1

Introduction

The development of complex software systems on a large scale is usually a complicated activity and process. It may involve many developers, possibly with different backgrounds, who need to work together as a team or teams in order to ensure the productivity and quality of systems within a required schedule and budget. Each developer plays a specific role, for example, as an analyst, designer, programmer, or tester, and is usually required to produce necessary documents. The documents may need to be provided to other developers in the team for reading or for assisting them in performing their tasks. For this reason the documents need to be well presented, with appropriate languages or notations, so that they can be understood accurately and used effectively.

In the early days of computing, software was seen as synonym of program, but this view was gradually changed after the birth of the field *software engineering* in the late 1960s [1, 2]. Software is no longer regarded only as a program, but as a combination of documentation and program. In other words, documentation is part of software that represents different aspects of the software system. For example, the documentation may contain the user's requirements, the goal to be achieved by a program, the design of the program, or the manual for using the program.

The documentation is important for ensuring the quality and for facilitating maintenance of a program system. If the documentation containing the user's requirements or the program design is difficult to understand accurately by the developers undertaking subsequent development tasks, the risk of producing an unsatisfactory program system will run high. The consequence of this can be serious: the program system either needs more time and effort to be improved to the level that is deliverable or needs to be completely rebuilt. In either case, a loss of money and time is unavoidable.
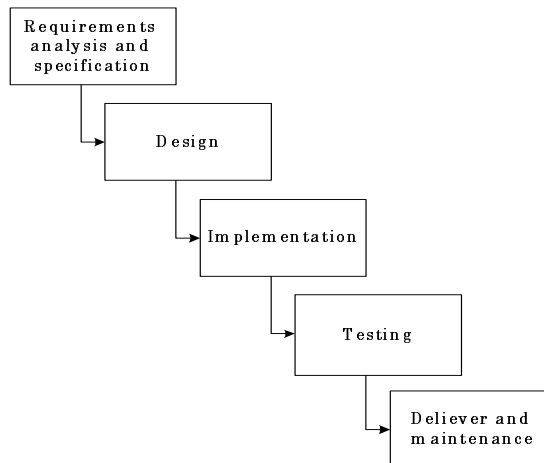
Fig. 1.1. The waterfall model for software development

1.1 Software Life Cycle

Software, like a human being, has a *life cycle*, composed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a specification or a test plan [32]. A typical software life cycle, known as *waterfall model*, is given in Figure 1.1. Although the real picture of the software life cycle may be much more complicated than the waterfall model, it depicts primary features of the software development process. Almost every other model uses the idea of the waterfall model as its foundation [11, 84, 8, 111, 98].

The typical waterfall life cycle model comprises five phases: *requirements analysis and specification*, *design*, *implementation*, *testing*, and *delivery and maintenance*.

Requirements analysis and specification is a study aiming to discover and document the exact requirements for the software system to be constructed [23][51][52]. To this end, the system in the real world, which is to be computerized, may need modeling so that all the necessary requirements can be explored. The result of such a study is usually a document that defines the identified requirements. A requirement in the document can be a statement, a formal logical expression, a text, a diagram, or their combinations that tell *what is to be done* by the system. Such a document is usually called a *requirements specification*. For example, "build a student information system" can be an abstract level requirement.

Design is an activity to construct a system, at a high level, to meet the system requirements. In other words, design is concerned with *how* to provide a solution for the problem reflected in the requirements [56]. For this reason, design is usually carried out on the basis of the requirements specification.

Design can be done in two stages: *abstract design* and *detailed design*. Abstract design is intended to build the architecture of the entire system that defines the relation between software modules or components. Detailed design usually focuses on the definition of data structures and the construction of algorithms [15, 99]. The result of design is a document that represents the abstract design and detailed design. Such a document is called *design* or *design specification*. To distinguish between the activity of design and the document resulting from the design activity, we use *design* to mean the design activity and *design specification* to mean the design document in this book.

Implementation is where the design specification is transformed into a program written in a specific programming language, such as Pascal [37], C [58], or Java [4]. The implemented program is executable on a computer where the compiler or interpreter of the programming language is available. The primary concerns in implementation are the functional correctness of the program against its design and requirements specifications.

Testing is a way to detect potential faults in the program by running the program with test cases. As there are many ways to introduce faults during the software development process, detecting and removing faults are necessary. Testing usually includes the three steps: (1) test case generation; (2) the execution of the program with the test cases; and (3) test results analysis [115, 53].

There are two approaches to program testing: *functional testing* and *structural testing*, which are distinguished by their purposes and the way test cases are generated. Functional testing, also known as *black-box testing*, aims to discover faults leading to the violation of the consistency between the specification and the program, and test cases are generated based on the functional specification (requirements specification or design specification or both) [45, 9, 108]. Structural testing, alternatively known as *white-box testing*, tries to examine every possible aspect of the program structure to discover the faults introduced during the implementation, and test cases are therefore generated based on the program structure [106]. In general, both functional testing and structural testing are necessary for testing a program system because they are complementary in finding faults.

Deliver and maintenance is where the system is ultimately delivered to the customer for operation, and is modified either to fix the existing faults when they occur during operation or to meet new requirements [111]. Maintenance of a system usually requires a thorough understanding of the system by maintenance engineers. To enhance the reliability and efficiency of maintenance, well documented requirements specification and design specification are important and helpful.

In addition to the forward flow from upper level phases to lower level phases in the software life cycle, we should also pay attention to the backward flow from lower level phases to upper level phases. Such a backward flow represents a feedback of information or verification. For example, it is desirable to check whether the design specification is consistent with the requirements

specification, whether the implementation satisfies the design specification, and so on.

1.2 The Problem

One of the primary problems in software projects is that the requirements documented in specifications may not be accurately and easily understood by the developers carrying out different tasks. The analyst may not understand correctly and completely the user requirements due to poor communication; the designer may misunderstand some functional requirements in the specification due to their ambiguous definitions; the programmer may make a guess of the meaning of some graphical symbols in the design specification; and so on. The major reason for this problem is the use of informal or semi-formal language or notation, such as natural language (e.g., English) and diagrams that lack a precise semantics. Let us consider the requirements for a *Hotel Reservation System* as an example:

A Hotel Reservation System manages information about rooms, reservations, customers, and customer billing. The system provides the services for making reservations, checking in, and checking out. A customer may make reservations, change, or cancel reservations.

This specification defines necessary resources to be managed and desirable operations to be provided for the management of the resources. The resources include rooms, reservations, customers, and customer billing. The operations are making reservation, checking in, checking out, changing reservations, and canceling reservations. As all the terms representing either resources or operations are given in English, they may be interpreted differently by different developers. For instance, by customers the analyst might mean persons with a full name, address, telephone, and room reservations, but the programmer may misunderstand it as persons with only a full name; by checking in the analyst might mean that the customer has arrived at the hotel, obtained the room key, and made payment for all his or her room charges in advance, but the programmer may misunderstand that checking in does not require advanced payment.

This problem is caused not only by the lack of the detailed and precise definition of the terms, but also by the *free* style of the documentation. Informal specifications can be written in a manner where every important term is defined in detail, but the free style of writing may make the specification tedious and keep important information hidden among irrelevant details. In fact, a well-organized documentation, even if written in an informal language or notation, can greatly help improve its readability. However, no matter how much the organization is improved in an informal documentation, it is usually impossible to guarantee no misunderstanding occurs because ambiguity
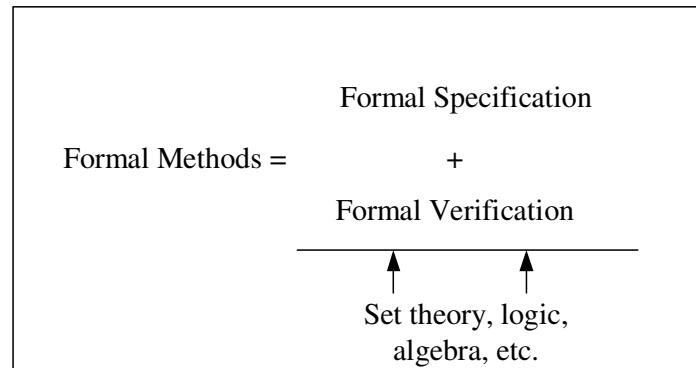
Fig. 1.2. The description of formal methods

is an intrinsic feature of informal languages. Furthermore, in an informal description it is difficult to show the clear relations among different parts of a complicated specification.

A specification should be *consistent* in defining requirements, that is, no contradiction should exist between different requirements in the specification. The specification is also expected to document all the possible user requirements; such a property is called *completeness* of specification. Since informal specifications lack formality in both syntax and semantics, it is usually difficult, even impossible in most cases, to support automated verification of their consistency and completeness in depth. Furthermore, informal specifications offer no firm foundation for design and coding, and for verifying the correctness of implemented programs in general.

1.3 Formal Methods

One way to improve the quality of documentation and therefore the quality of software is to provide formalism in documentation. Such a formalized documentation offers a precise specification of requirements and a firm basis for design and its verification.

1.3.1 What Are Formal Methods

Formal methods for developing computer systems embrace two techniques: *formal specification* and *formal verification* [55, 3, 38, 116, 43]. Both are established based on elementary mathematics, such as set theory, logic, and algebraic theory, as illustrated in Figure 1.2.

Formal specification is a way to abstract the most important information away from irrelevant implementation detail and to offer an unambiguous documentation telling *what is to be done* by the system. A formal specification is
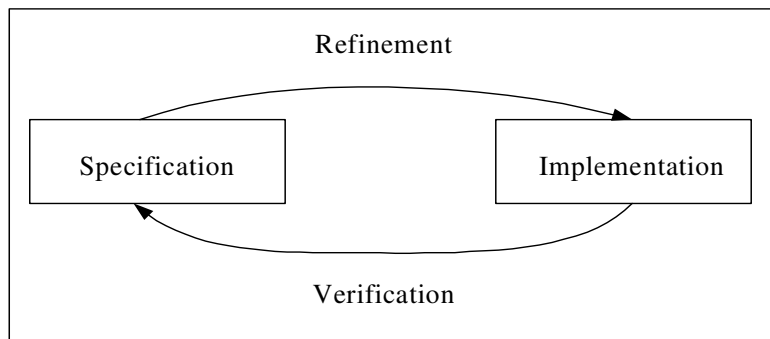
Fig. 1.3. The principle of formal methods

written in a language with formal syntax and semantics. Of course, programming languages are also formal languages, but they are for implementation of computer systems, not for specifications. In a specification language, there is usually a mechanism that allows the definition of what to be done by the system without the need of giving algorithmic solutions, whereas in a programming language all the mechanisms are usually designed for writing algorithmic solutions (i.e., code). For this reason, formal specifications are more concise and abstract than programs.

Formal verification is a way to *prove* the *correctness* of programs against their specifications [42][24][36][101]. A program is correct if it does exactly what the specification requires. The proof of the correctness is usually based on a logical calculus that provides necessary axioms and inference rules. An axiom is a statement of a fact without any hypothesis, while a rule is a statement of a fact under some hypotheses. Program correctness proof aims to establish a logical consistency between the program and its specification.

A *method* offers a way to do something. This is true to formal methods as well. Figure 1.3 shows the principle of formal methods. A specification is constructed first, and then refined into a program by following appropriate refinement rules [90][6]. In general, since this refinement may not be done automatically, the correctness of the program may not be ensured. Therefore, a formal verification of the program against its specification is needed to ensure its correctness. Such a verification may sometimes also help detect faults in the specification.

In principle, the activities of *specification*, *refinement*, and *verification* advocated by formal methods may not necessarily be completed within a single cycle; they are usually applied repeatedly to several level specifications. Thus, an entire software development can be modeled as a successive refinement and verification process, after the informal requirements are formalized into the highest level formal specification and the specification is validated against
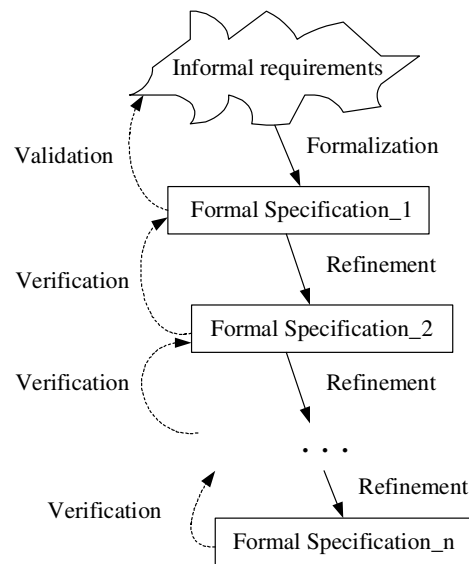
Fig. 1.4. Software development using formal methods

the informal requirements, as illustrated in Figure 1.4. In this model, each level document is perceived as a specification of the next lower level document, and each refinement takes the current level specification more toward the final executable program, represented by the lowest level specification (i.e., formal specification_n). Since refinement is a transitive relation between specifications, the final program must theoretically satisfy the highest level specification (i.e., formal specification_1).

1.3.2 Some Commonly Used Formal Methods

Many formal methods have been reported in the literature so far, such as VDM, Z, B-Method, HOL [35], PVS [20], Larch [39], RAISE [38], and OBJ [34, 31], but in accordance with the international survey on industrial applications of formal methods [19] and the applications described in Hinchey and Bowen's edited book [88], the most commonly used formal mthods include VDM, Z, and B-Method.

VDM (The Vienna Development Method) offers a notation, known as VDM-SL (VDM-Specification Language), and techniques for modelling and designing computing systems. It was originally developed based on the work of the IBM Vienna Laboratory in the middle 1970s. The publication of Jones's book titled "Systematic Software Development using VDM" [54, 55] has contributed considerably to the wide spread of VDM technology in education and application. The most important feature in VDM is the mechanism for defining operations. An operation can be regarded as abstraction of a procedure

in the programming language Pascal (or similar structure in other programming languages) and defined with a precondition and a postcondition. The precondition imposes a constraint on the initial state before operation, while the postcondition presents a constraint on the final state after operation. The most essential technique in writing an operation specification is definition of a relation between the initial and final states in the postcondition of the operation. This technique allows the specification to focus on the description of the function of the operation, and therefore facilitates the clarification of functional requirements before providing them with a program solution. In order to model complex systems, VDM provides a set of built-in types, such as *set*, *sequence*, *map*, and *composite* types. In each type, necessary *constructors* and *operators* are defined, which allow for the formation and manipulation of objects (or values) of the type. Using those built-in types as well as their constructors and operators in specifications, complex functions of operations can be modeled precisely and concisely. With the progress in software supporting tools [29], VDM has been gradually adopted in the development of industrial systems, and has been extended to VDM++ to support object-oriented design [27].

Z was originally designed as a formal notation based on axiomatic set theory and first order predicate logic for describing and modeling computing systems by the Programming Research Group at Oxford University around 1980 [107][100], and later developed to a method by providing rules for refinement and verification [116]. An essential component used in Z specifications is known as *schema*. A schema is a structure that can be used to define either system state or operation. The definition of state includes the declarations of variables and their constraints given as predicate expressions. A schema defining an operation is usually composed of two parts: declarations and predicates. The declarations may include declarations of input variables and/or output variables, as well as the related state schemas. The predicates impose constraints on the input variables, output variables, and the related state variables. Complex specifications can be formed by using the *schema calculus* available in the Z notation. Although Z uses syntax different from VDM, they share the similar model-based approach to writing formal specifications. Based on Z notation, other formal notations have also been developed to support object-oriented design and concurrency, such as Object-Z [105] and TCOZ [81].

The B-method has been developed by Jean-Raymond Abrial. It provides an Abstract Machine Notation for writing system specifications and rules for refinement of specifications into programs [3, 102]. A specification in B is constructed by means of defining a set of related abstract machines. An abstract machine is similar to a module in VDM, which contains local state variables, invariants on the state, and necessary operations. Each machine must have a name in order to allow other machines in a large specification to refer to it. A machine can extend another machine in order to expand its contents (e.g., state and operations) and include another machine in order to allow for calling

of its operations. Following the refinement rules, an abstract specification can be transformed step by step into a concrete representation (or implementation in B termnology) that can then be translated into a program of a specific programming language. With the progress of tools development, the B-Method has been applied in a few industrial projects [44].

1.3.3 Challenges to Formal Methods

In my opinion formal methods have presented the most reasonable, rigorous, and controllable approach to software development so far, at least theoretically, but their application requires high skills in mathematical abstraction and proof. The situation seems that if all the suggested steps in formal methods could be taken in practice, with no compromise, we would have no doubt in the correctness of the program produced. However, since software engineering is a human activity (with support of software tools), the effect of formal methods depends heavily on whether and how they can be applied in practice by software engineers, usually with many constraints. The major challenges are:

- Formal specifications for large-scale software systems are usually more difficult to read than informal specifications, and this would be aggravated for complex systems. Informal specifications are usually easy to read, but offer no guarantee of correct understanding because of ambiguity in language semantics. Formal methods offer precise specifications, but they are difficult to read, and there is no guarantee of correct understanding either. The two cases may result in the similar situation that the reader of the specification would make a guess about the meaning of some expressions, but for different reasons. The specification may be too *imprecise* to be correctly understood in the first case whereas it may be too *difficult* to be correctly understood in the second case.
- Formal verification of program correctness is too expensive to be deployed in practice. Although it is the most powerful technique for demonstrating the consistency between programs and their specifications among existing verification techniques, such as testing, static analysis, animation, and model checking, but only a small number of experts can apply this technique, and it may not be cost-effective for complex systems. Except for safety-critical systems or the safety-critical parts of systems, formal verification is usually out of reach of most software engineers in industry, including even many formal methods researchers.
- Another challenge is that the use of formal methods usually costs more in time and human effort for analysis and design. One of the important reasons is the constant change of requirements during a software development process. When the initial high level specification is written, it is usually incomplete in terms of recording the user requirements. When it is refined into a lower level specification, the two specifications may not

satisfy the refinement rules, not necessarily because the lower level speci-
fication has errors, but rather because the high level specification is often
not sufficiently complete. In this case, the high level specification needs to
be modified or extended in order to reflect the user requirements, discov-
ered during its refinement. Such a modification often occurs, not only to
one level specification, but also to almost every level specification. This
imposes a strong challenge to developers, both in psychology and in cost,
especially when the project is under pressure from the market.

Having given the challenges to formal methods above, we should not deny the
positive role of formal methods. In fact, formal methods have two advantages
over informal ones. One is the high potential for automation in processing for-
mal specifications due to their formally defined syntax and semantics. Another
is that formal methods can work effectively for compact specifications. If one
has experience reading research papers in software engineering or other areas,
one will easily understand that reading a paper full of mathematical definitions
and formulas, with less informal explanations, is much harder than reading
a paper with a proper combination of informal explanations and small-scale
formal descriptions (leaving necessary large-scale formal definitions in the ap-
pendix). Using formal notation in specifications has a similar effect on their
readability. This is an important point about formal methods that has made
us realize the importance of integrating formal methods with commonly used
and comprehensible informal or semi-formal notations in software engineering.
Formal notation can be used for the most critical and lower level components
of a complex system, while a comprehensible notation can be adopted to inte-
grate those formal definitions to form the entire specification, without losing
the focus on *what to do*.

Furthermore, although formal verification may be difficult to be deployed
directly in practice, its principles may be incorporated into existing practical
techniques, such as testing, static analysis, and animation to achieve more
effective verification and validation techniques. It is important to strike a
good balance between rigor and practicality in integrated verification and
validation techniques.

1.4 Formal Engineering Methods

Formal Engineering Methods, FEM for short, are the methods that support
the application of formal methods to the development of large-scale computer
systems. They are a further development of formal methods toward industrial
application. I proposed to use this terminology for the first time in 1997 when
organizing the first International Conference on Formal Engineering Methods
(ICFEM) in Hiroshima [63] and continued to use it in many publications since
then [76, 70, 74, 75, 64, 78, 69].

Formal engineering methods are equivalent neither to application of formal
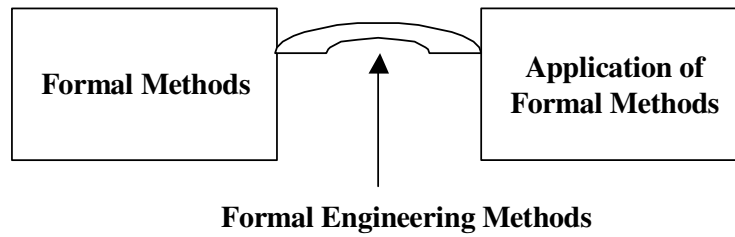methods, nor to formal methods themselves. They are intended to serve as

Fig. 1.5. An illustration of formal engineering methods

a *bridge* between formal methods and their applications, providing methods and related techniques to incorporate formal methods into the entire software engineering process, as illustrated in Figure 1.5. Without such a bridge, application of formal methods is difficult. The quality of the bridge may affect the smoothness of the formal methods technology transfer. Some types of bridges may make the transfer easier than others, so the important point is how to build the bridge.

Similar to formal methods, formal engineering methods are also aimed at attacking the problems in specification and verification of computer systems, but take more practical approaches. In principle, formal engineering methods should allow the following:

- Adopting specification languages that properly integrate graphical notation, formal notation, and natural language. The graphical notation is suitable for describing the overall structure of a specification comprehensibly, while the formal notation can be used to provide precise abstract definition of the components involved in the graphical representation. The interpretation of the formal definitions in a natural language helps understand the formal definitions. Many graphical notations have already been used for requirements analysis and design in practice, such as Data Flow Diagrams (DFDs) [23, 117], Structure Charts [15], Jackson Structure Diagrams [50, 16], and UML (Unified Modeling Language) [30, 18], but most of them are informal or semi-formal. This is the reality, but not necessarily a definitive feature of graphical notation. In fact, a graphical notation can also be treated as formal notation, as long as it is given a precise syntax and semantics. Compared with textual mathematical notation, a graphical notation is usually easier to read, but it usually takes more space than textual notation, and perhaps drawing diagrams is less efficient than typing in textual notation. Therefore, an appropriate integration can create a comfortable ground for utilizing the advantages of both graphical notation and formal notation.
- Employing rigorous but practical techniques for verifying and validating specifications and programs. Such techniques are usually achieved by integrating formal proof and commonly used verification techniques, such

as testing [108, 72, 91], reviews [94, 97], and model checking [49, 17]. The integrated techniques must take a proper approach to make good use of the strong points of the techniques involved and to avoid their weaknesses.

- Advocating the combination of prototyping and formal methods. A computer system has both dynamic and static features. The dynamic feature is shown only during the system operation, such as the layout of the graphical user interface, usability of the interface, and performance. The requirements for these aspects of the system are quite difficult to capture without actually running the system or its prototype. For this reason, prototyping – the development of an executable model of the system can be effective in capturing the user requirements for some of the dynamic features in the early phases of system development. The result of prototyping can serve as the basis for developing an entire system using formal methods, focusing on the functional behaviors of the system. Of course, sometimes prototyping can go along, in parallel, with the development using formal methods.
- Supporting evolution rather than strict refinement in developing specifications and programs [57, 109, 82, 73, 7]. Evolution of a specification, at any level, means a change, and such a change does not necessarily satisfy the strict refinement rules (of course, it sometimes does). The interesting point is how to control, support, and verify changes of specification during software development in a practical manner. Although some of these issues are still open to be resolved, they have been increasingly paid attention to by researchers.
- Deploying techniques for constructing, understanding, and modifying specifications. For example, effective techniques for specification construction can be achieved by integrating existing requirements engineering techniques with formal specification techniques [77], and techniques in simulation and computer vision can be combined to form visualized simulation to help specification understanding, and so on.

In summary, formal engineering methods embrace integrated specification, integrated verification, and all kinds of supporting techniques for specification construction, transformation, and system verification and validation. They can be simply described as

FEM = Integrated specification +
Integrated verification +
Supporting techniques

Note that formal engineering methods are a collection of specific methods, so we should not expect a single formal engineering method to cover all the features given previously.

1.5 What Is SOFL

SOFL, standing for *S*tructured *O*bject-Oriented *F*ormal *L*anguage, is a formal engineering method. It provides a formal but comprehensible language for both requirements and design specifications, and a practical method for developing software systems. The language is called *SOFL specification language*, while the method is called *SOFL method*. Unless there is the need of clear distinction, SOFL is used to mean either the language or the method or both throughout this book, depending on the context.

SOFL is designed by integrating different notations and techniques on the basis that they are all needed to work together effectively in a coherent manner for specification constructions and verifications. The SOFL specification language has the following features:

- It integrates Data Flow Diagrams [23], Petri nets [12], and VDM-SL (Vienna Development Method - Specification Language) [54, 55, 110]. The graphical notation Data Flow Diagrams are adopted to describe comprehensibly the architecture of specifications; Petri nets are primarily used to provide an operational semantics for the data flow diagrams; and VDM-SL is employed, with slight modification and extension, to precisely define the components occurring in the diagrams. A formalized Data Flow Diagram, resulting from the integration, is called *Condition Data Flow Diagram*, or CDFD for short. It is always associated with a *module* in which its components, such as processes (describing an operation), data flows (describing data in motion), and data stores (describing data at rest), are formally defined. In semantics, the CDFD associated with a module describes the behavior of the module, while the module is an encapsulation of data and processes, with an overall behavior represented by its CDFD. Furthermore, the use of a natural language, such as English, is facilitated to provide comments on the formal definitions in order to improve the readability of formal specifications [76, 41, 26].
- Condition data flow diagrams and their associated modules are organized in a hierarchy to help reduce complexity and to achieve modularity of specifications. Such a hierarchy is formed by decomposition of processes. A process is decomposed into a lower level CDFD and its associated module when the details of how to transform its input to output needs to be spelled out.
- *Classes* are used to model complicated *data flows and stores*. A store is like a file or database in many computer systems; it offers data that can be accessed by processes in a CDFD or by different CDFD in the hierarchy. The value of a store can be used and changed by processes. If the changes are made by processes at different levels, it will be difficult to control the changes. For this reason, a store can be modeled as an instance of a class. A class is a specification for its instances or objects that contains definitions of attributes and methods (similar to processes, but with constraints). Any change of the attributes of an instance must be made by its own methods.
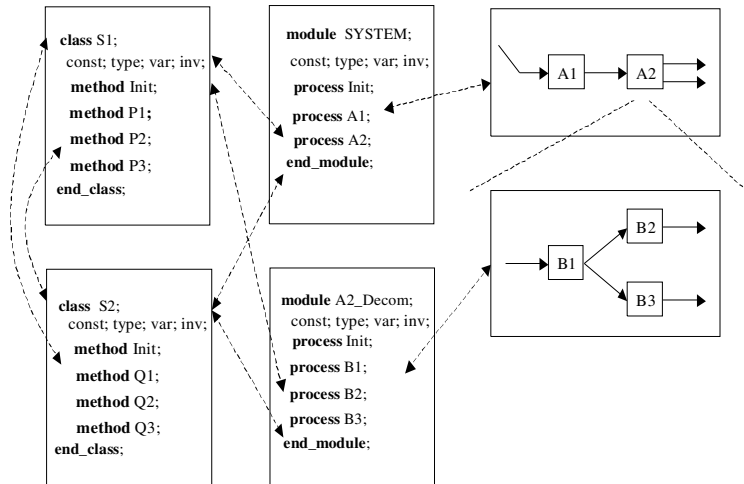
Fig. 1.6. An outline of a specification in SOFL

Modules and classes are similar in their internal structures, but different in the way used in specifications. A module represents a decomposition of a high level process and has an overall behavior. No instance can be derived from a module; therefore, a module cannot be used as a type to declare variables. On the other hand, objects may be instantiated from a class that may offer many individual behaviors, as services, and are used to model a data flow or store in CDFDs.

Figure 1.6 shows an outline of a specification in SOFL. The hierarchy of CDFDs and modules contains two CDFDs and associated modules. Each small rectangle in the CDFDs denotes a process, and each directed line represents a data flow. The CDFD involving processes A1 and A2 is the top level CDFD, corresponding to the module SYSTEM. In this module, the functions of A1 and A2 are formally defined. In addition, process Init is provided for the initialization of the local data stores (which are not given in this abstract figure) and necessary declarations are given. For some reason process, A2 is decomposed into the CDFD containing processes B1, B2, and B3, and its associated module, named A2_Decom, provides formal definitions of its processes, data flows, and so on. For the specification of processes in the hierarchy of CDFDs, classes S1 and S2 are defined; they may be used in both modules, SYSTEM and A1_Decom.

The SOFL method has the following features:

- It integrates *structured methods* and *object-oriented methods* for specification construction, in order to utilize their advantages and to avoid their disadvantages. Structured methods are a top-down approach by which the construction of a specification starts from the top level module, and then

proceeds by decomposing high level operations defined in the modules into low level modules. The structured methods are usually intuitive for requirements analysis and design, because their way of documentation is consistent with the way in which people think in developing and organizing large-scale projects, such as building a bridge, launching a rocket, or making an aircraft. On the other hand, object-oriented methods are basically a bottom-up approach to software development. In this approach the low level classes are first built, and then they are composed to form more complicated classes. Furthermore, an object-oriented approach is effective in achieving system properties, such as encapsulation of data and operations, inheritance, and polymorphism. These properties are very important in achieving the qualities of information hiding, software reuse, and maintainability. However, this approach may be less intuitive than structured methods for requirements analysis and design. The integration of these two different but related approaches in SOFL offers a way to effectively support functional decomposition and object composition. The specifications are easy to be translated into commercially object-oriented programming languages, such as C++ [61] and Java [22].

- It supports a *three-step approach* to developing formal specifications. Such a development is an *evolutionary* process, starting from an informal specification, to a semi-formal one, to finally a formal specification. The informal specification, usually written in a natural language, serves as the basis for deriving the semi-formal specification in which SOFL syntax, to a certain extent, is enforced. The formal specification is then derived from the semi-formal specification by formalization of the informal parts in the semi-formal specification.

  By considering the roles of requirements and design specifications, SOFL advocates the idea that requirements specifications are written in a semiformal manner, while design specifications need to be completely formal. The obvious reason for this is that requirements specifications are often used for communication between the user and the developer, which requires the comprehensibility of documentation, while the primary role of design specification is to provide an unambiguous ground for implementation. Furthermore, the construction of design specification requires study of requirements given in the requirements specifications, and formalization can greatly help in this regard.

  An evolution of specification is a change, which can be a *refinement*, *extension*, or *modification* [66]. The evolution approach is suited to developing design specifications on the basis of semi-formal requirements specifications, since it usually results in many changes in the specifications. But for implementation from a design specification, refinement must be enforced, since we must make sure that the implementation does exactly same thing required by the design. For the details of this approach, see Chapter 14.

- It adopts *rigorous review* and *testing* for specification verification and validation. Specification verification aims to detect faults in specifications. Rigorous review is a technique resulting from the integration of formal proof and fault tree analysis, a method for safety analysis. The reviews must be done on a precise ground, and supported by a rigorous mechanism [67][68]. They are usually less formal than formal proof, but easy to conduct.

  Testing can be applied to both specifications and programs. Since some formal specifications are not executable, the testing needs a special technique [72]. The test cases used for specification can be reused for black-box testing of programs [91]. For the detailed discussions of these techniques, see Chapters 17 and 18.

When building a specific software system, the techniques supported by SOFL can be used with flexibility, depending on the application domain. For critical systems, such as safety- and security-critical systems, a profound use of formal notation, rigorous testing, and rigorous review are recommended. But for less critical systems, semi-formal notation and reasonably rigorous verification may be sufficient.

1.6 A Little History of SOFL

The initial development of SOFL was made at the University of Manchester in the United Kingdom in 1989, when I was studying for a doctoral degree in formal methods. The motivation was to integrate the most well-known formal method, VDM at that time, with traditional DFDs to support the application of formal methods in industry. I strongly believed, and still do now, that software development is not a pure mathematical process, although the relation between specifications and programs can be interpreted mathematically. It is, in fact, a highly disciplined *human* activity featured by creativity and constant changes, although it is likely supported by software tools. If any powerful method wants to be accepted by practitioners at large, it must provide a user-friendly interface and effective mechanism to facilitate the structuring of large-scale systems. On the other hand, informal methods that have been using in practice offer no guarantee for the quality of software systems. It was my belief that it is necessary to develop a kind of formal method from the *engineer's* point of view, and a *proper* combination of formal, semi-formal, and informal notations can possibly provide a good solution.

I chose VDM and DFDs for three reasons. One is that both are appropriate notations to describe "*what to do*" rather than "*how to do it*," but on different level. In DFDs this feature is reflected by focusing on data flows among processes (rather than control flows in algorithms), while in VDM it is featured by using pre- and postconditions for operation specifications. Another reason is that VDM lacks an effective and comprehensible structuring

mechanism to allow a large specification to be formed by integrating different operations. Although the notion *module* is used to organize operations in specifications, its expressive power and scale-up ability are limited. In addition to this weakness, the readability of large-scale specifications may not be satisfactory. However, it became quite clear to me after a period of study that VDM and DFDs are complementary in providing rigorous and comprehensible specifications, and that the notation for operation specification in VDM is well suited to describing specifications for processes used in DFDs. This provided the third reason for the integration. The language resulting from this research was called FGSL, standing for *F*ormal *G*raphical *S*tructured *L*anguage.

FGSL was evolved continuously later on, by combining my experiences gained from several projects on formal methods and safety-critical systems at the University of York, RHBNC of London University, Hiroshima City University, The Queen's University of Belfast, Oxford University, and Hosei University. It was an important step when FGSL was developed into SOFL by integrating the structured method and object-oriented method on the project titled "Formal Methods and Intelligent Software Engineering Environments" sponsored by the Ministry of Education, Culture, Sports, Science and Technology of Japan in 1996. It was an international joint project involving the researchers from several universities in Japan, USA, UK, and Australia. Since then, SOFL has been improved after being applied to the modeling or development of some critical systems and information systems on national and international projects [74, 75, 78, 62, 71].

1.7 Comparison with Related Work

It is quite difficult within a section to give a comprehensive comparison of SOFL with all the existing work on integration of formal methods and informal or semi-formal methods. To help the reader understand the commonality and difference between SOFL and other related work, we try to focus on the work that attempts to integrate model-oriented formal methods (e.g., VDM, Z, Alloy [48]) and semi-formal methods (e.g., data flow diagrams, UML).

From late 1980s more and more researchers began to realize the importance of combining formal and informal methods, and proposed several different approaches to integrating formal specification languages with informal notations (and associated methods). The approach taken by most researchers for integration is to use the Yourdon or the DeMarco approach to constructing data flow diagrams and their associated data dictionaries for expressing high level user requirements, and then to refine the data flow diagrams into formal specifications by defining data flows, necessary processes, and their integration with formal notation. The examples of this approach include Semmens and her colleagues' work on integrating Yourdon's data flow diagrams and Z [103], Bryant's work on Yourdon's method and Z [14], Plat and his colleagues' integration of data flow diagrams and VDM [96], and Fraser's work on data

flow diagrams and VDM [80]. In contrast to this approach, SOFL is aimed at achieving both the improvement of structuring mechanism in the VDM specification language for modularity and the comprehensibility of the ultimate specifications. This target is realized by incorporating classical data flow diagram notation into a formal specification language to provide a decompositional method for structuring system specifications and a graphical view for the system specifications. In this way, data flow diagrams are treated as part of formal specifications. Although adopting a rather different data flow model for describing computer systems, Broy and Stolen's FOCUS formalism [13] shares the idea of employing visual formal notation in specifications. However, the major difference between FOCUS and SOFL is that the former tries to provide a mathematical and logical foundation for the specification and refinement of interactive systems, while the latter emphasizes the techniques for incorporating formal specification and verification into the entire software development process to improve the quality of the software process and to achieve the practicality of formal methods.

Apart from the integration of formal methods and the structured method based on the data flow paradigm, much work has also been done in combining formal notations with the object-oriented paradigm or notation for concurrency to improve the rigor of object-oriented development or concurrent development. Examples of this approach include VDM++ [27], Object-Oriented Z [85], TCOZ [81, 25], and OCL [112], the Object Constraint Language of UML (Unified Modelling Language) [93, 18, 30]. Although SOFL also adopts object-oriented features, such as class and object, class inheritance, and polymorphism, it emphasizes a quite different development paradigm than UML in that the structured method is mainly used for user requirements analysis and abstract design specification in order to effectively capture the desired functions and the overall architecture of the system, while the object-oriented method is mainly used for detailed design and implementation to achieve good maintainability and reusability of the system. Another distinct feature of SOFL is that it emphasizes a balance between and compatibility with graphical notation and formal notation: it advocates the use of both formal and graphical notations for good readability and efficiency in constructing specifications, but does not encourage concentration on the use of only one of them.

Developing practical techniques for verification and validation of software systems based on formal specification and proof has also been an intensively researched area. The proposed techniques include *specification animation* [40, 89], *model checking* [17, 5], *specification-based testing* [108, 104, 113, 91, 92], and *software review, inspection, and analysis* [94, 87, 79, 21]. Since we take the view in SOFL that harmony among methods, tools, and human developers is the key to the success of software projects, we adopt the most practical techniques, software review and testing, for verification and validation, although the specific methods for review and testing may be different from traditional approaches. In our methods, we emphasize utilizing formal specification and

proof principle to achieve *rigor* for the practical review and testing techniques, as well as their supportability using software tools.

1.8 Exercises

1. Answer the following questions:
   a) What is the software life cycle?
   b) What is the problem with informal approaches to software development?
   c) What are formal methods?
   d) What are the major features of formal engineering methods?
   e) What is SOFL?
2. Explain the role of specification in software development.
3. Give an example of using a principle similar to formal methods to build other kinds of systems rather than software systems.