

Shaoying Liu

Formal Engineering for Industrial Software Development

Using the SOFL Method

Erratum

Due to an error that occurred during the data conversion process, the lettering in the figures on

page 2, Fig. 1.1,
page 97, Fig. 4.37,
page 230, Fig. 13.3,
page 238, Fig. 14.2,
page 262, Fig. 15.1,
page 353, Fig. 19.2

are not printed properly or are garbled. Please find attached the correct versions of the figures.

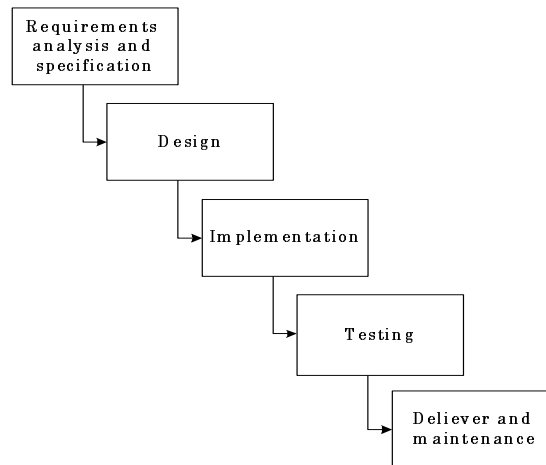


Fig. 1.1. The waterfall model for software development

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.

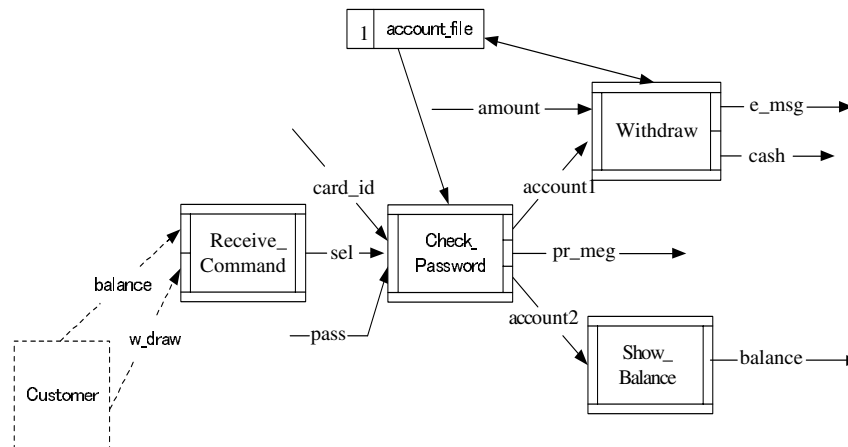


Fig. 4.37. The CDFD with an external process

entity may be a person, machine, organization, a group, or any object with the function of providing useful data information to the system concerned. Since such entities may not be suitable for being part of the system, we need to distinguish them from the normal processes used in the system. We call such entities *external processes*, because their behavior can be modeled as a process. An external process is represented graphically by a dashed-line box, such as the external process **customer** in the CDFD given in Figure 4.37.

Since external processes are not treated the same as normal processes in CDFDs, their syntax and semantics do not need to conform to the rules for normal processes; they are just designed to provide useful information about the system to help communication between the developer and the user via the CDFDs. For example, the external process in Figure 4.37 is named **customer**, because this information can help us to understand who provides the command for displaying the balance of or withdrawing cash from the account.

4.13 Associating CDFD with a Module

A graphical notation like CDFD is comprehensible, but may not be capable of defining all the components precisely for the sake of the space it occupies. As we have described previously, the components of a CDFD can be formally defined. To organize all the formal definitions related to the CDFD, the concept of a module is provided.

A module is an encapsulation of data and processes with a behavior represented by the CDFD it associates with. Generally speaking, a module has the following structure:

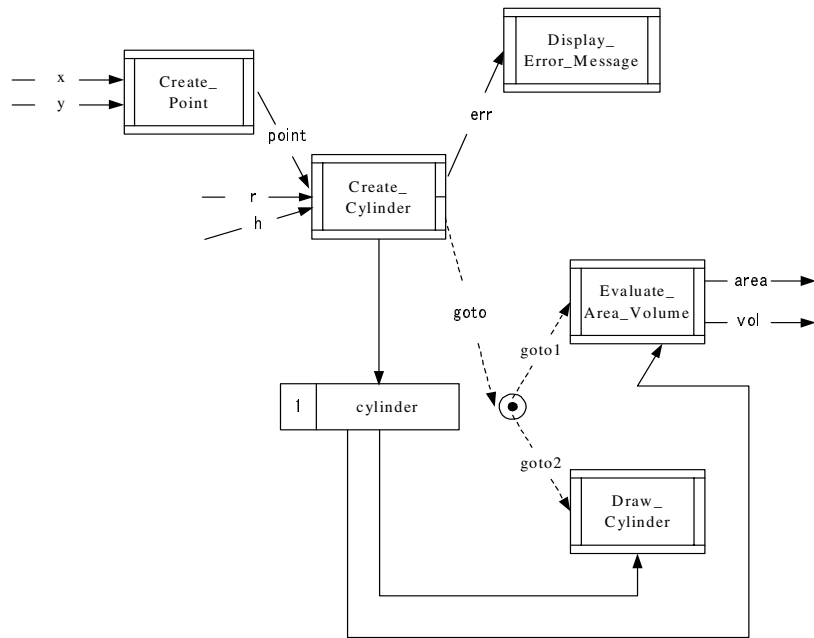


Fig. 13.3. A module using objects of classes

by process `Create_Point` based on the input coordinates `x` and `y`. Then, store `cylinder`, an object of class `Cylinder`, is created by process `Create_Cylinder`, based on the inputs `point`, `r` (radius), and `h` (height). If radius `r` or height `h` is not positive, error message `err` is produced; otherwise, control data flow `goto` is made available. The copies of this data flow, `goto1` and `goto2`, are then used as the inputs of processes `Evaluate_Area_Volume` and `Draw_Cylinder`, respectively. Process `Evaluate_Area_Volume` calculates the surface area and volume of the `cylinder`, while `Draw_Cylinder` draws the `cylinder` on an output device (e.g., screen).

The specification of the module whose behavior is described by the CDFD in Figure 13.3 is given below. For constructing this module, the classes `Cylinder` and `Point` are assumed to have been defined previously. Thus, they can be directly used in the module.

```

module Cylinder_Test;
var
  cylinder: Cylinder;
  /*class Cylinder is assumed to have already been defined.*/
process Init()
explicit
  cylinder := new Cylinder

```

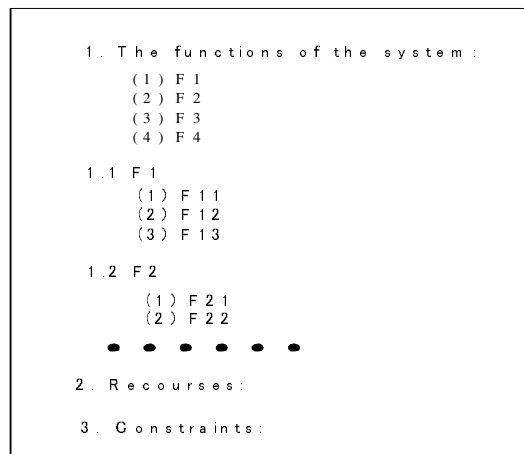


Fig. 14.2. The outline of an informal specification

This informal specification consists of three parts: the required functions, the resources of the hotel to be managed, and the constraints reflecting the policy of the hotel. The functions includes the room reservation, cancellation of reservations, change of reservations, checking in, and checking out services. The room resources which the hotel manages include 100 single rooms, 50 twin rooms, and 50 double rooms. The hotel has a policy requiring that a customer can reserve only one room each time and no customer can check into the hotel without reservation (e.g., for security reasons).

This example only shows an abstract idea of how to organize an informal specification. For a more complex system, some of the functions may indicate a complex task. In that case, those functions may need to be described in detail, indicating their sub-functions. For this purpose, each function can be taken as an abstraction of a module in the specification. There is no need to strictly follow the syntax of the module in this stage, but conceptually each function being treated as a module will help in the creation of the semi-formal specification in the next stage. For example, if function **Reserve room** needs more detailed description, it can be written as follows:

1.1 Reserve room includes the functions:

- Check the vacancy.
- Register the customer on the reservation list.
- Issue the reservation number.

In a similar way, the other functions in this example can also be decomposed into detailed sub-functions, if necessary. Thus, the entire informal specification may take the form illustrated in Figure 14.2.

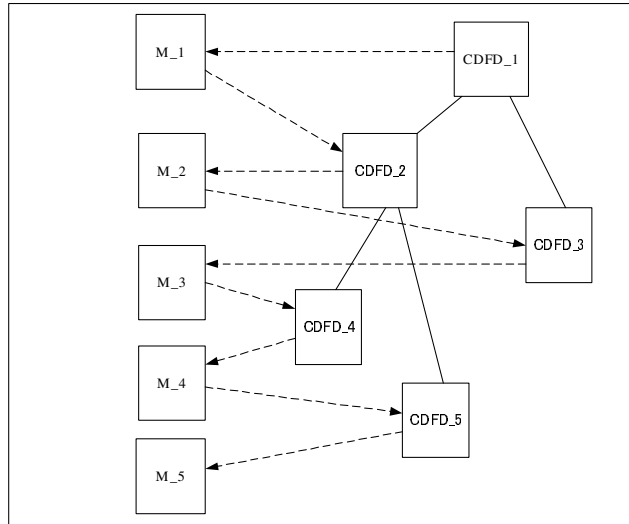


Fig. 15.1. An illustration of CDFD-module-first strategy

emphasizes the importance of the architecture of the system in providing an outline for formalization in the modules.

15.1.1 The CDFD-Module-First Strategy

The fundamental idea of this strategy is that after a CDFD is constructed, its associated module must be defined precisely, before any decomposition of processes in this CDFD takes place. After both the CDFD and module are finalized, the decomposition of another process can take place. Such a process goes on until no process needs further decomposition. Figure 15.1 depicts this strategy. CDFD_1 is the top level CDFD of the specification, and its two processes are decomposed into CDFD_2 and CDFD_3, respectively. Furthermore, two processes of CDFD_2 are decomposed into CDFD_4 and CDFD_5, respectively. Taking the CDFD-module-first strategy, this specification is constructed by first drawing CDFD_1, and then defining module M_1. Then, a process in CDFD_1 is decomposed into CDFD_2, and the associated module M_2 is defined. This process continues until all the CDFDs and their modules are defined.

Since a CDFD usually represents only an outline of an idea, formed on the basis of an initial consideration, it is usually subject to modification when the precise picture of its components and their relations becomes clear. For this reason, before taking any further action in decomposing processes, ensuring desirable components and structure of the current CDFD is important. This can be achieved by defining the associated module of the CDFD. In addition, defining the module may also result in the following effects:

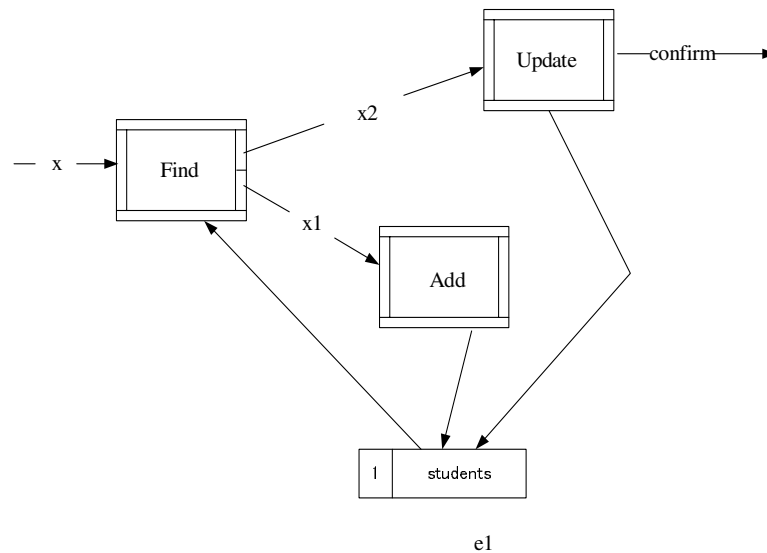


Fig. 19.2. The CDFD of module `Student_Management`

That is, the class hierarchy in the specification must be transformed to a proper class hierarchy in the program. Furthermore, there is no need to take into account the CDFD of a source class, if it is given, in the transformation, because semantically it does not represent anything additional, but is just “syntactical sugar” to depict the source methods and their relations with the state variables of the class.

Consider the transformation of the module `Student_Management` and the class `Student`, given below, as an example. We assume that the module `Student_Management` is a decomposition of a process defined in the high level module `Faculty_System`, which is presumably defined somewhere in the specification. The CDFD given in Figure 19.2 describes the behavior of the module `Student_Management`: when a student `x` is available, the process `Find` tries to find `x` in the store `students`; If `x` exists, it is passed to process `Update` through `x2`; otherwise, `x` is passed to the process `Add` through variable `x1`.

Furthermore, a class `Student` is defined as a subclass of an already defined class `Person`, and used for defining the type `Students` as a set of `Student` objects.

```

module Student_Management / Faculty_System;
const
  PI = 3.14159;
type
  Students = set of Student;
var

```

Formal Engineering for Industrial Software Development
Using the SOFL Method

Liu, S.

2004, XXII, 408 p., Hardcover

ISBN: 978-3-540-20602-6