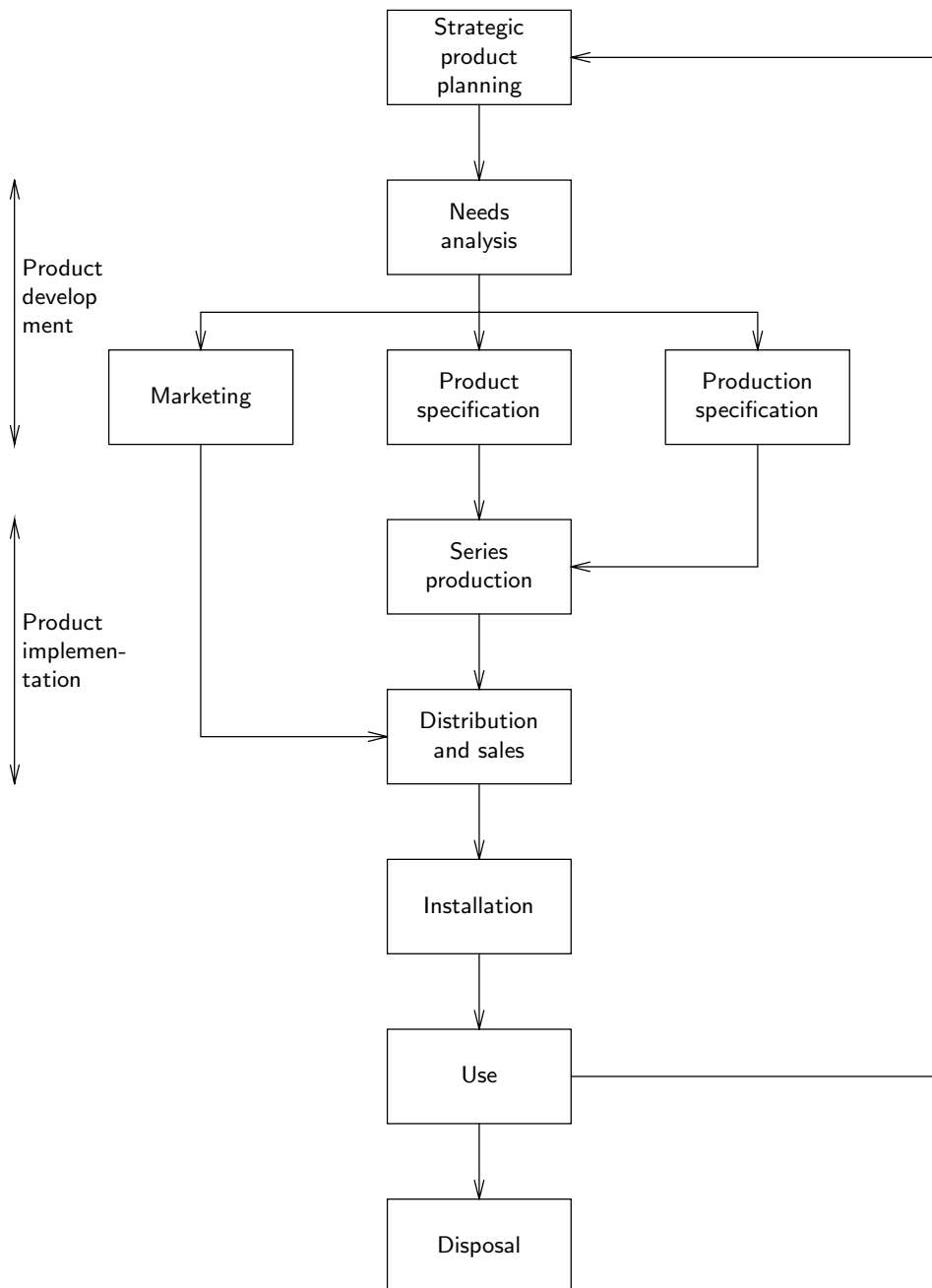
## 3.3 The Product Life Cycle

### 3.3.1 Product development

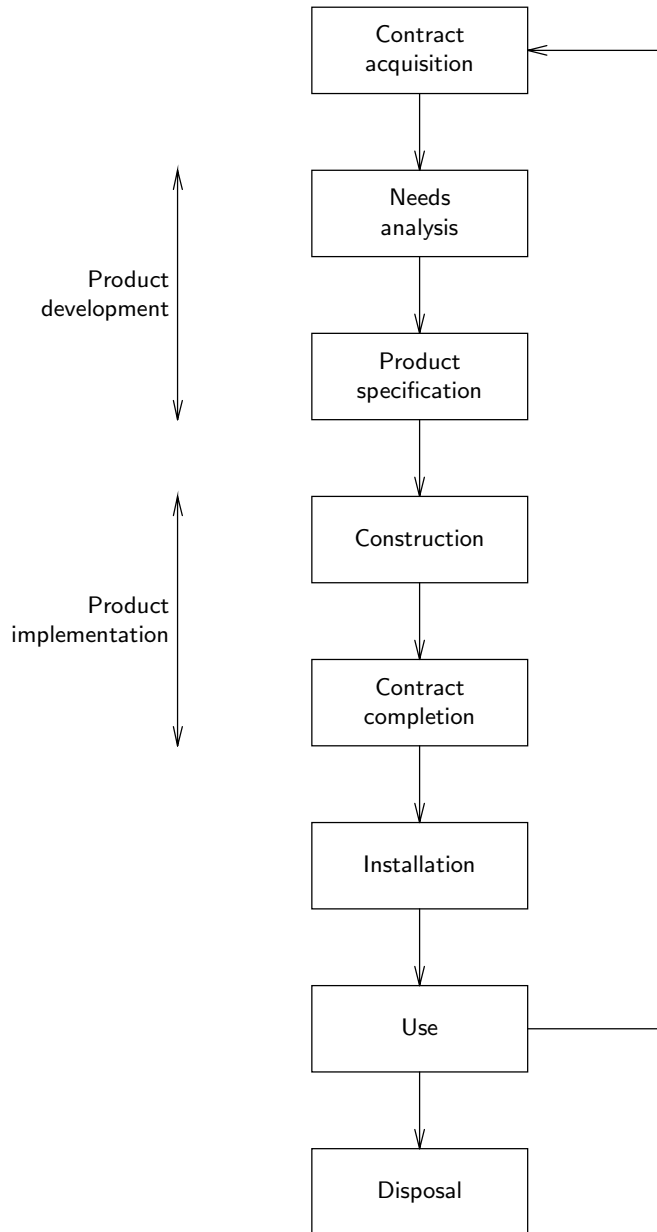
The life of a product can be partitioned into a number of stages, which are slightly different for market-oriented and client-oriented products. The stages are shown in figures 3.3 and 3.4, respectively. Market-oriented development typically follows a **strategic product planning** task, in which innovative new products or renovations of existing products are planned. Strategic product planning includes the management of the product evolution process discussed later. Using the results of strategic product planning, the market-oriented development process consists of the following tasks:

- **Needs analysis:** analyze the needs of the market and determine the objectives that the solution must satisfy.
- **Marketing:** describe the market segments to which the product is targeted and the distribution channels and outlets for the product.
- **Product specification:** describe the product to be implemented. There are two subtasks in product specification:
  - **Behavior specification:** specify observable product behavior.
  - **Decomposition specification:** specify a decomposition of the product into parts.
- **Production specification:** describe the process by which instances of the product are produced.

Each of these tasks produces deliverables, often with the same name as the task. For example, the development of a new bicycle model would result in a specification of the objectives that the bicycle should fulfill, a specification of a marketing and distribution



**Figure 3.3:** The life cycle of a market-oriented product.



**Figure 3.4:** The life cycle of a client-oriented product. Contract completion can lie anywhere between development to use of the product.

plan, a specification of the observable behavior of the bicycle in different circumstances, a specification of the decomposition into parts, and a specification of the production process that describes how the parts can be assembled along a production line. The specification of observable bicycle behavior would include such things as how the product looks and feels, its performance characteristics, its color, size and weight, etc.

Market-oriented *software* development is exceptional because production specification is nearly absent or extremely trivial compared to other kinds of market-oriented product development. Unlike other kinds of products, software can be duplicated indefinitely with very little effort. The major problem of software production for a market is the realization of portability of the software product across different hardware and software platforms. However, the solution of this problem does not require a specification of a production process, but of platform-independent interfaces of the product. There is thus no production specification to speak of for software produced for a market.

A borderline case of market-driven and client-oriented software development is the development of parametrized software, such as packages for order-processing, sales administration, personnel administration, etc. These packages are developed for a market but they have parameters that can be set to meet the individual customer's needs. Development of parametrized software should therefore include a software product specification as well as a specification of the production process in which the software is customized for an individual customer.

Turning to client-oriented development, we see in figure 3.4 that this typically follows **contract acquisition**, in which an agreement is made with a customer to produce a product. Depending upon the contents of the contract, **contract completion** can lie anywhere between development and use of the product. Figure 3.4 shows contract completion after construction and before installation of the product. Client-oriented development consists of the following tasks:

- **Needs analysis:** analyze the needs of the client and determine the objectives that the solution must satisfy.
- **Product specification.** As before, this consists of two tasks:
  - **Behavior specification:** specify observable product behavior.
  - **Decomposition specification:** specify a decomposition of the product into parts.

This yields three deliverables: a specification of product objectives, a specification of required product behavior and a specification of the product decomposition. For example, a tailor will specify the objectives to be reached by the suit he or she will make (make the person look taller, give a distinguished look, etc.) specify the observable behavior of the suit (color, look and feel, size etc.) and the decomposition of the suit into parts (material, form of the parts, etc.).

### 3.3.2 Product implementation

**Implementation** of a product developed for a *market* consists of acquiring resources for production, setting up a production facility, producing the product in series, and distributing the product according to the marketing plan. The production process consists of integrating the parts of the product into a whole according to the decomposition specification,

so that the whole behaves as required by the behavior specification. Implementing a product developed for a *client* consists similarly of integrating the parts of the product into a whole that behaves as required of the product. Depending upon the contract, implementation of a product developed for a client may also include installation of the product. For example, information systems are usually installed in an organization as part of the contract.

Products developed for a client are often implemented and installed by the development organization, although different people within that organization may perform the development, implementation and installation tasks. For example, a kitchen may be specified by a designer, and then implemented and installed by carpenters, plumbers and electricians, all working for the organization that sold the kitchen.

Software product implementation is special, because *the specification of the decomposition into parts is already part of the implementation of the software*. Software is text that is executable by a computer and this makes it possible to write a specification in an executable language. The smallest parts into which the software product is decomposed may be programming language statements, or routines from a software library, or other reusable components that are executable by computer. As a consequence, software product implementation consists of specification of a decomposition of the product into executable parts and then of integrating these parts into an executable whole. Software product development and implementation therefore overlap and software engineers usually consider integration of the software components to be part of the development task. The result of software development is then an implemented software product, not a software product specification. Contrast this with the development of a piece of custom-made hardware such as a house: the specification of the product decomposition is still a piece of text and diagrams, and implementation is a separate task in which the parts are constructed and assembled into a whole. The overlap of software development with software implementation may lead to linguistic confusion when software engineers talk with hardware engineers.

### 3.3.3 Product evolution

Almost any development process starts because some dissatisfaction exists with the current version of an existing product. Here are some examples:

- When a new type of bicycle is developed, there is experience with current bicycles. It is this experience that goes into the strategic product planning process and that leads to novel product ideas.
- Any company has a current information system, be it an automated system, a paper-based system or even a system implemented in the memory of the employees. It is experience with this information system that leads to the needs which are analyzed in the development process.
- Any elevator system has a control system, that may be implemented as a system of electromechanical relays or as software. Experience with this control system is analyzed when a new control system is specified.

In all these cases, there is a gap, small or large, between the existing system and existing needs, and over time this gap is likely to increase. There are several causes of the increase of the gap between an existing product and existing needs:

- By using the product, the user discovers new possibilities and thereby acquires new desires, that cannot be satisfied by the current version of the product.
- The environment of the product may change. For example, new laws may impose more stringent safety requirements, competitors may have introduced a more successful product, new technology may create new possibilities, etc.
- In client-oriented development, evolution may be even more dynamic than in market-oriented development, because clients tend to change their needs *as a result of the development process* (even before the product is delivered and used). When we interview someone about his or her desires, these desires are refined and new desires are discovered by the very process of talking about them. The process of change continues after we finished the interview, and by the time we return to verify our notes, the notes may have become out of date. We call this phenomenon **requirements uncertainty**.

Whatever the cause, after a sufficiently long period of time, the gap between product and needs is sufficiently large to justify the start of a new development process, oriented at reducing the size of the gap. This has been represented in figures 3.3 and 3.4 by an arrow from the use of a product to strategic product planning or contract acquisition for redevelopment.

We call the iteration through improvements of a product on the bases of experience in actual use **product evolution**. In software engineering, this process is called **adaptive maintenance** and in market-oriented product development, it is called **product innovation**. We saw that product innovation, and hence product evolution, are part of the strategic product planning process. The essence of the evolution process is that a new development is started due to an increase in the gap between existing needs and the existing version of the product.

### 3.3.4 The regulatory cycle

We may try to influence the course of product evolution by interventions that aim to steer the process in a certain direction. If we do this, the evolution process is subjected to a **regulation process**. The logical structure of the regulation process is that of the **regulatory cycle**, which is shown in figure 3.5. Note that the regulatory cycle is nothing else but a cyclic process of feedback control. It is followed by anyone who wants to achieve a desired effect by his or her actions. Managers control the primary process of an organization by observing the performance of this process, evaluating this and taking action to steer the process towards the intended business objectives. A physician analyzes the state of his or her patient, evaluates possible remedies, chooses one and prescribes it, and then observes the result.

The regulatory cycle can be used to steer the product evolution processes in a desired direction. Product specification is the planning part of the regulatory cycle. After implementation, the customer, user or marketing department evaluates the product in real use, and this may lead to an action ranging from a change request of the current product to an idea for a product innovation that is implemented in a new development process. (In this context, not doing anything is also an action.)

The important feature of the regulatory cycle is that we evaluate the effects of an action *after* the action is performed. We choose a next action to perform on the basis of an

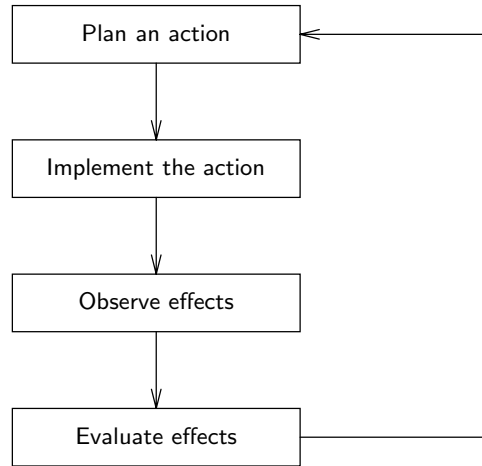


Figure 3.5: The regulatory cycle of action.

evaluation of the experience with the actual implementation. The regulatory cycle is to be contrasted with the engineering cycle, discussed next.

## 3.4 Product Engineering

### 3.4.1 The engineering cycle

In this book, we view product engineering as a species of rational problem solving. A simple model of the rational problem solving cycle is the following one:

- Analyze the problem to determine what the current situation is and what the objectives are.
- Generate some possible solutions.
- Estimate the expected effects of the generated solutions.
- Evaluate the expected effects with respect to the objectives.
- Choose a solution or go back to analysis or to solution generation.

Applying this rational method to the problem of finding a product specification, we get the **engineering cycle** shown in figure 3.6. After a needs analysis, we generate alternative product specifications, which are then simulated to discover their likely effects. These effects are evaluated and one of them is then chosen, or we return to the generation process to generate additional product alternatives. We may even return to the needs analysis task in order to learn more about the objectives of the engineering process. Note that needs analysis occurs in both the product life cycles of figures 3.3 and 3.4. The rest of the engineering cycle corresponds to the task of finding a product specification. Looking at it in another way, product specification is *solution* specification.

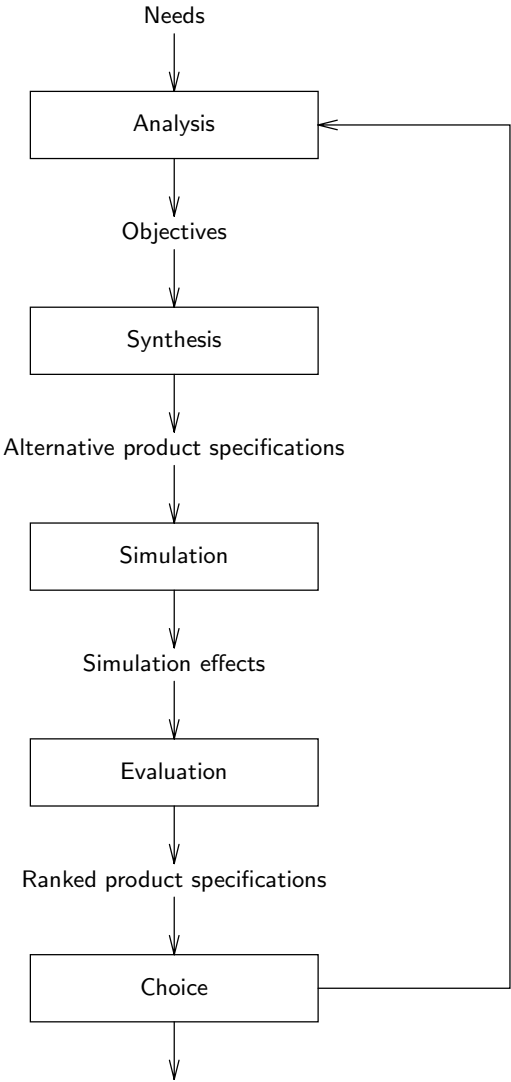


Figure 3.6: The engineering cycle.

The essence of the engineering process is that we estimate the likely effects of a possible solution, and evaluate the likely effects of implementing the solution, *before* the solution is implemented. This contrasts with the regulatory cycle, where we observe and evaluate the effects of an action *after* they have occurred. Thus, where the regulatory cycle is the logical structure of a feedback control loop, the engineering cycle is the logical structure of a feedforward control loop.

Engineering as well as regulation presuppose the existence of *regularities* in the problem domain, so that it is at least meaningful to speak of the effects of an action. Without regularities, there would be no effects but only random phenomena; and if actions would have no effects, we could not rationally choose among actions on the basis of expected effects, nor could we evaluate performed actions on their effects. The presence of regularities guarantees the existence of repeatable cause-effect relationships.

Engineering additionally requires *knowledge* of these regularities by the problem solver, so that the likely effects of intervention and non-intervention can be estimated. In the case of regulation, this kind of knowledge is not necessary, because we can perform an action and observe what its effects are without being able to predict these effects. In the case of engineering, we need this knowledge to perform *simulation* of alternative solutions before they are implemented. We take a wide view of simulation here, including simulation with scale models, by analytical computations, by deduction, by numerical approximations using difference equations, by rules-of-thumb, by observing the results of actions performed elsewhere, etc. For example, nuclear engineers who want a quick and dirty estimate of the amount of energy a power plant will generate use the rule of thumb that one gram of uranium gives one megawatt day of energy, and mechanical engineers estimate the point of failure of a wide variety of materials by the rule of thumb that the yield strength of a material is equal to a 0.02% offset on the stress-strain curve [182]. This is validated knowledge, even if the underlying physical processes may not be fully understood. Using this knowledge to estimate likely effects counts as simulation of the likely effects.

To summarize, the essence of engineering thus contains two elements:

- Separation of specification from implementation of an action.
- The prediction of the likely effects of the action from the specification, before the action is implemented.

There are domains where it is an open question whether engineering is possible, either because there are no regularities in the domain or because we have insufficient knowledge of the regularities that exist. For example, the stock market is a domain that sometimes behaves so chaotically that no regularities exist. The stock broker may try to predict the effect of buying or selling stock beforehand, but these predictions have a large margin of uncertainty. This margin is rumored to be so big that in the long run, stock market specialists perform on the average just as good or bad as random generators.

As another example, management scientists and consultants have amassed a large body of theories about aspects of organizational behavior. If these theories are true, they represent regularities in the organizational domain and they can be used as a basis for rational action. Management would then become a social engineering process, in which the manager estimates the likely effects of alternative actions and chooses an action based on an evaluation of the estimated effect of each alternative action. As a matter of fact, the estimated effects of an action often differ from their real effects and managers must decide what to

do next on the basis of an evaluation of an effect after it has occurred, rather than on the basis of an estimated effect. To the extent that this happens, organization development is an evolutionary process rather than an engineering process. Terms like “business process reengineering” must therefore be taken with an appropriate grain of salt.

Software engineering too strives for the status of an engineering discipline. This requires the ability to predict how a software system will behave *before* it is implemented. In practice, we content ourselves with observing the behavior of a software system *after* it is implemented and fiddling around with the system until it behaves as required. Most software development is *evolutionary development*, where part of the evolution takes place after construction but before installation (i.e. as testing) and part of the evolution takes place as adaptive maintenance. We return to evolutionary development in the final part of this book.

### 3.4.2 Example of an application of the engineering cycle

It is interesting to follow the engineering cycle through a few iterations in a case study of industrial product development given by Roozenburg and Eekels [289, pages 325–339].

*TANA Nederland* (henceforth TANA) produces shoe polish sold to consumers in glass jars and is a market leader in the Netherlands in this area. The problem that triggered the development process is that TANA's sales of shoe polish are stagnating. Competitors produce shoe polish with the same range of colors and TANA sees its market share decreasing.

1. According to marketing management theories, there are three possible actions to be taken in this case:
  - search for new markets with the current products,
  - develop a new product for the same market, or
  - adjust the marketing mix.

TANA management chose the second of these options for implementation. It was decided to develop a new kind of packaging for existing kinds of shoe polish.

2. A number of product ideas were generated for the packaging by combining different product and market characteristics.

Figure 3.7 shows the alternative product-market combinations that were considered [289, page 238, figure A]. To make the alternatives more vivid for TANA management, a typical example of each alternative was sketched with pencil on paper. The alternatives were discussed with management and evaluated on a number of criteria derived from the chosen marketing strategy. Product idea marked with an X was selected for further development.

3. The product idea was analyzed by following the product through its life cycle, from production to distribution, use and disposal. For each stage in the life cycle, requirements were determined. This resulted in twenty-nine design objectives, three of which are reproduced here.
  - Subassembly of parts of the packaging should be done by current suppliers of TANA.
  - The cost to store the packaging, including its contents (polish) and its bulk packaging, should not be larger than Dfl 0.95 per packaging.
  - It should be impossible that the polish leaks through the packaging.
  - The packaging should be transparent, so that the color of its contents can be seen by the consumer.

Product types	Product functions				
	Storing Closing	Storing Closing Measuring dosage	Storing Closing Measuring dosage Applying spreading	Storing Closing Measuring dosage Applying Spreading Rubbing	Storing Closing Measuring dosage Applying Spreading Rubbing Cleaning
Single use					
Refillable packaging					
Multiple use, dispose when empty			X		

**Figure 3.7:** Matrix for generating product ideas [289]. Reproduced with permission from Lemma B.V., 1991.

Next, twenty different usage concepts and twenty different implementation mechanisms were generated. A usage concept is a way in which the consumer could perform the desired product functions. Examples of the generated usage concepts are:

- application of the package to the shoe as a marker pen,
- application of the packaging as a deodorant stick, etc.

The implementation mechanisms were generated by looking at the way products with a similar function are implemented. Examples of the mechanisms that were considered are:

- a twister mechanism,
- a starlock mechanism, etc.

From the 400 possible combinations of usage concepts and implementation mechanisms, six combinations were evaluated by the developer by scoring them on the objectives. In a further evaluation by management, three combinations were selected for further development. The first selection was a package in the shape of a bulb, the second one had the shape of a flacon, and the third one was a jar with a special kind of lid with which the polish could be applied.

4. The selected packages were analyzed and three prototypes were built on the basis of this analysis. These were evaluated by presenting them to a group of consumers as far as possible in the form in which they would be marketed, and conducting qualitative interviews with these consumers. On the basis of these interviews, the third design (jar with lid) was selected by the consumers. In parallel with this consumer evaluation, the three designs were evaluated on their consumer price, cost of assembly, investment in a new production line, and some other criteria. This parallel evaluation led to the selection of the jar design too.
5. The chosen design was worked out by adding all technical details and making experimental moulds for the product, to be tested in a trial series production.

There are a number of interesting observations to be made about this example.

- The case consists of five iterations through the engineering cycle. In the first iteration, it is the *business* that is developed. In the other four iterations, a product is developed.

- In all iterations, simulation of some possible actions is performed in order to estimate what the likely effects of the action would be. For example, in the first iteration, this simulation was done by the managers by thinking through the best response to a business problem using their knowledge of the business and its market.
- Each iteration through the product engineering cycle produces a product specification of increasing explicitness. The first product engineering iteration (the second of the five iterations in the example) produces a **product idea** consisting of a combination of product type and product function. As explained in chapter 2, a product idea is the underlying idea of what the product is. Having a product idea increases the coherence and modularity of the product. The final iteration produces a technical product specification and the initial version of a production specification.
- Explicit design objectives to be satisfied by the product are only produced in the second of the product engineering iterations. They are stated in such a way that their achievement or nonachievement by a product is observable. In other words, they are stated in behavioral, operational, measurable terms.
- A final point to notice, which may be of interest to software developers, is that in the second iteration through the product engineering cycle (the third of the five iterations in the example), decisions about the decomposition of the product into components are made in *parallel* with decisions about observable behavior (the “usage concepts”).

Of course, there are some obvious differences between developing a computer-based system and developing a new kind of packaging for shoe polish. First of all, software is a major part of computer-based systems. Software is intangible and many people conclude from this that software is very flexible. By contrast, industrial products are tangible and, once they are produced, everyone can see that they are not very flexible. The problem of changing requirements, which often occurs in client-oriented development, is therefore even more widespread in the development of computer-based systems. If requirements changes become too frequent or too erratic, then the development cycle will never stop or, if it stops, it will deliver a product that does not meet client needs. This makes the management of software development considerably more complex than the management of industrial product development.

Second, computer-based systems are often closely interwoven with organizational infrastructures. For example, as observed in chapter 2, an information system can be viewed as an *aspect system* of a business. All business units interact by exchanging information with other business units and with the business environment. Viewed in this way, the information system is inextricably bound to the organizational infrastructure. The interweaving between organizational infrastructure and computer-based systems becomes even tighter with EDI systems, which require standardization of business procedures of the connected businesses. This means that the development of those systems is integrated with business development, and this in turn makes the complexity of managing the development of a computer-based system larger than the management of industrial product development.

Despite these differences, an engineer developing a computer-based system iterates through the same engineering method as an industrial designer developing a manufacturing product. The rational method of engineering in both kinds of development is the same, even though process attributes like complexity and flexibility differ. Especially noteworthy

is the interleaving of the specification of observable product behavior at one level of aggregation, and the specification of the decomposition of the product into parts at a lower level of aggregation. Above, we saw that this interleaving takes place in the development of such humble systems as a packaging for shoe polish.

One last lesson to be learned from the shoe polish packaging example is that rational product development *is* possible. The engineering cycle is a rational process that does not deal with mistakes, personnel changes, budget cuts, departmental politics and changing requirements other than by iterating through the problem solving cycle. As explained in chapter 15, these phenomena are dealt with by process management methods. We will argue there that the engineering cycle is useful as the target structure for a *rational reconstruction* of the development process, by which the results of development can be justified to others. Here, it is important to realize that the engineering cycle is the *logical* structure of development, not the *temporal* structure. Its purpose is to allow us to classify tasks in different development methods, i.e. to act as a framework for understanding development methods.

## 3.5 System Modeling

### 3.5.1 Current system modeling and UoD modeling

There are two tasks in the development of computer-based systems where the developer has to make a descriptive model of some system. First, in many cases of computer-based system development we must make a model of the observable behavior of an *existing* system. For example, in the development of a new version of an information system, we may at some time have to model that part of the behavior of the current one that must be preserved by the new one. Current system modeling is often called **reverse engineering** in computer science. This is a curious but accurate term, for we will see below that current system modeling is the exact opposite of engineering. Reverse engineering is often part of a larger process called **reengineering**, in which a current system must be reimplemented with largely the same functionality. This is an important development heuristic recommended by classical structured analysis (chapter 10).

There is a second situation in which the developer must make a model of an existing system. We have seen in chapter 2 that computer-based systems have a UoD in which their data is interpreted. Developers of computer-based systems must therefore build conceptual models of the UoD that can be understood by stakeholders such as implementors and users of the system.

Finding an accurate and useful conceptual model of the UoD requires expertise of the UoD. In some cases, this expertise is shallow enough to be learned by the system developers that are not specialists in the UoD. For example, the UoD of a library administration is simple enough to be learned by system developers themselves — but even here they may bump into surprises. Even simpler, the UoD of a pocket calculator requires knowledge of arithmetic at the primary school level and some simple mathematics at the secondary school level extended with knowledge of finite arithmetic, knowledge which any developer of pocket calculators can be expected to have. At the other extreme, a decision support system uses a model of its UoD that requires econometric expertise to build and understand. Building such a model is a profession in its own right and may take a lifetime. The development of

decision support systems must therefore involve specialists that have the required econometric expertise, as well as software engineers who can cooperate with econometrists in building a decision support system.

### 3.5.2 The empirical cycle

Whenever a descriptive model of a currently existing system or of a UoD must be made, we should follow the **empirical cycle of discovery**. Once again, this is a species of the rational problem solving cycle. Figure 3.8 shows the structure of the empirical cycle. The similarity with the engineering cycle arises from the fact that both are specializations of the rational problem solving method. The difference with the engineering cycle arises from the fact that in engineering we aim at finding a product specification, which is a *prescriptive* model of a product, whereas in the empirical cycle we aim at finding a *descriptive* model of an existing part of the world.

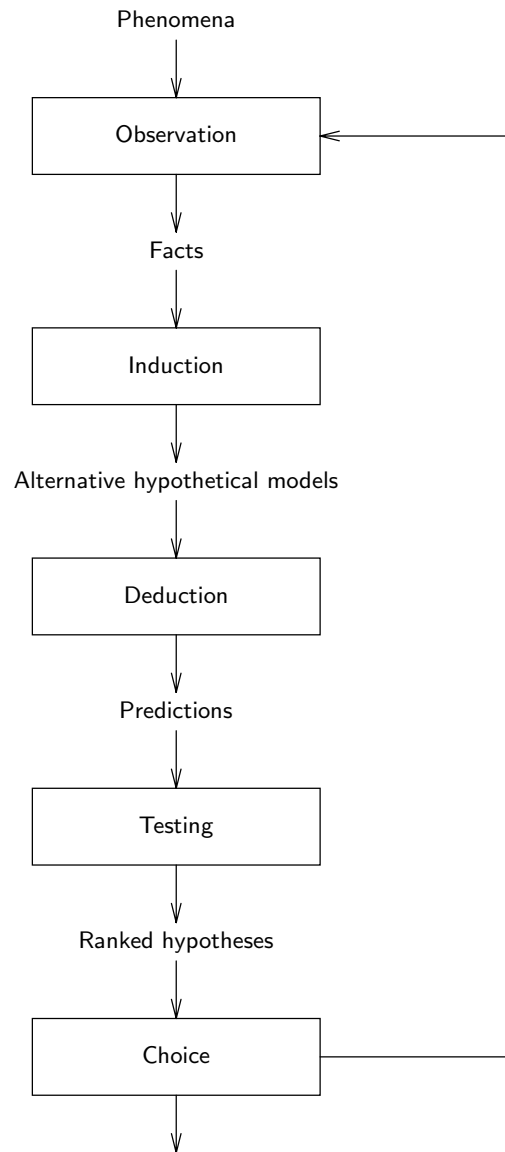
Following the empirical cycle, we start from observations and generate one or more models that explain the observed phenomena in something called an induction step. These models are evaluated by deducing observable consequences from them and testing these in experiments. As a result of these tests, we choose a model or we generate additional models that explain the phenomena. We may even return to the observation task to collect more data.

**Induction** is a reasoning form in which, from a finite number of observations, a model is specified that accounts for these and infinitely many other phenomena. For example, from repeated observation that a physical object expands when heated, we may induce that *any* object expands when heated. Induction is logically unsound because it may lead from true premises to a false conclusion. In daily life as well as in science, we must continuously jump to conclusions by inducing general models from particular observations.

The unsoundness of induction is a consequence of the fact that any finite set of observations can always be explained by an infinite set of possible models of those observations. Some of these models are true, but most of them are false. For example, if objects  $a$ ,  $b$  and  $c$  are observed to expand when heated, we can induce any of the following models:

- All objects expand when heated.
- Most objects expand when heated.
- Some objects expand when heated; others burn immediately.
- Until today, all objects expand when heated, but from tomorrow onwards they will contract when heated.
- Objects  $a$ ,  $b$  and  $c$  expand when heated but all other objects contract when heated.
- Here, all objects will expand when heated; elsewhere, the same objects would contract when heated.

We could go on indefinitely. Each of these models is a **hypothesis** that must be tested by deducing observable consequences from it, and by testing these consequences in experiments. This leads to confirmations or falsifications. There is an asymmetry between confirmation and falsification that follows from the structure of deduction. Suppose that from hypothesis  $H$  we deduce observable consequences  $O$  and we do not observe  $O$  but something else. Then



**Figure 3.8:** The empirical cycle.

we have falsified  $H$ . The reason is that if  $H \rightarrow O$ , then it logically follows that  $\neg O \rightarrow \neg H$  (here we represent logical implication by  $\rightarrow$  and negation by  $\neg$ ). On the other hand, if we do observe that  $O$  takes place, then we have not confirmed the truth of  $H$ .  $O$  follows from indefinitely other hypotheses as well, and among these other hypotheses there may very well be better explanations of the phenomena than  $H$ .

In practice, confirmation and falsification are not so far apart as suggested here. If the consequences of  $H$  are observed frequently by different people in different circumstances, then our belief in  $H$  tends to become very strong. If we strongly believe in  $H$  and once in a while one of  $H$ 's consequences is falsified, then we attribute this to unknown disturbances, measurement errors or other secondary causes, and forget about the falsification. In general, no model can be completely verified or completely falsified by experiments, and we end up with a ranking of models in order of plausibility, just as in the engineering cycle we end up with a ranking of product specifications in order of utility.

The following classical example of an application of the empirical cycle is taken from Kemeny [173].

**Example of the empirical cycle.** In the beginning of the 19th century, Newtonian mechanics was a well-established body of theory, that had been validated in numerous physical experiments and astronomical observations. Observations made in the 1820s of the orbit of the then-known outermost planet, Uranus, revealed a discrepancy between the predictions made by Newton's theory and the actual orbit of Uranus. There were several alternatives to explain this discrepancy. For example, Newton's theory could be considered refuted, or the observations unreliable. The first option is very unattractive and the second could be shown to be very implausible. Astronomers therefore generated another hypothesis, namely that Uranus is not the outermost planet, but that there is a planet beyond it. The French mathematician Leverrier showed that a planet with a certain size, position and orbit could account for the aberration between the observed and predicted orbit of Uranus. Using the mathematical techniques available at that time, this was a considerable feat. In addition, he could predict the position of this hypothesized planet in the sky. Having deduced these observable consequences, it was relatively easy for Berlin Observatory to observe the hypothesized planet in the real world, and Neptune was discovered.

Just as the engineering cycle represents the logical structure of product specification, so the empirical cycle represents the logical structure of model building. It does *not* represent what actually happens during discovery, nor does it represent how scientists think. In these processes, competition between research groups, budget cuts, personnel turnover, mistakes, blunders, blind spots and prejudices all play a role. The study of what happens during discovery is part of the sociology and history of science, and the study of how scientists think is part of psychology. The empirical cycle does however show how the resulting models must be *justified*. Scientific publications must present their results as if the empirical cycle were followed, so that others can reproduce these results, following the same cycle. In chapter 15, we refer to this as a *rational reconstruction* of the modeling process. The purpose of the empirical cycle for us is to act as a framework to understand methods for making a model of a UoD, not to help the research manager deal with such problems as budget cuts and personnel turnover. Note the analogy with the engineering cycle, which can be used to justify a product specification and to understand engineering methods, but does not represent the temporal sequence of engineering tasks.

### 3.5.3 Comparison of the empirical and engineering cycles

The empirical and engineering cycles are closely related, because both are specializations of the rational problem solving method. There are also a number of important differences between the two cycles, which all stem from the fact that in engineering we search for a specification of *useful behavior*, whereas in modeling we search for a specification of *true knowledge*. Of course, it is possible that the engineering cycle leads to new knowledge and the empirical cycle leads to new technology, but these are secondary products that do not alter the primary aim of following these cycles.

One consequence of this difference is that in the engineering cycle, we are searching for a *prescriptive* model, the product specification, whereas in the empirical cycle we are searching for a *descriptive* model. The difference is that in case of a mismatch between a prescriptive model and the modeled product, the product is wrong and the prescription is right. In case of a mismatch between a descriptive model and the modeled part of reality, the model is wrong and modeled reality is right.

Another important consequence of this is that the engineering cycle has a *normative* orientation where the empirical cycle has a *normatively neutral* orientation. The norms by which the results of engineering cycle are evaluated are instrumental, i.e. they are norms that tells us that a particular user need is to be satisfied. The justification of a product always has the form “because this human desire must be satisfied”. However, it is always meaningful to ask whether this justification itself is *morally* sound. Some human desires may be immoral in some contexts; and even if they are moral, the means by which they are fulfilled may be immoral. The desire to observe people without being observed is moral in some contexts and immoral others; and the production of cheap products by means of child labor is immoral in all contexts. Engineering cannot be separated from moral problems.

By contrast, the only norm in the empirical cycle is the norm that true models be found. Even if morally reprehensible behavior is modeled, then the norms by which this behavior is rejected do not play a role in evaluating the outcome of empirical research, which is a descriptive model. A descriptive model is true or false, not morally good or bad. The empirical researcher should *not* try to improve the behavior he or she is studying. By contrast, the engineer does try to improve the product on which he or she is working.

The empirical researcher must keep a *distance* from his or her subject in order not to disturb the behavior he or she is modeling. The engineer, on the other hand, is *engaged* in his or her subject and is committed to improving it in a shared value judgement with the client. This creates normative problems which may become moral problems. As a result, professional societies usually set up codes of conduct, which include guidelines for professional and moral behavior.

## 3.6 A Framework for Product Development Methods

A **framework** for product development methods is a conceptual structure which provides us with points of reference in an analysis of development methods. Viewing the set of all development methods as the terrain, a framework for development methods provides a grid to be laid upon the terrain. The grid defines reference points that tell us where we are and where we might go to and, importantly, allows us to communicate our current location and destination to others. Frameworks for development methods usually have a number of

	Needs analysis	Behavior specification			
		Synthesis	Simulation	Evaluation	Choice
Social system					
Computer-based system					
Software system					
Software subsystem					

**Figure 3.9:** A framework for development methods.

dimensions that represent the orthogonal conceptual substructures out of which the framework is built. As the number of dimensions *increases*, the ease with which the framework can be understood *decreases* but the accuracy with which methods can be mapped by the framework *increases*. We must therefore search for an optimum between understandability and accuracy of the framework. Most frameworks have two or three dimensions, something which may be related to our limited mental capacity for understanding multidimensional spaces as well as with the ease with which two-dimensional diagrams can be drawn on paper. In this section, we start with a two-dimensional framework, but we indicate how this can be extended to a four-dimensional one — without trying to draw this more complicated framework.

### 3.6.1 The specification of needs, product behavior and product decomposition

If we apply the engineering cycle at every level of the aggregation hierarchy presented in chapter 2, then we get the two-dimensional framework for product development methods shown in figure 3.9. The vertical dimension of the framework indicates the *aggregation level* at which the engineering cycle is applied and the horizontal dimension indicates the *logical tasks* to be performed in the engineering cycle. There are three groups of tasks identified by the framework:

- In **needs analysis**, the function that the system must have for its users is determined. Needs analysis establishes *why* a system should exist. It is the problem analysis task of the engineering cycle. It is oriented to the environment of the product, i.e. to the client or to the market. It produces a specification of the **objectives** that the system must satisfy.
- The **behavior specification** activity corresponds to the rest of the engineering cycle at the same level of aggregation. Behavior specification establishes *what* a system should do. It produces a specification of the observable behavior of a system that would satisfy the stated objectives. If performed according to the engineering cycle, the behavior specification task consists of generating a number of alternative specifications, simulation of their likely effects, evaluation of these effects against the

objectives, and a choice of one specification. Behavior specification is the interface between client/market needs on the one hand and product decomposition on the other. To write a behavior specification, the engineer should be able to switch between the viewpoints of the client or user and that of the designer. This requires social and communicative skills as well as technical knowledge.

- **System decomposition** moves down in the aggregation hierarchy and results in a specification of the internal structure of the system. System decomposition establishes *how* a system shall be structured. This task too can be performed following the engineering cycle, by generating alternative decompositions, simulating their effects, evaluating these against the behavior specification and the system objectives, and choosing one. This application of the engineering cycle is not shown in figure 3.9.

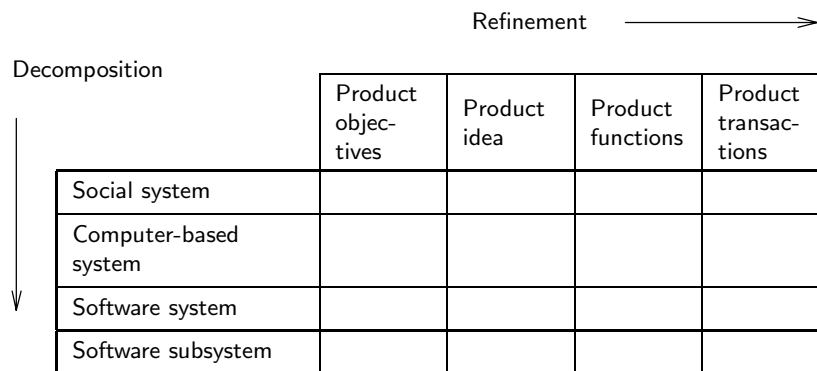
These three tasks correspond to the three views of a system identified in chapter 2: the function, behavior and implementation view, respectively. In the product life cycles shown in figures 3.3 and 3.4, needs analysis is explicitly shown. Behavior specification and the specification of product decomposition are jointly called *product specification* in those life cycles.

A behavior specification is rarely found in one iteration through the engineering cycle, and generally we iterate through the cycle at one level of aggregation several times before we settle upon a specification. This leads to a specification of product behavior at different levels of refinement that are at the same level of aggregation. Three useful levels of refinement are the following.

- The **product idea** is the most abstract concept of what the product should do. It is comparable in abstraction to a mission statement of an organization.
- A **function specification** is a description of the functions that the product should offer its users.
- A **transaction specification** is a list of transactions of the product with its users. As explained in chapter 2, each transaction is an interaction with the environment considered to be atomic.

The product objectives specify a *problem* to be solved; a specification of product objectives is always a specification of user needs. The specifications of the product idea, product functions and product transactions describe a *solution* of the problem at increasing levels of refinement. As illustrated in figure 3.10, solution specifications at different refinement levels describe a product at the same level of aggregation. We call the table in figure 3.10 the **magic square**.

The movement from left to right in the magic square represents an act of **refinement** of an initial product idea by becoming more specific about the observable behavior of the solution. Even then, later discoveries can cause us to perform some more iterations. In the shoe polish packaging example, the developers first delivered a product idea to management, and refined this in several iterations through the engineering cycle until there was a product specification. During refinement, we move from the *why* to the *what*. Refinement reduces the uncertainty about a problem situation because we select a solution to a set of design objectives. The movement from right to left in the magic square decreases the level of refinement and increases the level of **abstraction**. During abstraction, we move from the



**Figure 3.10:** The magic square.

*what* to the *why* at the same level of aggregation, i.e. from behavior to function. Because refinement is a decomposition of functions into more detailed functions, it is also called **function decomposition**. It is important not to confuse this with system decomposition, discussed next.

The movements in the vertical dimension are **system decomposition** moving down and **system integration** moving up. System decomposition moves from the *what* at one aggregation level to the *how* of that level (which is the *what* of the next lower level). Just as refinement resolves uncertainty about the behavior that would satisfy system objectives, system decomposition resolves uncertainty about which decomposition would satisfy system objectives.

It is worth repeating that the system decomposition dimension is orthogonal to the function refinement dimension. There is an important tradition in system design in which these dimensions are mapped to each other. This is the tradition of **functional system decomposition**, in which we decompose a system into subsystems that each implement a system function in such a way that different subsystems implement different functions (at the same level of refinement). In functional system decomposition, the levels of decomposition correspond to levels of refinement. There are however other ways to decompose a system and in general, system decomposition is orthogonal to function refinement.

Comparing the magic square with figure 3.9, we see that in a rational development process, a move from a lower to a higher level of refinement (left to right in the square) takes place according to the engineering cycle. In an actual development process, development may proceed in a less rational way. In general, at any point in the development process, decisions may be made at any level of aggregation and at any level of refinement. For example, a particular development process may contain a sequence of decisions as shown in figure 3.11. Each square indicates a level of aggregation and a level of refinement at which a decision was made. A number in a square represents the fact that a decision has been made at that level of aggregation and refinement, and higher numbers represent later decisions. Occurrences of the number 0 represent the decisions already made before development started. As illustrated by the placement of numbers in figure 3.11, the progression of decisions is not necessarily in the direction of increasing behavior refinement or towards

	System objectives	System idea	System functions	System transactions
Social system	1	2,5	3,5,7	4,5,7
Computer-based system	2,5	6	0	7
Software system		0	3	
Software subsystem		3		0

**Figure 3.11:** A possible sequence of design decisions.

lower levels of system decomposition. The figure contains only one example and other development processes may contain other decision sequences. We return to strategies for ordering the tasks in a development process in chapter 15.

### 3.6.2 Disentangling requirements and other ambiguities

The framework is helpful in disentangling a number of ambiguities in a number of important terms used in product development. First, let us define **requirements engineering** as the process consisting of needs analysis and behavior specification, and **product specification** as the activity of producing a behavior specification and a decomposition specification. As shown in figure 3.12, these activities overlap. To relate these concepts to others introduced earlier in this chapter: product specification occurs in market-oriented as well as in client-oriented development. In market-oriented development, the two additional tasks of product development are writing a marketing plan and a production specification. Requirements engineering and product specification jointly form the development activity. *Implementation* is separate from development, except in software development, where we can decompose a product into components whose specifications are executable. In software development, decomposition *is* implementation. Depending upon the aggregation level, decomposition/implementation of software is called **global design, detailed design** or **coding**.

Figure 3.13 relates the results of development with each other. Results are also called **deliverables**. The deliverables of requirements engineering are called a **requirements specification** and the deliverables of product specification are, due to the process-product ambiguity in the word “specification”, called a **product specification**. A requirements specification includes everything except the product decomposition; a product specification includes everything except product objectives.

An important ambiguity introduced by the above definitions is that specifications of objectives, behavior and decomposition can be produced at any level of the aggregation hierarchy. Thus, what is a specification of an objective at one level of aggregation is “really” a system decomposition when viewed from a higher level of aggregation. As a result, arguments about whether we are specifying system objectives or a system decomposition may be interminable. The answer to both sides in the argument is usually “Yes”. The

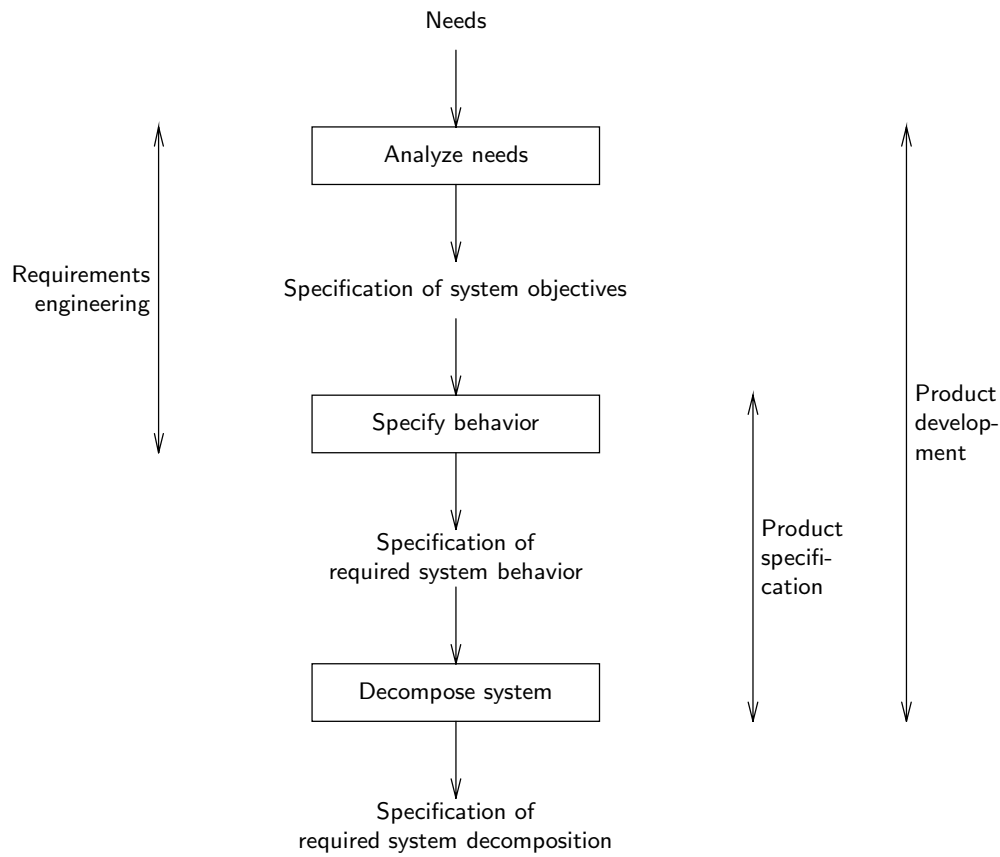


Figure 3.12: Requirements engineering and product specification.

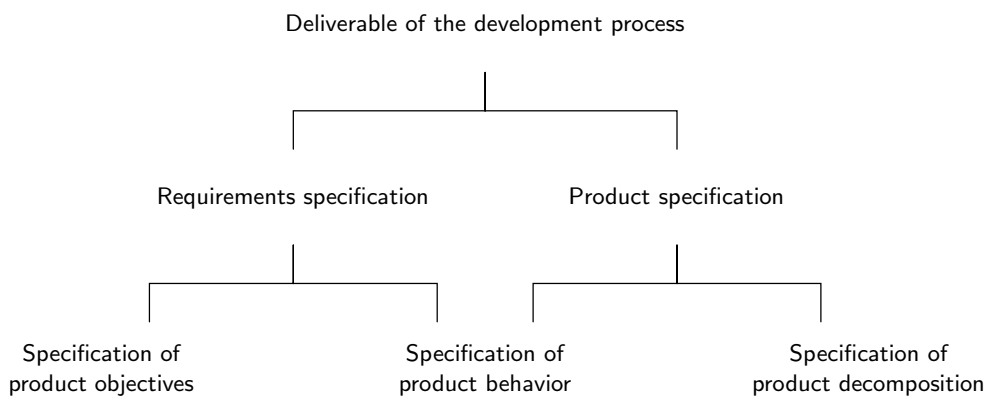


Figure 3.13: Classification of different kinds of deliverables of development.

resolution of the argument simply lies in the identification of the aggregation level at which we observe a system: a *system* decomposition gives us *subsystem* objectives.

Note that this ambiguity is analogous to the ambiguity in the concept of transaction. Arguments about whether a certain interaction is a transaction or not are interminable for the same reason. As we found in subsection 2.4.2, these arguments must be resolved by identifying the aggregation level at which we consider the interactions.

The term **requirement** is perhaps the most overloaded term of all. Are system objectives requirements, or are requirements really behavior specifications? But a system decomposition also gives us requirements. Add to this the ambiguity in the word “specification”, and it becomes clear that the expression “requirements specification” by itself is virtually meaningless. Whenever we use the term, it refers to a deliverable of development consisting of a specification of product objectives and of required product behavior. Where necessary, we will disambiguate the term by indicating the level of aggregation to which we refer.

The term **implementation** is subject to all of the ambiguities identified so far, and some more. Just as “specification”, “implementation” can refer to a process as well as the product of that process; and just as with “specification”, this ambiguity is harmless, because it is always removed by the context of the term. More seriously, for one person, an “implementation” is an organizational subsystem that implements the information supply infrastructure, where for another person it is a piece of code that implements an algorithm. Again, the resolution of many of these ambiguities lies in the identification of the aggregation level.

Figure 3.14 presents a grand picture of the cycle of product evolution. The arrows indicate direction of influence. The reader may want to identify the place of requirements engineering and product specification in product evolution, and check the meaning of the bidirectional arrows in the diagram.

### 3.6.3 Other framework dimensions

So far, our framework has only two dimensions, aggregation level and logic. This is sufficient for an analysis of development *methods*. If we want to analyze the development *process*, we must include *time* as the third dimension of our framework. This will allow us to represent different ways of ordering the tasks of development in time. The distinction between the logical and the temporal dimension of the framework is that the logical dimension represents the process as it would take place in a rational world — which is not the one we live in — and that the temporal dimension represents the process as it can be planned to take place in the actual world. We turn to strategies for *planning* the temporal sequence of tasks of the development process in chapters 15 and 16. We defer the addition of the temporal dimension to those chapters.

A fourth dimension that it is useful to add is that of *aspect*. In section 2.3, an aspect of a system is defined as a subset of the interactions of a system. Each system has potentially infinitely different aspects such as the financial aspect, the technical aspect, the social aspect, the organizational aspect, the economic aspect, the legal aspect, the ergonomic aspect, etc. Most information system development methods focus on the technical aspect of computer-based systems but there are some methods that include organizational or social aspects. For example, ISAC and Information Engineering (part II) include the organizational aspect in their method, and ETHICS (appendix C) includes the social aspect.

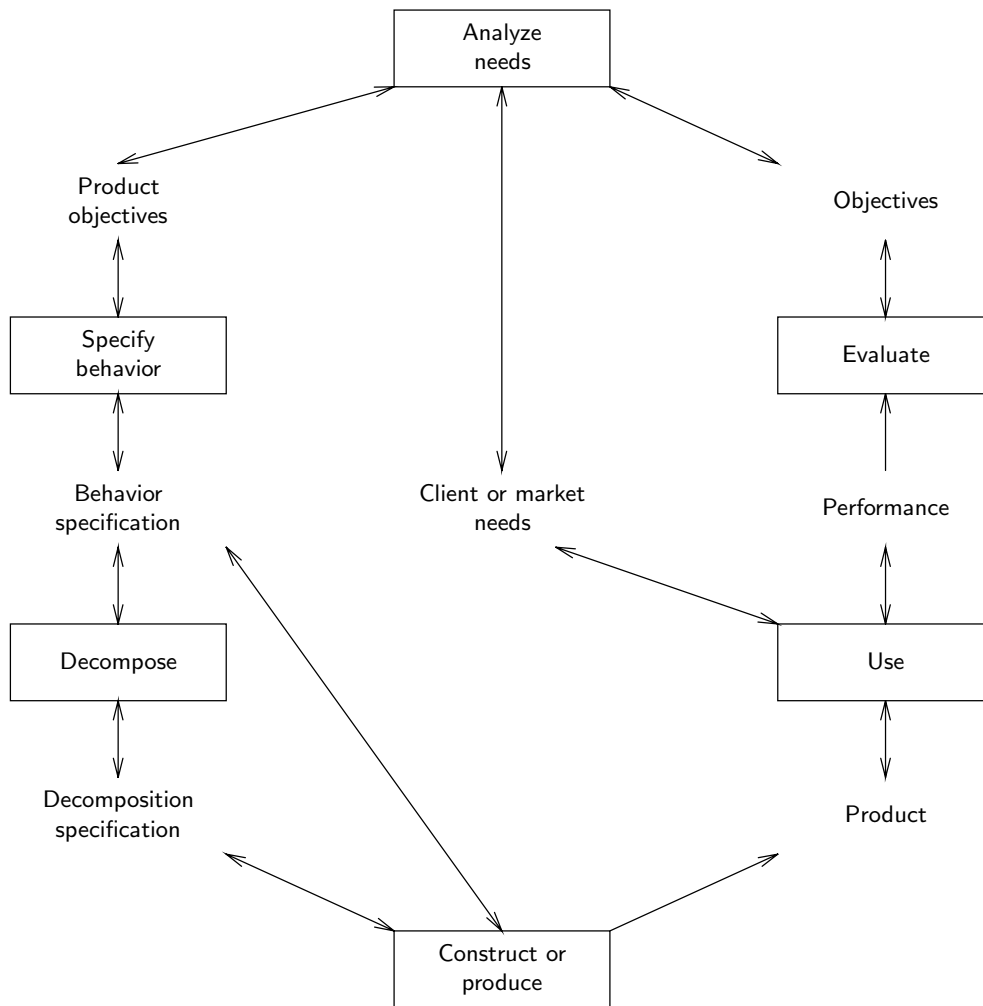


Figure 3.14: Product evolution. The arrows indicate direction of influence.

	ISAC	Information Engineering	Entity-Relationship modeling	Structured analysis/Structured design	Jackson System Development
Social system	<b>Change analysis</b>	<b>Information strategy planning</b>			
Computer-based system	<b>Activity study</b>	Business area analysis			
Software system	Information analysis	Business system design	<b>Entity-Relationship modeling</b>	<b>Structured analysis</b>	<b>UoD modeling, function specification</b>
Software subsystem	Implementation	Technical design and construction		Structured design	Implementation

**Figure 3.15:** Application of the framework to five requirements specification methods.

Because all these aspects are relevant, the engineer will have to cooperate with specialists from different professions. This puts a high premium on the communicative skills of the engineer. Contrast this with the communicative skills required of a scientist who is trained to communicate his or her discoveries to other specialists in the *same* science, all of whom can be assumed to have a similar training and to use the same vocabulary. The engineer will always work in an interdisciplinary context and will have to communicate with people who use another language, geared to the description of other system aspects.

### 3.6.4 Application of the framework

The tasks in the methods described in part II of this book can be mapped to different aggregation levels as shown in figure 3.15. In part II, we only discuss in-depth the tasks printed in boldface. The figure only maps tasks to aggregation levels and does not show the internal structure of the tasks in terms of the engineering cycle. The internal structure will be analyzed after we discussed the methods in some detail.

All methods are *requirements engineering methods*, i.e. they produce a statement of objectives and/or a specification of required system behavior. Entity-relationship modeling, Structured Analysis and Jackson System Development can be applied to the level of computer-based systems as well as of software systems, but they are more at home in the specification of software systems.

One methodological question asked in this book about the methods is

- How can we map the tasks recommended by the methods to the engineering cycle?

Because all studied methods are methods for specifying the behavior of computer-based systems, and these systems have a UoD, there is an additional question to be answered:

- How does the method mix UoD modeling with the specification of required system functions?

It will be shown that Entity-Relationship modeling is primarily a method for finding a conceptual model of the UoD, whereas Structured Analysis does not distinguish a conceptual model of the UoD from a system model. Jackson System Development is unique in making a clear distinction between conceptual model of the UoD and a specification of required system behavior.

The two questions above are about the tasks to be performed in requirements specification process. Two other important questions to be asked concern the structure of the specifications produced when the methods are followed:

- What is the structure of system specifications produced by the methods?
- Can the specifications produced by different methods be integrated?

We will answer these questions in detail for the software system specification methods. As a preparation, we look at the structure of product specifications in general in the next chapter.

## 3.7 Summary

*Product development* is a process that, given a need, delivers a specification of a solution that is expected to satisfy this need. In *client-oriented* development, the need is felt by a single client, in *market-oriented* development, the need is identified to exist in a market. Client-oriented development produces a specification of an individual product, whereas market-oriented development produces a marketing plan, a product specification and a production specification. In both cases, we can identify a number of *stakeholders* in the development process. The *development organization* is the person or organization performing the development process. The *sponsor* pays for the development process, the *customer* buys the delivered product, and the *user* uses the product. In client-oriented development, the sponsor and customer are identical. The client may be the sponsor, customer or user of the product.

All products *evolve*, because when they are used, the needs of the user change and the environment of uses changes as well. In client-oriented development, the needs of the client may even change because of the determination of objectives. This is called *requirements uncertainty*. The characteristic feature of product evolution is that an evaluation of experience of the product *after* it is developed, leads to a (re)development of the product. The logical structure of product evolution is the same as the logical structure of feedback control, viz. the *regulatory cycle*.

In *product engineering*, a product specification is produced by means of a rational problem solving process. The essence of this process is that the effects of alternative specifications are simulated and evaluated *before* the specification is implemented. This evaluation may lead to a new iteration through the engineering process until the desired effects are found.

The *empirical cycle* is a rational problem solving process in which a true model of observed phenomena is sought. The essence of this process is the experimental evaluation of alternative models of the phenomena. The empirical cycle is a useful method for current system modeling as well as for conceptual modeling.

The engineering and empirical cycles are applications of the rational problem solving cycle and are not methods for planning the development or modeling process. They can however be used as targets for a rational reconstruction of the process after the fact, used for justifying the results of the processes.

The empirical and engineering cycles are closely parallel in their form, but they have a few fundamental differences. The empirical cycle is a search for knowledge, but the engineering cycle is a search for the satisfaction of requirements. The empirical cycle is therefore value-free in the sense that it does not pass judgement on the subject of study. The engineering cycle is normative in the sense that the engineer continually passes value judgements on the subject of study, viz. how well the behavior satisfies the objectives of the development process. Related to this difference is the fact that the empirical scientist is an observer with a detached attitude with respect to his or her subject of study, but that the engineer is involved in his or her subject. The engineer is an actor in the world, not an observer of the world. As a consequence, the engineer may be engaged in situations that raise moral questions. There are codes of conduct that provide guidelines for professional and moral behavior.

If we combine the levels of aggregation identified in the previous chapter with the engineering cycle, we get a framework that can be used to classify development methods. In this framework, *requirements engineering* consists of two tasks, *needs analysis* and *behavior specification*. *Product specification* is the partly overlapping task consisting of behavior specification and product decomposition. Market-oriented development consists of needs analysis, product specification, production specification and marketing; client-oriented development consists of needs analysis and product specification.

The development framework is not a strategy to plan the development process. *Time* would be a third dimension of the framework, orthogonal to the partitioning in aggregation levels and to the logical tasks of product development. A fourth dimension would be *aspect*. Important system aspects that may have to be subject of development include the organizational, social, ergonomic, and legal aspects.

### 3.8 Exercises

1. For each of the following development processes, identify the client(s), sponsor(s), customer(s), and user(s).
  - (a) The development of air traffic control software which, by law, must be installed on board of airplanes by the aircraft builder.
  - (b) The development of a new version of a database management system (DBMS) by a vendor of database software.
  - (c) The development of parametrized software for the administration of document circulations.
  - (d) The development of software to register military conscripts.

2. This question is about the TANA case study.
  - (a) What kind of simulations are performed in each of the iterations through the engineering cycle?
  - (b) How are the evaluations of the simulation results performed?
3. Loucopoulos and Karakostas [198] define requirements engineering as the transformation of business concerns into software system requirements. They present a framework in which requirements engineering is decomposed into the following three interacting processes, that are performed in parallel with each other:
  - **Requirements elicitation:** reaching an understanding of the problem. Sources of requirements knowledge are identified and their relevance and significance estimated.
  - **Requirements specification:** describing the problem. In this process, the acquired knowledge is analyzed and organized into a coherent requirements model. A series of conceptual models is produced that represent the problem domain. The conceptual models in the series become increasingly refined, and increasingly represent the software domain instead of the problem domain.
  - **Requirements validation:** ensure that the requirements model agrees with the user's expectations. In this process, experiments are prepared, performed and the results of experiments evaluated.

Compare these tasks with the framework for development of figure 3.9.

4. Nadler [238] observed the problem solving processes of an engineer, a commercial artist, an architect, a physician and a lawyer and found following common structure:
  - (a) Determine the function of the required system.
  - (b) Develop a number of very high level versions of the required system. Evaluate them and select one.
  - (c) Gather information necessary to continue the design.
  - (d) Generate alternative systems on the basis of the abstract ideal and the information gathered.
  - (e) Select the feasible solution.
  - (f) Specify the chosen solution.
  - (g) Subject the selected solution to internal and external review.
  - (h) Test the system design.
  - (i) Install the system.
  - (j) Measure the performance of the system.

Map this process to the regulatory cycle.

5. Verification by testing cannot show the correctness of an implementation, but can show its incorrectness. Compare this with the fact that experiments can falsify a hypothesis but cannot prove it to be true.

6. In the discovery of Neptune, one induction step took place. Identify this step.
7. Dewey [86, pages 107-115] defines what he calls *the process of reflective thinking* as follows:
  - (a) *Suggestions* of alternatives for action. This is the first moment at which we become aware of a problem. Direct, unreflective action is inhibited and the hesitation and delay necessary for reflective thinking arises.
  - (b) *Intellectualization* of the situation, in which the conditions that constitute the trouble and cause the stoppage of action are investigated.
  - (c) A *hypothesis* about what to do “springs up” to the mind and is formulated.
  - (d) *Reasoning* about the possible consequences of the hypothesis.
  - (e) *Testing* the hypothesis by action.

Compare this with the method of rational problem solving discussed in this chapter by correlating each task in Dewey’s method with task(s) in the rational problem solving cycle. In addition, indicate a relationship with the regulatory cycle.

8. Kolb and Frohman [184] define the process of planned change, to be followed by organization consultants, as follows.
  - (a) Scouting: the consultant and the organization explore a potential relationship with the other.
  - (b) Entry: Negotiate a contract.
  - (c) Diagnosis: The consultant analyses the problem of the client, the objectives of the client, the resources of the client and the resource of the consultant.
  - (d) Planning: Generate alternative solutions or change strategies and simulate the consequences of each plan.
  - (e) Action: Implement the best action.
  - (f) Evaluation: Evaluate the results of the change efforts.
  - (g) Termination: Terminate the contract between the consultant and the client organization.

Compare this with the engineering method discussed in this chapter by correlating each task in Kolb and Frohman’s method with task(s) in the engineering cycle. Is there also a similarity with the regulatory cycle?

9. The arrows in figure 3.14 indicate direction of influence. For each arrow, give an example of a case where influence goes in the indicated direction(s).

### 3.9 Bibliographical Remarks

**Rational problem solving** The model of rational problem solving used in this chapter goes back to a proposal by Dewey [86, pages 107-115], used in exercise 7 above. A simplified version of Dewey’s proposal was adopted by Simon [315, 316]. This model has been very

influential in artificial intelligence, where it took the shape of means-ends analysis [318] and was implemented in the General Problem Solver (GPS) [97].

The rational problem solving method has been recognized to be useful by management scientists and consultants too. For example, in a famous descriptive study of 25 strategic decision processes in organizations, Mintzberg, Raisinghani and Théorêt [234] found that the rational problem solving method serves as a useful framework to understand what managers do. Managers do not blindly follow through this process step by step, but choose a path through it that may skip tasks that are easy or for which there is no time, and that may iterate over tasks that are important for the problem at hand.

As a second example, Kolb and Frohman [184] define the process of planned change, to be followed by organization consultants, as a rational problem solving cycle (see exercise 8 above).

Against this prevalence of rational problem solving methods one can observe that often the time to generate all alternatives, or the information to investigate their consequences, are not available. Simon [315] proposed the model of *bounded rationality*, in which only some alternatives are investigated, using the resources available. Secondly, March and Olsen [211] remark that rational problem solving in organizations presupposes that different people have the same preferences and that these preferences can be specified before the choice is made. Both presuppositions are often not satisfied. As an alternative model for organizational decision making they propose the *garbage can model of organizational choice*, in which an organization is viewed as a sea in which problems, solutions and people float around and occasionally meet each other. When the three meet and the organization is expected to behave in a way that is called “making a choice”, some people that happen to be around connect solutions that happen to be available with problems they find themselves confronted with. This model is not suitable to use as a basis for planning product development, but it may help the manager of the development process to understand some of the things going on inside and around the development process.

**The engineering cycle.** The engineering cycle used in this chapter is based upon a model of the decision cycle in engineering design given by Roozenburg and Eekels [289] and by Hall [131]. Additional explanations of the engineering cycle are given by Asimov [15], Jones [169] and Archer [13]. Pugh [276] shows how these ideas can be implemented in a process for integrated product engineering. Archer [11] shows how the product development process is embedded in the larger process of product innovation (viewed in this chapter as a species of product evolution). The example of the development of a new kind of packaging for shoe polish is taken from Roozenburg and Eekels [289].

**Software engineering.** Peters [257] uses some of the insights from industrial design to define a framework for software engineering methods. Jensen and Tonies [165] analyze the software engineering process from a general engineering standpoint as a problem solving process and come up with a model of the engineering cycle similar to the one presented in this chapter.

The view that software engineering should be an *engineering* discipline is supported by D.M. Berry in a report for the Software Engineering Institute [30]. However, Berry ignores or at least underplays the role of simulation in engineering before implementation. He quotes Koen [183] in defining engineering as the strategy for causing the best change in a

poorly understood situation with the available resources, and he then quotes an unpublished definition of Mary Shaw and Watts Humphrey that software engineering is “that form of engineering that applies computer science and mathematics to achieving cost-effective solutions to software problems”. In this definition, software engineering is problem solving, but the use of simulation and evaluation before the software product is implemented, is not mentioned. Baber [17] takes the same view of engineering as we do and emphasizes the use of scientific knowledge and principles in the computation of product properties before the product is actually built. Consequently, he doubts whether software engineering currently is really an engineering discipline.

Although software engineering, if it is to fulfill the promise of its name, can learn from product engineering, there are nevertheless some fundamental differences between software engineering and product engineering. In a famous position paper published in 1985, Parnas [254] argues that due to discrete state-changing behavior and complexity, software is inherently unreliable. Software engineering, formal methods and artificial intelligence techniques may provide some help to master the complexity of software, but will do so only in small increments. In an equally famous paper that appeared two years later, Brooks [53] lists four essential differences between industrial product development and software development, that make the engineering of software essentially hard: the complexity, invisibility, and changeability of software when compared to other products, and the intertwining of software with other products. Because of this intertwining, software must conform to a large number of requirements imposed upon it by these other products. In a reaction to these papers, Harel [138] provides a more optimistic view by arguing that system modeling, visual languages and model execution may provide the means to tackle some of these problems. We may observe that modeling a system before it is built and executing the model to evaluate it, both proposed by Harel, are part of the essence of the engineering method as defined in this chapter.

**Empirical cycle.** The empirical cycle is described in any introductory textbook on the philosophy of science. Classic statements are given by Kemeny [173] and Nagel [239]. The observation that hypotheses can only be falsified, not verified, comes from Popper [264]. Popper’s falsification theory corresponds with Dijkstra’s [87] famous observation that tests can only show the presence of bugs in a program, not their absence. The comparison of the engineering cycle and empirical cycle is based on an analysis of the logic of social action given by van Strien [329] and on a comparison of the two cycles by Roozenburg and Eekels [289].

**Codes of conduct.** Codes of conduct relevant for engineering computer-based systems are the *ACM Code of Professional Conduct* [2] and the *IEEE Code of Ethics* [154]. Anderson *et al.* [9] illustrate the use of the ACM code in decision making. There are now several books on ethical aspects of decision making in computer-based system development, including Johnson [167], Ermann, Williams and Guttierrez [96] and Forester and Morrison [106].

**Needs analysis.** Needs analysis is the most difficult task in the entire development process. Mistakes made here have the most expensive consequences. The problem to be solved at this stage is not to find a solution to the requirements; it is to find out what the objectives of the product are [12], and this is a **wicked problem**. According to Rittel and

Webber [279], wicked problems have no definitive formulation, and the solution is determined to a large extent by the formulation of the problem. Wicked problems do not have a solution space that can be enumerated or even described, and once we start analyzing them, we always find they are symptoms of other problems. There is no criterion to determine the quality of solutions and the problem solving process has no stopping rule. Whatever we accept as a solution is not true or false, it is good or bad. Wicked problems are unique, and they are so urgent that the problem solver cannot afford to be wrong. There is no occasion to try a solution first and implement it later; every trial solution counts as an irreversible step in real life. In other words, a wicked problem is a *real* problem.

Given this predicament, one may wonder what to do. In a delightful little book, Gause and Weinberg [112] give a number of important hints on how to go about solving real problems. In a more extensive treatment [111], they give numerous practical methods and techniques by which to tackle problems in needs analysis. Gause and Weinberg argue that we can never know what the client's *needs* are and that requirements determination is all about determining the client's *desires*. As defined in chapter 2, user needs are user desires, so in the terminology of this book, needs analysis is the determination of the client's desires. A survey of experimental techniques for needs analysis is given by Gutierrez [129]. Byrd, Cossick and Zmud [55] give a synthesis of techniques for needs analysis and knowledge acquisition. Jones [168] gives a very interesting survey of techniques for different stages in product development, including needs analysis.

**Requirements uncertainty.** The principle of requirements uncertainty has been called the *uncertainty principle of data processing* by James Martin and Clive Finkelstein [217] and by McCracken and Jackson [221] and *Heisenberg-like uncertainty* by Lehman [191]. The essence of this uncertainty is that by installing and using a product, needs tend to change so that the product tends to become obsolete by being used.

**Behavior specification.** An early survey of behavior specification techniques (also called *requirements* specification techniques) was given by Taggart and Tharp [336]. In the same year, Ross and Schoman [294] published a classic paper on the specification of required system behavior, that is full of insights in the requirements engineering process. The distinction between function, observable behavior and structure (why, what and how), which is one of the structuring themes of this book, is already made in this paper.

Davis [77] gives a survey of techniques which uses roughly the same classification of techniques as we do here, viz. problem analysis (called objectives determination here) and requirements specification (called behavior specification here). However, Entity-Relationship modeling, Data Flow modeling and Jackson System Development are treated as problem analysis methods by Davis and as behavior specification methods in this book. This is explained by the fact that we focus on computer-based systems, whereas Davis focuses on software systems, which is one level lower in the aggregation hierarchy. What is behavior specification for a computer-based system is objectives determination, and hence problem analysis, for a software system.

**Requirements engineering.** There are a number of good surveys of issues in requirements engineering. Curtis, Krasner and Iscoe [74] report on an empirical study of software product development processes that the major problems in these processes are the thin

spread of application knowledge, changing and conflicting requirements, and communication and coordination breakdowns. The first two of these problems concern the requirements engineering task. Dorfman [89] gives a brief introduction to requirements engineering at the level of computer-based systems and of software systems and shows how requirements flow down from one level to the next. Stokes [326] gives a survey of the problems with requirements engineering, possible solutions to these problems, and methods and techniques for the specification of required system behavior. Brackett [50] gives an interesting survey of the subfields of requirements engineering in the form of a curriculum outline and an extensive annotated bibliography. Yeh and Ng [373] give a brief survey of methods and techniques for specifying requirements. They emphasize the need for specifying the environment of the product in addition to specifying the properties required of the product. In an interesting study by Lubars, Potts and Richter [199] of requirements engineering in ten organizations, it was found that most problems of requirements engineering in practice are organizational, and that organizational solutions to technical problems were sought. Examples of problems that exist in practice are poor interactions between developers and users, lack of guidance in finding a product specification, problems in specifying product objectives, and changing requirements. Hsia, Davis and Kung [149] suggest a number of research directions to tackle these issues, such as animation of specifications, computer-aided requirements engineering and method integration.

It is clear that interest in requirements engineering is rapidly rising. In 1991, a special issue of the *IEEE Transactions on Software Engineering* was devoted to requirements engineering [79] and in 1993 a two-yearly conference on requirements engineering [102] was started. The March 1994 issue of *IEEE Software* includes some papers from the first conference in this series [80]. Starting from 1996, Springer will publish the *Requirements Engineering Journal*. In addition to Davis' book [77] on software requirements specifications, there is now a textbook by Loucopoulos and Karakostas on requirements engineering [198]. This book gives a useful survey of techniques and heuristics for elicitation, modeling, and validation of system requirements.

**Frameworks.** There are numerous framework for system development. The framework defined in figure 3.9 has two dimensions, *logic* and *aggregation level*. In an influential paper, Hall [132] defines a framework with three dimensions: *logic* (problem solving procedure), *time* (problem solving process) and *aspect* (kind of knowledge of the developed system). The logic dimension is virtually the same as the logic dimension of our framework. Hall's time dimension is introduced in chapter 15, when we look at strategies to perform the actual system development process. A two-dimensional logic-time framework, corresponding to two of Hall's three dimensions, is now well-accepted in software engineering [222, 287, 240], but to my knowledge, no reference is made in the software engineering literature to Hall's framework. The term "magic square" comes from Harel and Pnueli [140], but they do not mention the particular levels of refinement and decomposition that I use. They give a clear argument for the importance of the magic square in understanding system development.

Roman, Stucki, Ball and Gillett [286] define a framework for system development that uses an aggregation hierarchy too. At each level of the hierarchy, a sequence of tasks is defined that can be construed as a division of requirements determination and conceptual modeling into subtasks, and a merging of the resulting steps.

Davis [76] gives a framework that resembles the one used in this book. Davis' framework consists of an aggregation dimension and a logical dimension. However, he focuses on

one aggregation level only, that of software product development, and he does not use the engineering cycle as a logical task structure at each level of aggregation. Rather, he distinguishes user needs analysis, solution space definition, external behavior definition and preliminary design as logical tasks at one level. In terms of our framework, these would be the result of successive iterations through the engineering cycle at one level of aggregation.

The frameworks discussed so far are *method-oriented*. They try to classify tasks in the development process. Some frameworks are *specification-oriented*, i.e. they are based on a classification of the intermediary or final specifications produced. An example is Blum's [34] framework, which distinguishes problem oriented from product oriented specifications along one dimension, and semi-formal and formal representations on the other. Pohl [259] distinguishes three specification dimensions: (in)completeness of the specification, (in)formality of the specification and (dis)agreement about the specification.

When we turn to frameworks for information system development methods, we find a wide variety, most of them oriented towards what aspect of the UoD is represented. Many of these are published in the proceedings of the conferences on Comparative Research in Information Systems (CRIS) [248, 247, 249]. These frameworks can best be understood as frameworks for UoD models, and should be compared with the modeling framework discussed in chapter 13.



## Chapter 4

# Requirements Specifications

### 4.1 Introduction

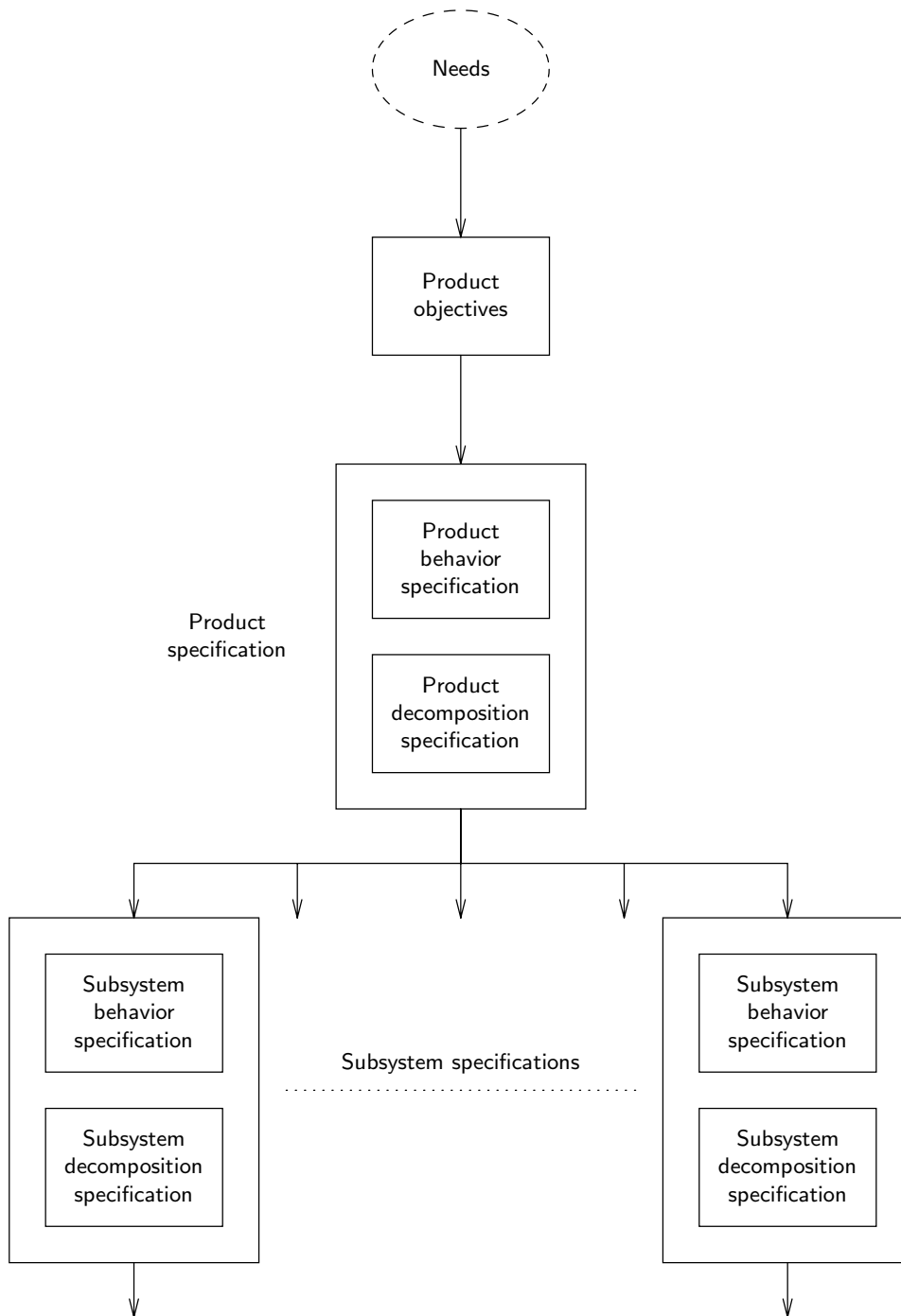
A requirements specification consists of a specification of product objectives and a specification of required product behavior. In this chapter, we discuss the structure of a specification of product objectives (section 4.2), classify different kinds of behavior specifications (section 4.3) and give a simple framework for the structure of behavior specifications (section 4.4). In section 4.5, we list a number of desirable properties of requirements specifications.

### 4.2 Product Objectives

The generic objective of any product is to answer needs that exist in its environment. Any development process starts with a statement of product objectives and produces behavior specifications and product decompositions along the way. As explained in chapter 3, what is a decomposition from one point of view is part of the objectives at the next lower level of aggregation. It is therefore useful to distinguish the top level objectives identified as the result of a needs analysis from lower level objectives. Figure 4.1 clarifies the situation. At each level below that of the top level objectives, the behavior specification and decomposition specification are the objectives for systems at the next lower level of aggregation.

An example top level objective in the TANA case study (chapter 3) was to maintain TANA's market share. The output of the first iteration through the engineering cycle in that example was the decision to develop a new product for the current market of TANA. This output then became the objective to be realized in the ensuing development process (figure 4.1). The product specification was then written in a number of iterations that stayed at the same level of aggregation. For example, in the second iteration, a product idea was delivered as initial product specification. This became the objective of the third iteration through the engineering cycle, in which a concept of operation was produced, etc.

Example top level objectives identified in the library case study of ISAC change analysis in chapter 5 are the improvement of acquisition procedures and the improvement of return procedures of the library. In all cases, top level objectives are the translation of client or market needs into objectives to be achieved by the product under development.



**Figure 4.1:** The role of objectives in the development process. Each product specification is a statement of objectives for its subsystems.

It is common to distinguish objectives from **constraints** that the product must satisfy. The difference between objectives and constraints is however more in the eye of the beholder than in the objectives or constraints themselves. Both are norms that the product must satisfy. The difference seems to be that limitations on the solution space, imposed from the outside, are experienced by some developers as constraints on their design freedom. By contrast, goals worth striving for, peaks of achievement to be realized by the choices that the developer makes, are viewed by those developers as objectives. This distinction is wholly subjective, and in what follows the term “constraint” is used as a stylistic variant of “objective”.

### 4.3 Behavior and Property Specifications

We can specify product behavior at different levels of refinement. We saw in subsection 3.6 that the following three levels are useful in practice:

- The **product idea** is the most abstract specification of what the product should do.
- A **function specification** is a description of the functions that the product should offer its users.
- A **transaction specification** is a list of transactions of the product with its users.

In subsection 2.4.4, we saw that aspects of product behavior can be summarized by product *properties*. At the level of abstraction of the product idea or that of required system functions, it is often not possible to specify properties behaviorally. Once we are able to specify required product transactions, we should also be able to specify most required product properties in a behavioral way. For properties for which we cannot find a behavioral specification, we should try to find *proxies*, whose presence can be behaviorally specified and that suggest the presence of the desired property. Indeed, decreasing the abstraction level from product idea to product transaction, we often replace a high level desired product property by lower level proxies for that property.

A behavioral specification of a property describes an experiment in which evidence is given for the presence or absence of the property. This experiment can be described at various levels of detail. For example, we can represent user-friendliness by the proxies that the average initial training time needed by all users must be short and that the number of help requests about the system must be low. These observation procedures must be made more precise by indicating how users are sampled, or over which period they are observed, what is the size of the user population over which we average help requests, etc. To keep the main flow of ideas in a specification clear, we should write a global specification of the property first and add measuring procedures later as footnotes [111, page 172].

There are a number of checklists of kinds of properties that a software system can have. Figure 4.2 gives one such list. *Correctness* is a meta-property, for it indicates the degree to which the product satisfies its (other) properties. *Security* is an example of a **negative** property, because it is about something that should *not* be observed. An example of a security property is the property that unauthorized access cannot occur. Other examples of negative properties are absence of deadlock, absence of collisions in a robot control system and, more generally, absence of unsafe behavior. Examples of **positive** properties are that a system should cost less than Dfl 10 to the customer and that it should have a response time

- **Correctness:** The extent to which the product satisfies the objectives.
- **Reliability:** The extent to which the product behaves as specified in different circumstances.
- **Efficiency:** The amount of resources needed by the product.
- **Security:** The extent to which unauthorized use of the product can be prevented.
- **User-friendliness:** The ease of use of the product.
- **Maintainability:** The ease with which product malfunction can be repaired.
- **Testability:** The ease with which the product can be ascertained to conform to its specification.
- **Flexibility:** The ease with which the product can be modified after delivery, so that it conforms to a changed specification.
- **Portability:** The ease with which the product can be ported from one hardware and software environment to another.
- **Reusability:** The ease with which components of the product can be used in other products.
- **Interoperability:** The ease with which the product can interface to other products.

**Figure 4.2:** Kinds of software product properties.

of less than 2 seconds. It is much harder to specify an observation procedure for negative properties than for positive properties. Experimental verification that a product satisfies a negative property would require observation of the product throughout its complete lifetime. Obviously, such an experiment is too expensive and is useless — we would like to verify the property before the product is used, not after it is disposed of. A convincing verification of the presence of a negative property in a product must take the form of a mathematical or logical proof that the unwanted behavior *cannot* occur.

Different required properties may conflict with each other. In case of conflict, the engineer must perform trade-off studies, and to perform these studies, it is desirable that required properties be specified in the form of **preferences** for certain properties above others. A minimal indication of preferences is a distinction between **essential** properties that the product must have, and **desired** properties that it would be nice to have. A more refined specification would give an ordering of all possible property values in order of preference. For example, preferences of the market for washing machines may be such that a noise level of 0 dB has the highest preference and 9 dB the lowest. A noise level of 0 dB may be achievable at a very high price by using expensive state-of-the-art anti-noise technology, and the market also happens to have a preference for cheap washing machines. As a result, the marketing department adjusts its preferences and settles on an acceptable combination of noise level and price. Of course, preferences depend upon the objectives to be realized by the product. Should the washing machine be targeted for the expensive but small high end of the market or for the mass consumer market? This determines the preferences used in the trade-off between high technology and low price.

	Static dimension		Dynamic dimension	
	States	Static constraints	Transitions	Dynamic constraints
UoD	ER	ER	JSD	JSD
SuD	ER	ER	SA JSD	JSD

Figure 4.3: A framework for behavior specifications.

## 4.4 A Framework for Behavior Specifications

The methods reviewed in part II all lead to the specification of required product behavior. ISAC change analysis and IE information strategy planning additionally lead to the specification of product objectives. In part III, we will show how to integrate behavior specifications produced by the different methods. We will use a simple framework for behavior specifications, based upon the following two dimensions.

- **Static dimension of behavior:** each behavior specification specifies the set of all possible observable system **states**. There is a small set of techniques and notations for the specification of a system state space, converging on the specification of entities, relationships, and properties. Constraints on the state space are called **static constraints**. Important classes of constraints on the state space are existence constraints and cardinality constraints.
- **Dynamic dimension of behavior:** each behavior specification specifies the set of all possible observable **state transitions** in the state space. Observable state transitions are the transactions of the system. There is a large set of techniques and notations used to specify these transitions. Examples of techniques are decision trees, decision tables, and state transition diagrams. In addition to specifying individual state transitions, most techniques allow the specification of constraints on those transitions, such as constraints on sequencing, delay and response time. Constraints on state transitions are called **dynamic constraints**.

Because computer-based systems manipulate data, there are two systems we can specify: the computer-based system and its UoD. In a development process, the computer-based system is also called the **system under development** or SuD. The specifications of the SuD and UoD are **conceptual models**, because they embody our shared understanding of these systems and are used to facilitate communication between people about the UoD or SuD. The conceptual model of the SuD is prescriptive, the conceptual model of the UoD is descriptive.

Figure 4.3 indicates the coverage of the behavior specification methods reviewed in part II.

- The ER method (chapters 7 and 8) can be used to write a specification of the state space of a data manipulation system and of some of the static constraints on the

states in this space. It is neutral with respect to modeling the SuD and modeling the UoD of the system. That is, by looking at the specification itself, we cannot discover whether it is a system model or a UoD model.

- The Structured Analysis method (chapters 9 and 10) can be used to write a specification of the transactions of the SuD. It produces a model of a data manipulation system and not of the UoD of such a system. It focuses on the specification of transactions.
- The JSD method (chapters 11 and 12) can be used to write a specification of the transactions of the SuD and of sequencing constraints on those transactions. It distinguishes a model of the UoD of the system from a model of the system functions itself.

A glance at figure 4.3 shows that jointly, the three methods have a good coverage of the different aspects of system behavior. In chapters 13 and 14, we use the framework of figure 4.3 to analyze the possibilities to combine and integrate these behavior specification methods.

## 4.5 Desirable Properties of a Requirements Specification

Product specifications should themselves have a number of properties, that follow from the purpose for which they are written. The purpose of a specification is to specify all and only the behavior required of a product, so that customers and sponsors know what they get and constructors know what to build (or buy). The requirements specification should be fulfill this purpose during construction or production as well as during product evolution. In order to fulfill this purpose, product specifications should be communicable, true, complete, feasible, verifiable, and maintainable. Figure 4.4 defines these properties and gives a number of aspects of each property.

The most important property of a specification is that it should be communicable. If it is to have any use for the sponsor, developer, client or marketing department, it must be able to use it as a channel of communication between them. Communicability is a pre-supposition of the other properties of a specification. For example, truth and completeness cannot be verified if the specification is not communicable. Communicability means, first, that the specification must be *understandable* by all stakeholders, and second, that the stakeholders must have the *same* understanding of the specification. In other words, the specification must be unambiguous. Understandability and unambiguity are sometimes at odds with each other. For example, mathematical expressions are unambiguous but are not understandable for many people. Nevertheless, without understandability and unambiguity, validation of a specification with stakeholders is problematic. Gause and Weinberg [111] single out unambiguity as the most important desirable attribute of specifications.

The requirement that a product specification be **true** does not mean that it describes the product accurately, but that it specifies the requirements on the product accurately. This is also called **validity** of the specification. This means, among others, that the specification should not specify the decomposition of the system, nor make any other implementation decisions. Of course, implementation constraints may be part of the product objectives, but this should not be a decision on the part of the writer of the specification.

- **Communicability:** The specification should serve as a channel of communication about the product among all stakeholders.
  - Understandability
  - Unambiguity
- **Truth:** The specification should describe requirements and nothing else.
  - Validity
  - Implementation-independence
- **Completeness:** The specification should describe all requirements and not less.
  - Validated
  - Preferences included
- **Feasibility:** The specification should describe behavior that can be realized in a product.
  - Consistency
  - Cost-effectiveness
- **Verifiability:** It should be possible to observe whether a product satisfies the specification.
  - Observation procedures
- **Maintainability:** The specification must be maintainable when requirements change after delivery of the product.
  - Traceability
  - Modifyability

**Figure 4.4:** Properties that a product specification should have.

**Completeness** of a specification entails that no requirement has been omitted from the specification. In practice, this means that the requirements have been validated with the sponsor and that the sponsor agrees that there are no other requirements to be satisfied. Completeness entails that where possible, constraints have been annotated with preferences.

**Feasibility** of a specification means, minimally, that the specification is *consistent*. By this we mean that a product that satisfies the specification can exist. A specification that requires a response time to be less than 3 seconds and greater than 5 seconds cannot be satisfied and is therefore inconsistent. Beyond this minimal requirement, a specification should be *cost-effective*. This means that the cost of implementing the product is justified by the benefit that accrues from implementing it.

A specification is **verifiable** if there is a cost-effective procedure for ascertaining whether a product satisfies the specification. This means that all properties must be specified in a measurable way. The observation procedure is the experiment by which the property can be observed, and it can serve as an acceptance test when the product is handed over to the customer.

A specification is **maintainable** if changes in client or market needs that arise after delivery of the product, can be easily incorporated in the specification. If product evolution is done in a controlled way, changed needs lead to a (re)development process that starts with an update of the product specification. Product evolution by updating the product specification requires at least modifyability of the specification itself, but in addition, it requires traceability of the specification in the forward and backwards directions.

## 4.6 Summary

A requirements specification at any level of the aggregation hierarchy consists of a specification of product objectives and a specification of observable product behavior. The top level objectives of the product link a product specification to the needs of the client or market. At any aggregation level, the behavior specification and decomposition specification of a product jointly form the objectives of the lower level product components.

A behavior specification can be as abstract as a product idea, or it can be a more concrete specification of required product functions or even atomic transactions. At each abstraction level, properties capture an aspect of product behavior. All properties should be specified in a measurable way, but there may be properties for which only nonbehavioral specifications can be found. For these properties, proxies should be given that can be behaviorally specified. Properties can be classified under the headings of correctness, reliability, efficiency, security, user-friendliness, maintainability, testability, flexibility, portability, reusability and interoperability. For *positive* properties, a finite experiment should be specified in which the property can be observed, but for *negative* properties, which require a certain behavior to be absent, specifying such a procedure is difficult, if not impossible. Presence of a negative property in a product can only be verified by mathematical or logical proof.

Where possible, preferences should be indicated for different values of a property. A minimal preference specification is a classification of properties into *essential* and *desirable* properties.

A simple framework for behavior specifications distinguishes a *static dimension*, in which states and static constraints are specified, from a *dynamic dimension*, in which transitions and dynamic constraints are specified.

A product specification must itself satisfy certain requirements. In particular it must be true, complete, feasible, verifiable and maintainable. Secondary features, which derive from these primary ones, include implementation-independence, consistency, cost-effectiveness, modifyability and forward and backward traceability. A very important requirement is the communicability of the specification, for without this, it could not satisfy any of its other properties. Two important aspects of communicability are understandability and unambiguity.

## 4.7 Exercises

1. A certain university faculty sells a lecture notes series to its students. For most courses, there is a volume of lecture notes that students can buy. In order to give these notes a more visible presence on the bookshelves of the students, the faculty decides to design a new style for the cover of the volumes in the series. You are to write a product specification for the new cover.
  - (a) Identify the stakeholders of this project.
  - (b) Write down the list of functions that the product should have.
  - (c) Write down a list of properties that the functions should have. Use the list of figure 4.2 as a checklist to see if you have forgotten any properties.
  - (d) List three alternative decomposition specifications that would satisfy the properties and evaluate them with respect to the properties.
2. Consider the TANA case study of chapter 3.
  - (a) The first iteration through the engineering cycle produced a deliverable containing, as components, a description of the objectives of that iteration, simulations of the alternatives considered and the evaluation of their effects. Describe the contents of the objectives and alternatives. How do you think the evaluations were recorded?
  - (b) In the second iteration, a deliverable was produced that contains, as components, a description of the generated alternatives and their simulations and evaluations. Describe the contents of these deliverable components.
3. Look up the IEEE/ANSI standard for requirements specifications [156, 90] and classify the sections in this standard under one of the following headings:
  - Top level objectives.
  - Product behavior. Classify this as a specification of the product idea, a function specification, or a transaction specification.
  - Properties, classified according to the list in figure 4.2.

## 4.8 Bibliographical Remarks

**Product objectives and properties.** Gladden [116] maintains that the specification of product objectives is a critical success factor in product development. Combined with vivid

simulation of expected product properties, it will, according to Gladden, lead to a successful product. Although this is probably an overstatement, it is clear that the specification of product objectives plays an important unifying and regulatory role in product development.

Two very useful discussions of the specification of product objectives and properties are given by Gause and Weinberg [111] and Gilb [114]. The observation that the difference between objectives and constraints is more in the eye of the beholder than in the requirements themselves was made by Simon [317]. The list of kinds of software product properties given in figure 4.2 is based on a similar list given by Vincent, Waters and Sinclair [350, page 12], who base themselves upon a report by McCall, Richards and Walters [220] and on a discussion by Davis [77]. Yeh et al. [374] also give a list of the kinds of “nonfunctional” properties that a software product can have. Keller, Kahn and Panara [172] describe procedures to make software product properties observable by defining metrics.

**Requirements specification.** Davis [77] gives a thorough survey of requirements specification techniques and discusses several standards for requirements specifications. Dorfman and Thayer [90] contains a large number of requirements specification standards, and Thayer and Dorfman [343] contains a number of useful papers on requirements specification. Surveys of the desirable characteristics of requirements specifications are given by Boehm [38], Davis [77] and Lindland *et al.* [194]. Davis convincingly shows that there are trade-offs among the desirable properties, such that increased satisfaction of one property implies decreased satisfaction of another.

Backward traceability of a specification to reasons is discussed by Potts and Bruns [265]. However, Potts and Bruns use a more elaborate argumentation theory than the simple listing of alternatives, simulations and evaluations proposed here. Backwards traceability to sources is proposed by Gotel and Finkelstein [124].