6.3 DATA FLOW DIAGRAMS (DFDs)

The DFD (also known as the *bubble chart*) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism—it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (Figure 6.1) to represent the functions performed by a system and the

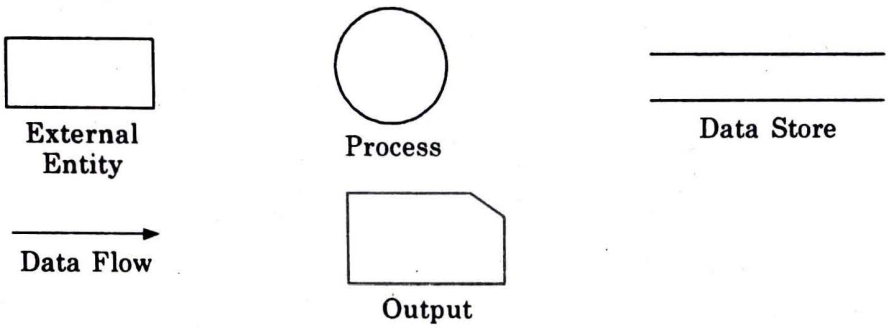


FIGURE 6.1 Symbols used for designing DFDs.

data flow among these functions. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various subfunctions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system—because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system can be slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. We now first explain the different symbols and then elaborate the various concepts associated with building a DFD model of a system.

6.3.1 Primitive Symbols Used for Constructing DFDs

Figure 6.1 depicts five different types of primitive symbols used for constructing DFDs. The meaning of each of these symbols is explained below:

Function symbol. A function is represented using a circle. This symbol is called a process or a *bubble*. Bubbles are annotated with the names of the corresponding functions (see Figure 6.4).

External entity symbol. An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software.

Data flow symbol. A directed arc or an arrow is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes, or between an external

entity and a process, in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names (see Figure 6.4).

Data store symbol. A data store represents a logical file. It is represented using two parallel lines. A logical file can represent either a data store symbol which can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting to a data store need not be annotated with the name of the corresponding data items.

Output symbol. The output symbol, as shown in Figure 6.1, is used when a hard copy is produced and the user of the copies cannot be clearly specified or there are several users of the output.

The notations that we are following in this text are closer to the Yourdon's notations than to the other notations. You may sometimes find notations in other books that are slightly different from those discussed here. For example, the data store may look like a box with one end closed. That is because, some authors may be following different notations such as those of Gane and Sarson [1979].

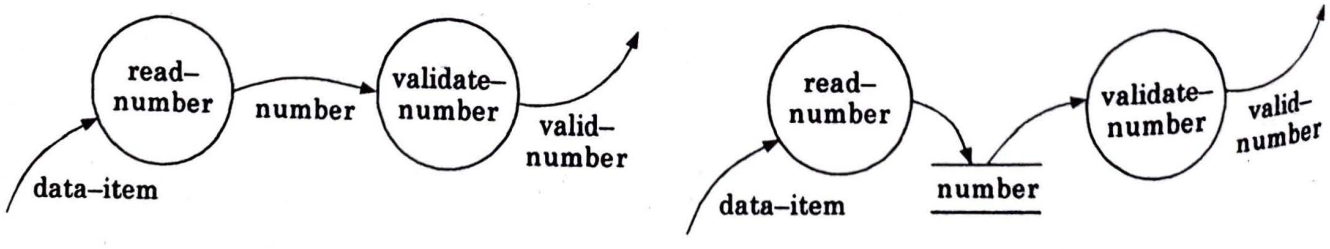
6.3.2 Some Important Concepts Associated with Designing DFDs

Before we describe how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs.

Synchronous and asynchronous operations

If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figue 6.2(a). Here, the validate number bubble can start processing only after the read-number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.

However, if two bubbles are connected through a data store, as in Figure 6.2(b), then the speeds of operation of the bubbles are independent. The data produced by a producer bubble gets stored in the data store. The producer bubble may store several pieces of data items in the data store before the consumer bubble consumes any of them.



(a) Synchronous operation of two bubbles

(b) Asynchronous operation of two bubbles

FIGURE 6.2 Synchronous and asynchronous data flow.

Data dictionary

Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in the DFD model of a system. It may be recalled that the DFD model of a system typically consists of several DFDs, namely level 0 DFD, level 1 DFD, level 2 DFDs, etc. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data *grossPay* consists of the components *regularPay* and *overtimePay*.

$$grossPay = regularPay + overtimePay$$

For the smallest units of data items, the data dictionary lists their name and their type. The operators using a composite data item can be expressed in terms of their component data items, and these are discussed in the next subsection.

A data dictionary plays a very important role in any software development process because of the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by all engineers working in the same project. A consistent vocabulary for data items is very important, since in large projects different engineers have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

For large systems, the data dictionary can be extremely complex and voluminous. Even moderate-sized projects can have thousands of entries in the data dictionary. It becomes extremely difficult to maintain a voluminous dictionary manually. Computer-Aided Software Engineering (CASE) tools come handy to overcome this problem. Most CASE tools usually capture the data items appearing in a DFD, and automatically generate the data dictionary. These tools also support some query language facility to raise queries about the definition and usage of data items. For example, queries may be formulated either to determine which data item affects which processes, which process affects which data items, or to find the definition and usage of specific data items, and so on. Query handling is facilitated by storing the data dictionary in a Relational Database Management System (RDBMS).

Data definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

`+`: denotes composition of two data items, e.g. `a+b` represents data `a` and `b`.

`[,,]`: represents selection, i.e. any one of the data items listed inside the square bracket can occur. For example, `[a,b]` represents either `a` occurs or `b` occurs.

( ): the contents inside the bracket represent optional data which may or may not appear. a+(b) represents that either a occurs or a+b occurs.

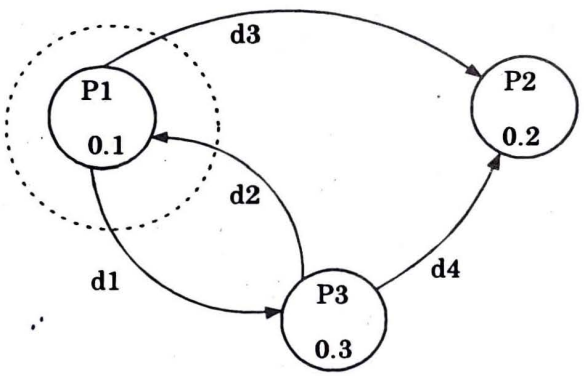
{}: represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.

=: represents equivalence, e.g. a=b+c means that a represents b and c.

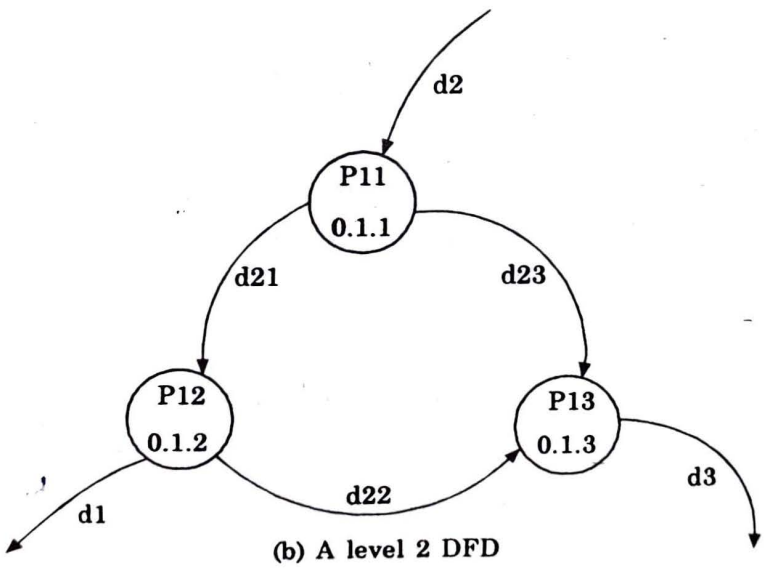
/* */: Anything appearing within /* and */ is considered as comment.

Balancing DFDs

The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as *balancing* a DFD. We illustrate the concept of balancing a DFD in Figure 6.3. In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble P1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed. The decomposition is balanced as d1 and d3 flow out of the level 2 diagram and d2 flows in. Note that the dangling arrows represent the data that flows into or out of a diagram.



(a) A level 1 DFD



(b) A level 2 DFD

FIGURE 6.3 An example showing balanced decomposition.

6.3.3 Developing the DFD Model of a System

A DFD model of a system graphically depicts the transformation of the data input to the system to the final result through a hierarchy of levels. A DFD starts with the most abstract definition of the system (lowest level) and at each higher level DFD, more details are successively introduced. To develop a higher-level DFD model, processes are decomposed into their sub-processes and the data flow among these sub-processes are identified.

To develop the data flow model of a system, first the most abstract representation of the problem is worked out. The most abstract representation of the problem is also called the context diagram. After, developing the context diagram, the higher level DFDs are developed.

Context diagram

The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labelled according to the main function of the system. The various external entities with which the system interacts and the data flows occurring between the system and the external entities are also represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names. The name context diagram is well justified because it represents the context in which the system is to exist, i.e. the external entities (users) who would interact with the system and the specific data items they would be supplying to the system and the data items they would be receiving from the system. The context diagram is also called the level 0 DFD.

To develop the context diagram of the system, we have to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also includes the external systems which supply data to or receive data from the system.

The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is in contrast with the bubbles in all other levels which are annotated with verbs. This is to be expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.

Level 1 DFD

To develop the Level 1 DFD, examine the high-level functional requirements. If there are between three to seven high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD. We can then examine the input data to these functions and the data output by these functions, and represent them appropriately in the diagram.

If a system has more than seven high-level requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the level 1 DFD. These can be split in the lower DFD levels. If a system has less than three high-level functional requirements, then some of the high-level requirements need to be split into their subfunctions so that we have roughly about five to seven bubbles on the diagram.

Decomposition

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD. Decomposition of a bubble is also known as *factoring* or *exploding* a bubble. Each bubble at any level of DFD is usually decomposed to anything between three to seven bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than seven bubbles at any level of a DFD make the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

We can now describe how to go about developing the DFD model of a system more systematically.

1. The SRS document is examined to determine:
   - Different high-level functions that the system needs to perform
   - Data input to every high-level function
   - Data output from every high-level function
   - Interactions (data flow) among the identified high-level functions

   These aspects of the high-level functions are then represented in a diagrammatic form. This forms the top-level Data Flow Diagram (DFD), usually called the DFD 0.
2. The high-level functions described in the SRS document are examined. If there are between three to seven high-level requirements in the SRS document, then each of the high-level function can be represented in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.
3. Each high-level function is decomposed into its constituent subfunctions through the following set of activities:
   - Different subfunctions of the high-level function are identified
   - Data input to each of these subfunctions is identified
   - Data output from each of these subfunctions is identified
   - Interactions (data flow) among these subfunctions are identified

   These aspects are then represented in a diagrammatic form using a DFD.
4. Step 3 is repeated recursively for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD to uniquely identify all bubble in the DFD. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc.

When a bubble numbered x is decomposed, its children bubble are numbered $x.1$, $x.2$, $x.3$, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Commonly made errors while constructing a DFD model

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore helpful to understand the different types of mistakes that beginners usually make while constructing the DFD model of systems.

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. The context diagram should depict the system as a single bubble.
- Many beginners have external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear at other levels of the DFD.
- It is a common oversight to have either too less or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed, i.e. each bubble should be decomposed to between three and seven bubbles.
- Many beginners leave different levels of DFD unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD. It is important to realize that a DFD is the data flow representation of a system, and it does not represent control information. The following examples represent some mistakes of this kind:
  * A book can be searched in the library catalogue by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalogue, then an error message is generated. While developing the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in Figure 6.4) to indicate that the error function is invoked after the search–book. But, this is a control information and should not be shown on the DFD.
  * Another error is trying to represent when or in what order different functions (processes) are invoked and not the conditions under which different functions are invoked.

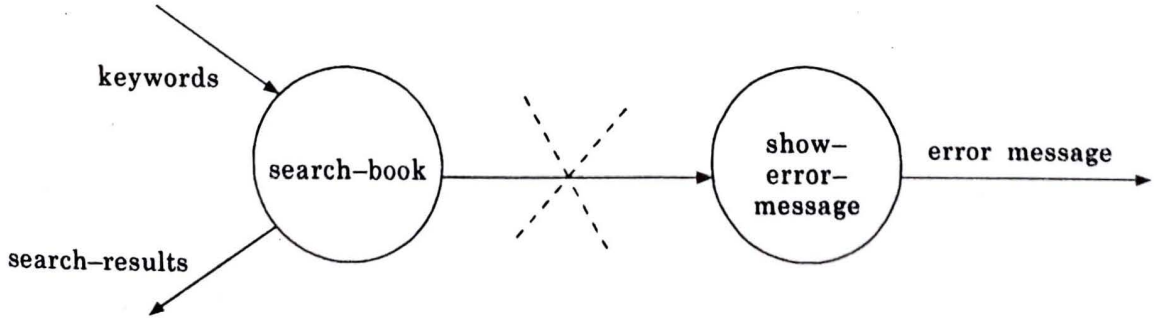


FIGURE 6.4 It is incorrect to show control information on a DFD.

* If a bubble *A* invokes either the bubble *B* or the bubble *C* depending upon some conditions, we need only to represent the data that flows between bubbles *A* and *B* or between bubbles *A* and *C* and not the conditions depending on which the two modules are invoked.

- A data store should be connected only to bubbles through the data flow arrows. A data store cannot be connected either to another data store or to an external entity.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in the SRS document of the system should be overlooked.
- Only the functions of the system specified in the SRS document should be represented, i.e. the designer should not assume any functionality of the system that is not specified by the SRS document and also not try to represent it in the DFD.
- Improper or unsatisfactory data dictionary.
- The data and function names must be intuitive. Some students and even practising engineers use symbolic data names such as a, b, c, etc. Such names hinder proper understanding of the DFD model.

We now illustrate the structured analysis technique through a few examples.

Example 6.1 RMS Calculating Software

A software system called RMS calculating software reads three integral numbers from the user in the range between –1000 and +1000 and determines the root mean square (rms) of the three input numbers and then displays it.

In this example, the context diagram is simple to draw. The system accepts three integers from the user and returns the result to him. To draw the level 1 DFD, from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform—accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result. After representing these four functions in Figure 6.5, we observe that the calculation of root mean square essentially consists of the functions: calculate the squares of the input numbers, calculate the mean, and finally calculate the root. This decomposition is shown in the level 2 DFD in Figure 6.5.

Data dictionary for the DFD model of Example 6.1

```
data-items:  {integer}3
rms:  float
valid-data:  data-items
a:  integer
b:  integer
```

```
c:   integer
asq:  integer
bsq:  integer
csq:  integer
msq:  integer
```

Example 6.1 is an almost trivial example and is only meant to illustrate the basic methodology. In practical structured analysis of non-trivial systems, decomposition is never carried on up to the basic instruction level as done in Example 6.1. On the other hand, a bubble is not decomposed any further once it is found to represent a simple set of instructions. Now, let us perform the structured analysis for a more complex problem.

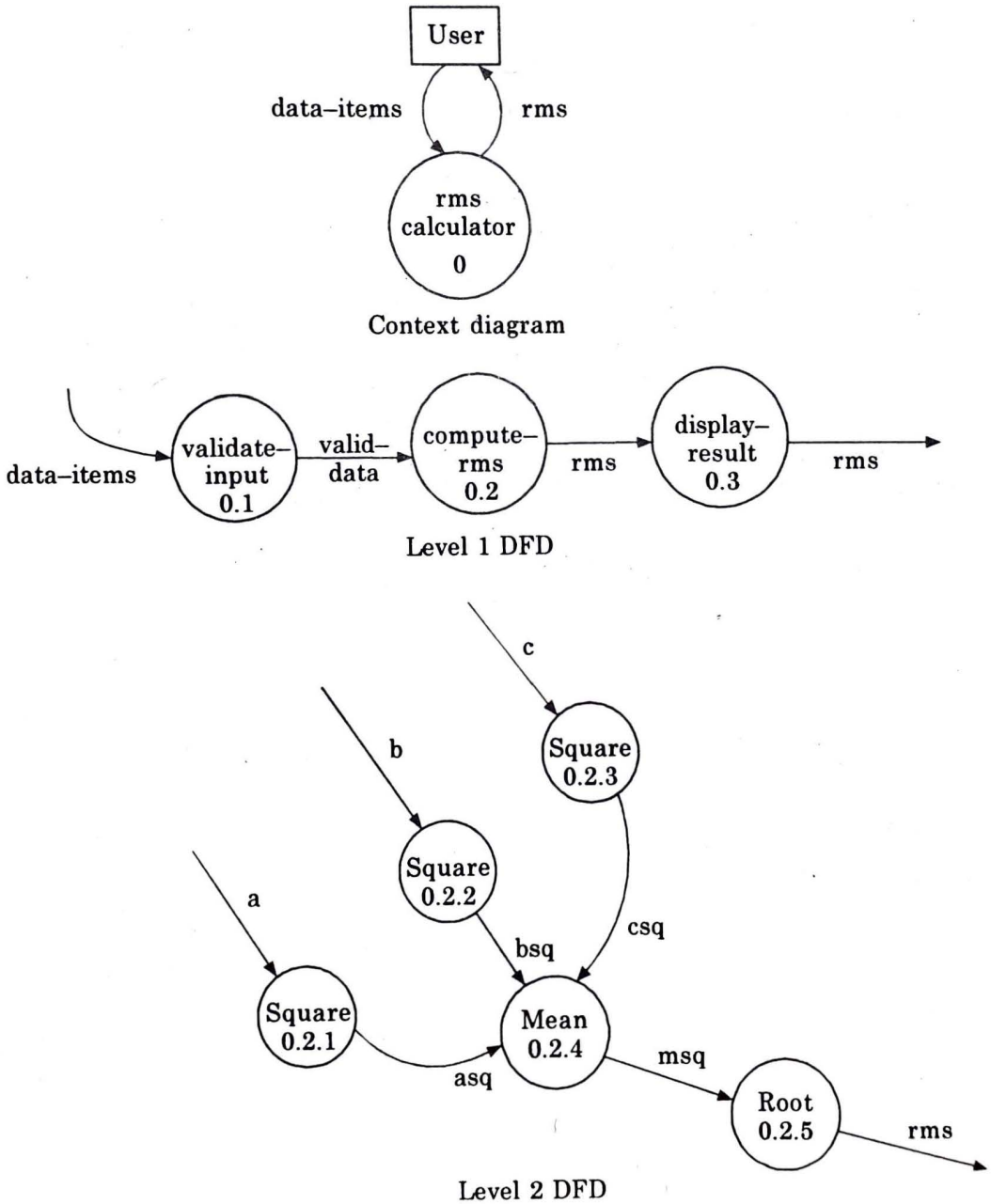
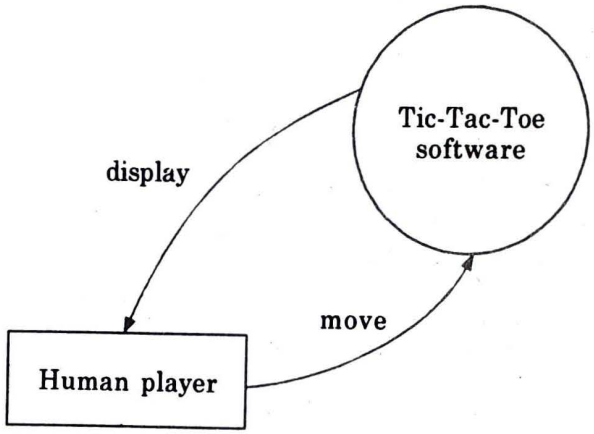


FIGURE 6.5 Context diagram, level 1 and level 2 DFDs for Example 6.1.

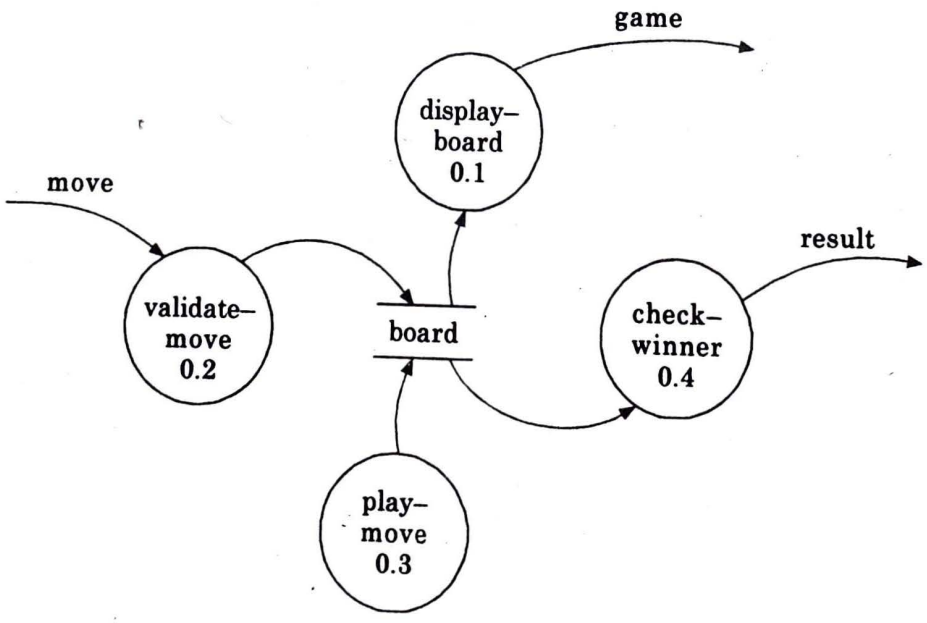
Example 6.2 Tic-Tac-Toe Computer Game

Tic-Tac-Toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. A move consists of marking a previously unmarked square. The player who is first to place three consecutive marks along a straight line (i.e. along a row, column, or diagonal) on the square wins. As soon as either the human player or the computer wins, a message congratulating the winner is displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The context diagram and the level 1 DFD are shown in Figure 6.6.



Context diagram



Level 1 DFD

FIGURE 6.6 Context diagram and level 1 DFDs for Example 6.2.

Data dictionary for the DFD model of Example 6.2

```
move: integer /* number between 1 to 9 */
```

```
display: game+result
game: board
board: {integer}9
result: ["computer won", "human won", "drawn"]
```

Example 6.3 Supermarket Prize Scheme

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his residence address, telephone number, and the driving licence number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check-out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket awards surprise gifts to 10 customers who make the highest total purchase over the year. Also, it awards a 22 carat gold coin to every customer whose purchases exceed Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated.

The context diagram for the supermarket prize scheme problem of Example 6.3 is shown in Figure 6.7, the level 1 DFD in Figure 6.8, and the level 2 DFD in Figure 6.9.

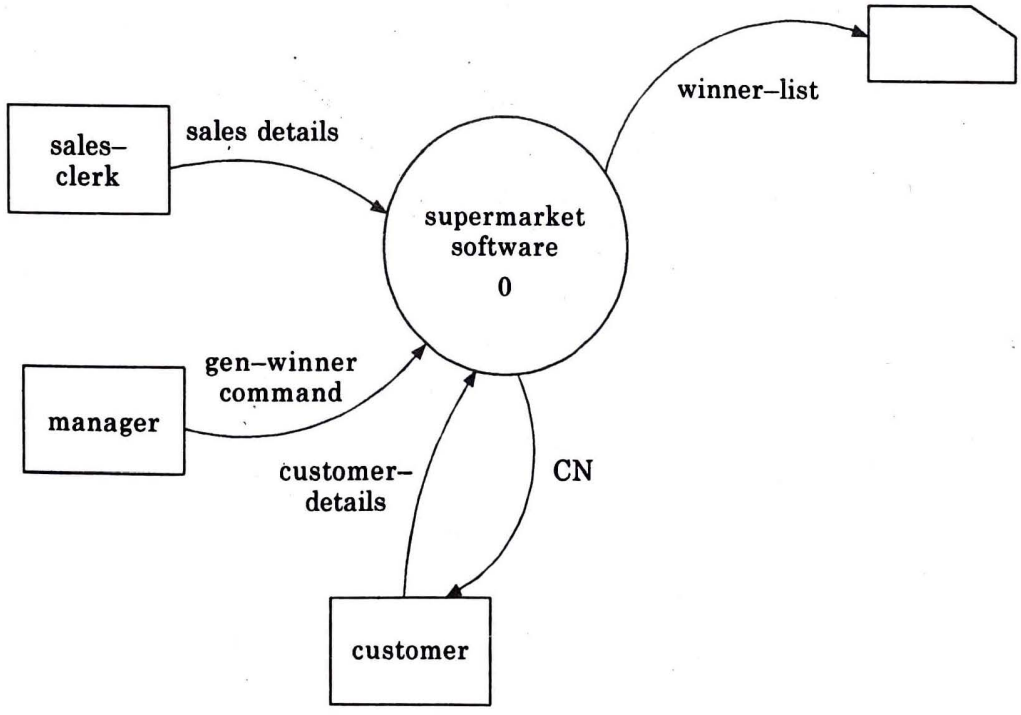


FIGURE 6.7 Context diagram for Example 6.3.

Data dictionary for the DFD model of Example 6.3

```
address: name+house#+street#+city+pin
sales-details: {item+amount}* + CN
CN: integer
customer-data: {address+CN}*
sales-info: {sales-details}*
```

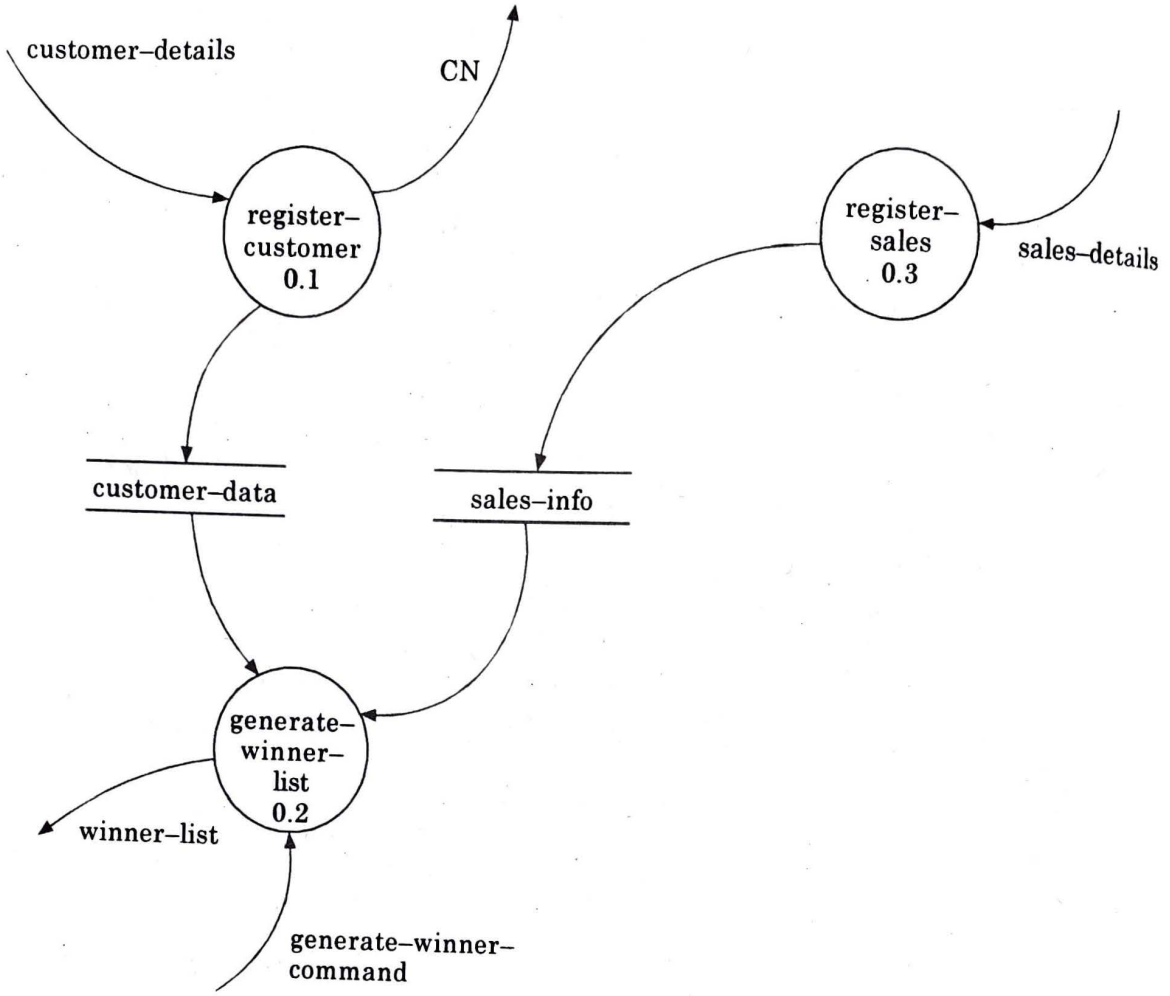


FIGURE 6.8 Level 1 diagram for Example 6.3.

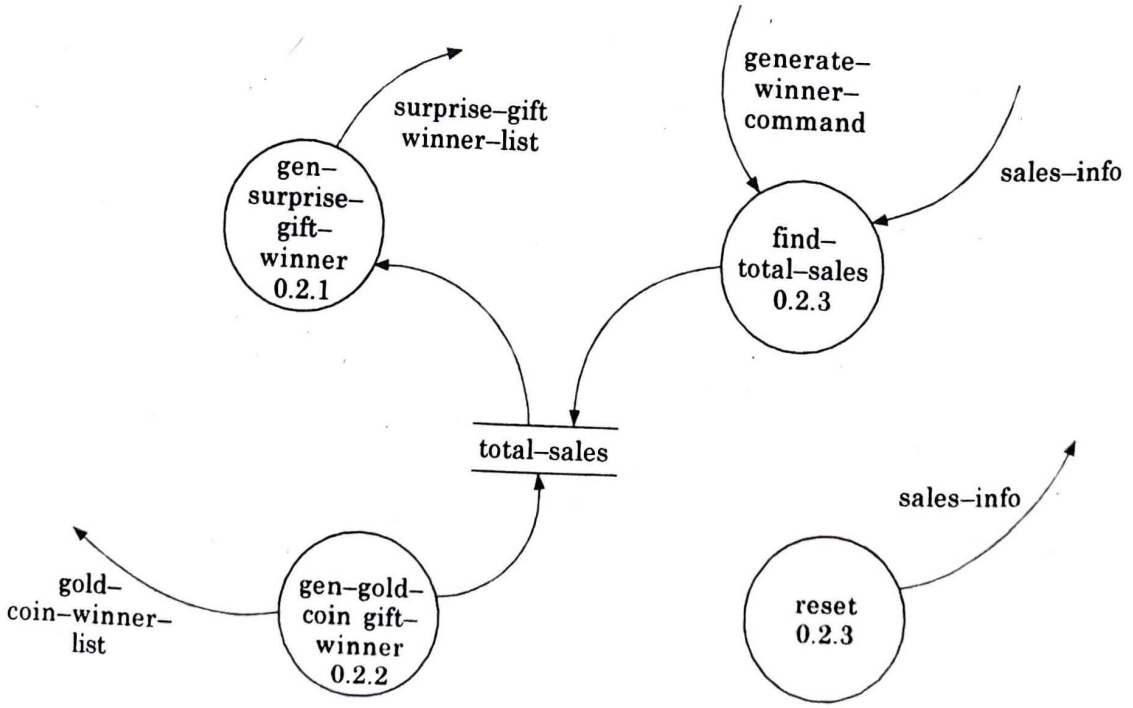


FIGURE 6.9 Level 2 diagram for Example 6.3.

```
winner-list: surprise-gift-winner-list + gold-coin-winner-list
surprise-gift-winner-list: {address+CN}*
gold-coin-winner-list: {address+CN}*
gen-winner-command: command
total-sales: {CN+integer}*
```

Observation. It may be noted that the data `generate-winner-list` is in fact pseudo data and represents control information. We have included it in the diagram to facilitate the structured design process discussed later. We could have also as well done without the `generate-winner-list` data.

Example 6.4 Trading-House Automation System (TAS)

A trading house wants us to develop a computerized system that would automate various bookkeeping activities associated with its business. The following are the salient features of the system to be developed:

- The trading house has a set of regular customers. The customers place orders with it for various kinds of commodities. The trading house maintains the names and addresses of its regular customers. Each of these regular customers should be assigned a unique customer identification number (CIN) by the computer. The customers quote their CIN on every order they place.
- Once an order is placed, as per current practice, the accounts department of the trading house first checks the credit-worthiness of the customer. The credit-worthiness of the customer is determined by analyzing the history of its payments to different bills sent to him in the past. After automation, this task has to be done by the computer.
- If a customer is not credit-worthy, his orders are not processed any further and an appropriate order rejection message is generated for the customer.
- If a customer is credit-worthy, the items that he has ordered are checked against a list of items that the trading house deals with. The items in the order which the trading house does not deal with, are not processed any further and an appropriate apology message to the customer for these items is generated.
- The items in the customer's order that the trading house deals with are checked for availability in the inventory. If the items are available in the inventory in desired quantity, then
  - * a bill with the forwarding address of the customer is printed.
  - * a material issue slip is printed. The customer can produce this material issue slip at the storehouse and take delivery of the items.
  - * inventory data is adjusted to reflect the sale to the customer.
- If any of the ordered items are not available in the inventory in sufficient quantity to satisfy the order, then these out-of-stock items along with the quantity ordered by the customer and its CIN are stored in a "pending-order" file for further

processing to be carried out when the purchase department issues the "generate indent" command.

- The purchase department should be allowed to periodically issue commands to generate indents. When a command to generate indents is issued, the system should examine the "pending-order" file to determine the orders that are pending and to determine the total quantity required for each of the items. It should find out the addresses of the vendors who supply these items by examining a file containing vendor details and then should print out indents to these vendors.
- The system should also answer managerial queries regarding the statistics of different items sold over any given period of time and the corresponding quantity sold and the price realized.

The context diagram for the trading house automation problem is shown in Figure 6.10, and the level 1 DFD in Figure 6.11.

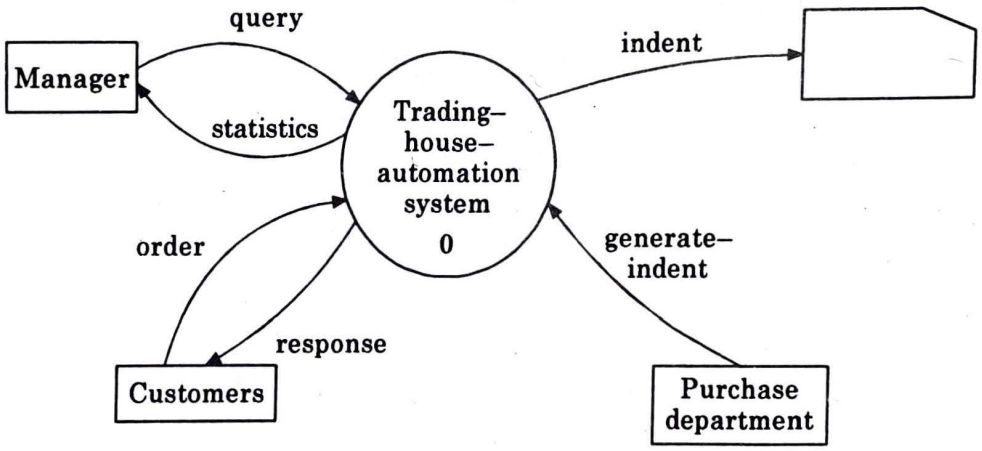


FIGURE 6.10 Context diagram for Example 6.4.

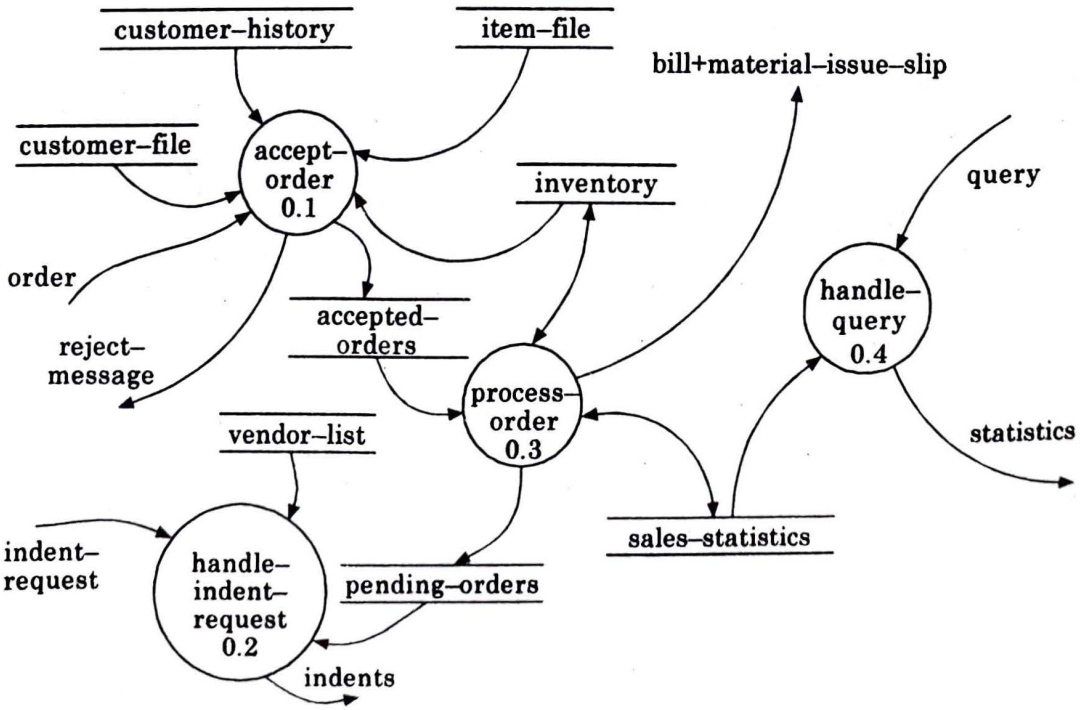


FIGURE 6.11 Level 1 DFD for Example 6.4.

Data dictionary for the DFD model of Example 6.4

```
response: [bill + material-issue-slip, reject-message]
query: period /* query from manager regarding sales statistics*/
period: [date+date,month,year,day]
date: year + month + day
year: integer
month: integer
day: integer
order: customer-id + {items + quantity}* + order#
accepted-order: order /* ordered items available in inventory */
reject-message: order + message /* rejection message */
pending-orders: customer-id + {items+quantity}*
customer-address: name+house#+street#+city+pin
name: string
house#: string
street#: string
city: string
pin: integer
customer-id: integer
customer-file: {customer-address}*
bill: {item + quantity + price}* + total-amount + customer-address + order#
material-issue-slip: message + item + quantity + customer-address
message: string
statistics: {item + quantity + price}*
sales-statistics: {statistics}* + date
quantity: integer
order#: integer /* unique order number generated by the program */
price: integer
total-amount: integer
generate-indent: command
indent: {item+quantity}* + vendor-address
indents: {indent}*
vendor-address: customer-address
vendor-list: {vendor-address}*
item-file: {item}*
item: string
indent-request: command
```

Observations. The following observations can be made from Example 6.4.

1. In a DFD, if two data stores deal with different types of data, e.g. one type of data is invariant with time whereas another varies with time (e.g. vendor address, and inventory data) it is a good idea to represent them as separate data stores. The inventory data changes each time fresh supply arrives or when an item is sold (and the inventory is updated), whereas the vendor data remains unchanged.

2. If we are developing the DFD model of a process which is already being manually carried out, then the names of the registers being maintained in the manual process would appear as data stores in the DFD model. For example, if TAS is currently being manually carried out, then normally there would be registers corresponding to accepted orders, pending orders, vendor list, etc.
3. We can observe that DFDs enable a software engineer to develop the data domain and the functional domain models of the system at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition. At the same time, the DFD refinement automatically results in refinement of the corresponding data items.

6.3.4 Shortcomings of the DFD Model

DFD models suffer from several shortcomings. The major shortcomings of DFD models are the following:

- DFDs leave ample scope to be imprecise. In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named `find-book-position` has only intuitive meaning and does not specify several things, e.g. what happens when some input information is missing or is incorrect. Further, the `find-book-position` bubble may not convey anything regarding what happens when the required book is missing.
- Control aspects are not defined by a DFD. For instance, the order in which the inputs are consumed and the outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modelling real-time systems.
- The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.
- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to then use subjective judgment to carry out decomposition.

6.4 EXTENDING DFD TECHNIQUE TO REAL-TIME SYSTEMS

A real-time system is one where the functions must not only produce the correct result but also should produce them by some pre-specified time. For real-time systems since reasoning about time is important to come up with a correct design, explicit representations of control and event flow aspects are essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the Ward and Mellor technique [1985]. In the Ward and Mellor notation, a type of process that handles only control flows is introduced.

These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Unlike Ward and Mellor, Hatley and Pirbhai [1987] show the dashed and solid representations on separate diagrams. To be able to separate the data processing and the control processing aspects, a Control Flow Diagram (CFD) is defined. This reduces the complexity of the diagrams. In order to link the data processing and control processing diagrams, a notational reference (solid bar) to a control specification is used. The CSPEC describes the following:

- The effect of an external event or control signal
- The processes that are invoked as a consequence of an event

Control specifications represent the behaviour of the system in two different ways:

- It contains a state transition diagram (STD). The STD is a sequential specification of behaviour.
- It contains a program activation table (PAT). The PAT is a combinatorial specification of behaviour. PAT represents invocation sequence of bubbles in a DFD.

6.5 STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. A structure chart represents the software architecture, i.e. the various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks which are used to design structure charts are the following:

Rectangular boxes. A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.

Module invocation arrows. An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a module calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.

Data flow arrows. These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. The data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.

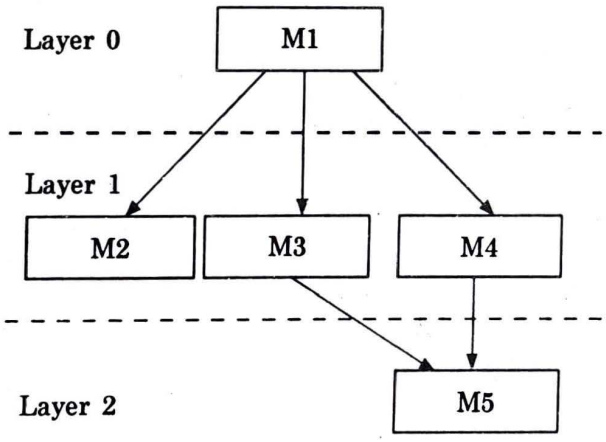
Library modules. A library module is usually represented by a rectangle with double

edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.

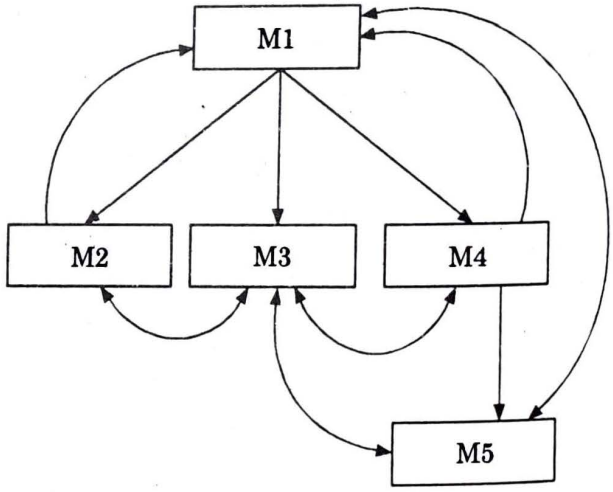
Selection. The diamond symbol represents the fact that one module out of several modules connected with the diamond symbol is invoked depending on the outcome of the condition attached to the diamond symbol.

Repetition. A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

In any structure chart, there should be one and only one module at the top, called the *root*. There should be at most one control relationship between any two modules in the structure chart. This means that if module *A* invokes module *B*, module *B* cannot invoke module *A*. The main reason behind this restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow the lower-level modules to be aware of the existence of the higher-level modules. However, it is possible for two higher-level modules to invoke the same lower-level module. Examples of a properly layered design and another of a poorly layered design are shown in Figure 6.12.



(a) A properly layered design



(a) A poorly layered design

FIGURE 6.12 Examples of properly and poorly layered designs.

6.5.1 Flow Chart vs. Structure Chart

We are all familiar with the flow chart representation of a program. A flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify different modules of the software from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

6.5.2 Transformation of a DFD Model into a Structure Chart

Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart:

- Transform analysis
- Transaction analysis

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. In the following, we elaborate these two techniques.

In order to determine whether transform or transaction analysis is applicable to a DFD, one has to observe the data input to the diagram. The data input to the diagram can be easily spotted because these are represented by dangling arrows. If all the data flows into the diagram are processed in similar ways (i.e. if all the input data are incident on the same bubble in the DFD) then transform analysis is applicable. Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very small problems or at the lower levels of a DFD model. Each different way in which data is processed corresponds to a separate transaction. Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

Transform analysis

Transform analysis identifies the primary functional components (modules) and the high level input and outputs for these components. The first step in transform analysis is to divide the DFD into three types of parts:

- Input
- Logical processing
- Output

The input portion in the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical form (e.g. internal tables, lists, etc.). Each input portion is called an *afferent branch.*

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch.* The remaining portion of a DFD is called *central transform.*

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the central transform, and the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them

are not central transforms. Processes which sort input or filter data from it are central transforms. The first level of structure chart is produced by representing each input and output unit as boxes and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called *factoring*. Factoring includes adding read and write modules, error-handling modules, initialization and termination processes, identifying consumer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example 6.5

Draw the structure chart for the rms software of Example 6.1.

By observing the level 1 DFD of Figure 6.5, we can identify validate-input as the afferent branch and write-output as the efferent branch and the remaining (i.e. compute-rms) as the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart as shown in Figure 6.13.

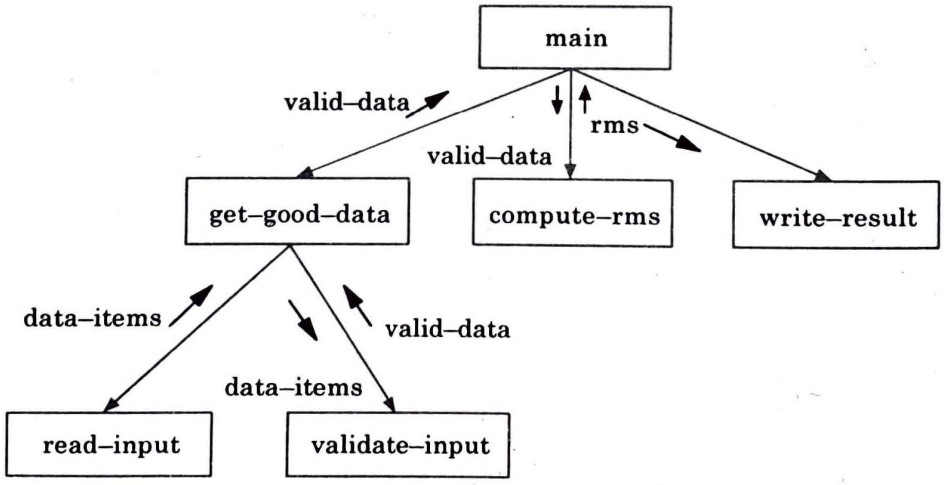


FIGURE 6.13 Structure chart for Example 6.5.

Example 6.6

The structure chart for the Tic-Tac-Toe software of Example 6.2 is shown in Figure 6.14.

Transaction analysis

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centred system which is characterized by identical processing steps for each data item. Each

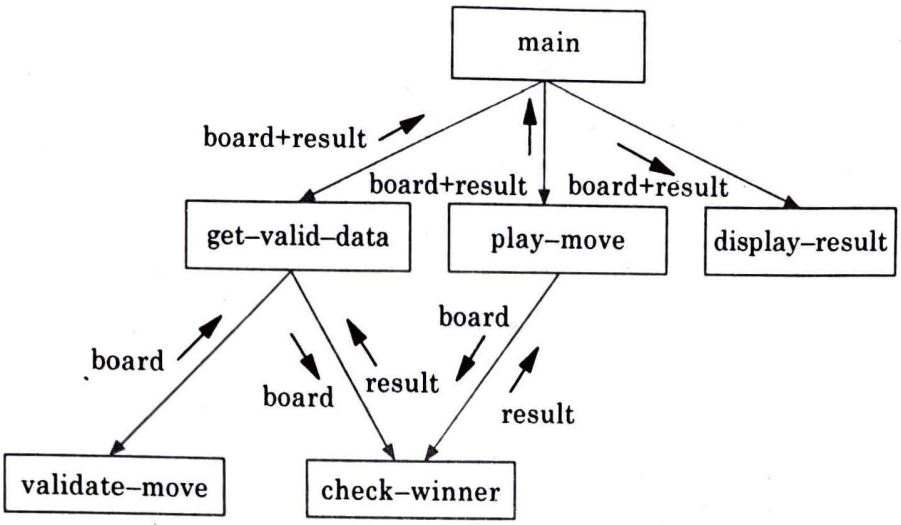


FIGURE 6.14 Structure chart for Example 6.6.

different way in which input data is handled is a transaction. A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified from the experience of solving a large number of examples.

For each identified transaction, we trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, we draw a root module and below this module we draw each identified transaction of a module. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction-centre module.

Example 6.7

The structure chart for the Supermarket Prize Scheme software of Example 6.3 is shown in Figure 6.15.

Example 6.8

The structure chart for the Trading-House Automation System (TAS) software of Example 6.4 is shown in Figure 6.16.

By observing the level 1 DFD of Figure 6.8, we can see that the data input to the diagram are handled by different bubbles and therefore transaction analysis is applicable to this DFD. Input data to this DFD are handled in three different ways (accept-order, accept-indent-request, and handle-query); we have three different transactions corresponding to these as shown in Figure 6.16.

6.6 DETAILED DESIGN

During detailed design the pseudo code description of the processing and the different data

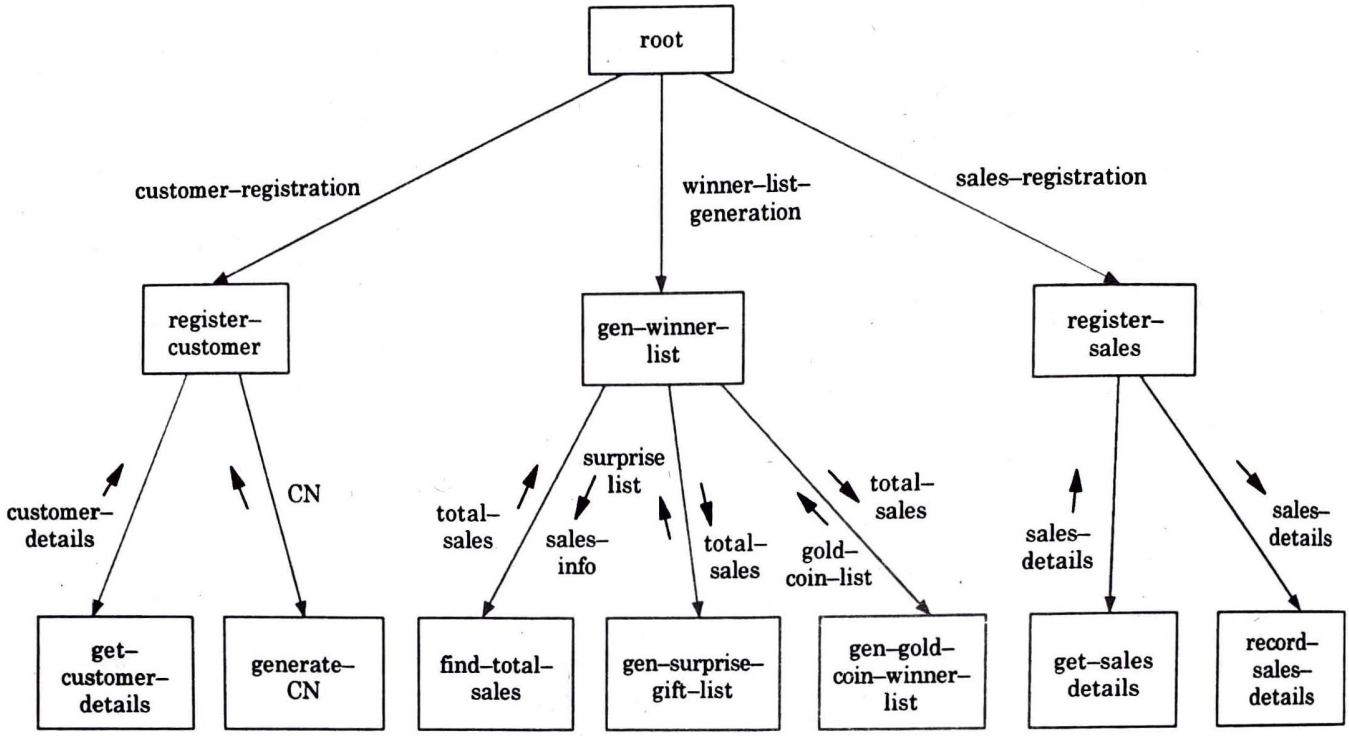


FIGURE 6.15 Structure chart for Example 6.7.

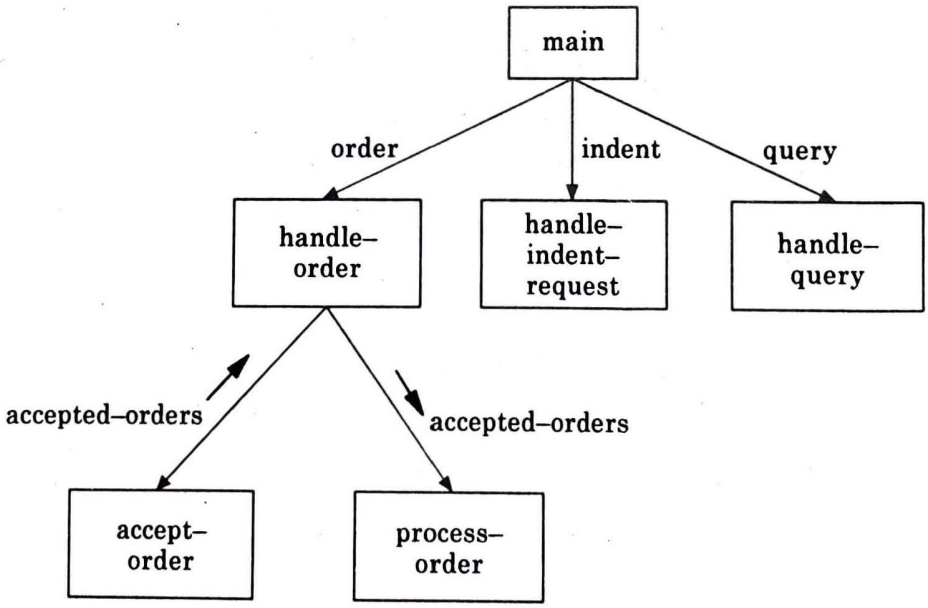


FIGURE 6.16 Structure chart for Example 6.8.

structures are designed for different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). The MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules. The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

6.7 DESIGN REVIEW

After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team. Normally, the members of the team who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents for the following:

Traceability. The members of the team check whether each bubble of the DFD can be traced to some module in the structure chart and vice versa. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and vice versa.

Correctness. They check whether all the algorithms and data structures of the detailed design are correct.

Maintainability. They check whether the design can be easily maintained in future.

Implementation. They check whether the design can be easily and efficiently implemented.

SUMMARY

- In this chapter, we discussed a sample function-oriented software design methodology called Structured Analysis/Structured Design (SA/SD) which incorporates features of some important design methodologies.
- Methodologies like SA/SD give us a recipe for developing a good design according to the different goodness criteria we discussed in Chapter 5.
- SA/SD consists of two important parts: structured analysis and structured design.
- The goal of structured analysis is to perform a functional decomposition of the system. Results of structured analysis are represented using Data Flow Diagrams (DFDs). The DFD representation is difficult to implement using a traditional programming language. The DFD representation can be systematically transformed to structure chart representation. The structure chart representation can be easily implemented using a conventional programming language.
- During structured design, the DFD representation obtained during structured analysis is transformed to a structured chart representation.
- Several CASE tools are available to support the software design process carried out using the important function-oriented design methodologies. In addition to laying out the DFDs, structure charts, maintaining the data dictionary, and helping in traceability analysis, these CASE tools can also perform some elementary consistency checking, e.g. they can usually check whether a DFD is balanced or not.

3

Agile software development

Objectives

The objective of this chapter is to introduce you to agile software development methods. When you have read the chapter, you will:

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan-driven development;
- know the key practices in extreme programming and how these relate to the general principles of agile methods;
- understand the Scrum approach to agile project management;
- be aware of the issues and problems of scaling agile development methods to the development of large software systems.

Contents

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions, and the emergence of competing products and services. Software is part of almost all business operations so new software is developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid development and delivery is therefore now often the most critical requirement for software systems. In fact, many businesses are willing to trade off software quality and compromise on requirements to achieve faster deployment of the software that they need.

Because these businesses are operating in a changing environment, it is often practically impossible to derive a complete set of stable software requirements. The initial requirements inevitably change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, the requirements are likely to change quickly and unpredictably due to external factors. The software may then be out of date when it is delivered.

Software development processes that plan on completely specifying the requirements and then designing, building, and testing the system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually prolonged and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, a plan-driven approach is the right one. However, in a fast-moving business environment, this can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

The need for rapid system development and processes that can handle changing requirements has been recognized for some time. IBM introduced incremental development in the 1980s (Mills et al., 1980). The introduction of so-called fourth-generation languages, also in the 1980s, supported the idea of quickly developing and delivering software (Martin, 1981). However, the notion really took off in the late 1990s with the development of the notion of agile approaches such as DSDM (Stapleton, 1997), Scrum (Schwaber and Beedle, 2001), and extreme programming (Beck, 1999; Beck, 2000).

Rapid software development processes are designed to produce useful software quickly. The software is not developed as a single unit but as a series of increments, with each increment including new system functionality. Although there are many approaches to rapid software development, they share some fundamental characteristics:

1. The processes of specification, design, and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to

implement the system. The user requirements document only defines the most important characteristics of the system.

2. The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version. They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface. The system may then generate a web-based interface for a browser or an interface for a specific platform such as Microsoft Windows.

Agile methods are incremental development methods in which the increments are small and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

3.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, the use of analysis and design methods supported by CASE tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems.

This software was developed by large teams working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. These plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design has to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led a number of software developers in the 1990s to propose new 'agile methods'. These allowed the development team to focus on the software itself rather than on its design

3.3.1 Testing in XP

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

To avoid some of the problems of testing and system validation, XP emphasizes the importance of program testing. XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system.

The key features of testing in XP are:

1. Test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

Test-first development is one of the most important innovations in XP. Instead of writing some code and then writing tests for that code, you write the tests before you write the code. This means that you can run the test as the code is being written and discover problems during development.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. This approach can be adopted in any process in which there is a clear relationship between a system requirement and the code implementing that requirement. In XP, you can always see this link because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation. The adoption of test-first development in XP has led to more general test-driven approaches to development (Astels, 2003). I discuss these in Chapter 8.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of 'test-lag'. This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

User requirements in XP are expressed as scenarios or stories and the user prioritizes these for development. The development team assesses each scenario and breaks it down into tasks. For example, some of the task cards developed from the story card for prescribing medication (Figure 3.5) are shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Figure 3.1 The principles of agile methods

As I discuss in the final section of this chapter, the success of agile methods has meant that there is a lot of interest in using these methods for other types of software development. However, because of their focus on small, tightly integrated teams, there are problems in scaling them to large systems. There have also been experiments in using agile approaches for critical systems engineering (Drobna et al., 2004). However, because of the need for security, safety, and dependability analysis in critical systems, agile methods require significant modification before they can be routinely used for critical systems engineering.

In practice, the principles underlying agile methods are sometimes difficult to realize:

1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and cannot take full part in the software development.
2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.
3. Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.

5. Many organizations, especially large companies, have spent years changing their culture so that processes are defined and followed. It is difficult for them to move to a working model in which processes are informal and defined by development teams.

Another non-technical problem—that is a general problem with incremental development and delivery—occurs when the system customer uses an outside organization for system development. The software requirements document is usually part of the contract between the customer and the supplier. Because incremental specification is inherent in agile methods, writing contracts for this type of development may be difficult.

Consequently, agile methods have to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. So long as all goes well, this benefits both the customer and the developer. However, if problems arise then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

Most books and papers that describe agile methods and experiences with agile methods talk about the use of these methods for new systems development. However, as I explain in Chapter 9, a huge amount of software engineering effort goes into the maintenance and evolution of existing software systems. There are only a small number of experience reports on using agile methods for software maintenance (Poole and Huisman, 2001). There are two questions that should be considered when considering agile methods and maintenance:

1. Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
2. Can agile methods be used effectively for evolving a system in response to customer change requests?

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is often not kept up to date and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. Agile practices therefore emphasize the importance of writing well-structured code and investing effort in code improvement. Therefore, the lack of documentation should not be a problem in maintaining systems developed using an agile approach.

However, my experience of system maintenance suggests that the key document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. In this

respect, the use of agile methods is likely to make subsequent system maintenance more difficult and expensive.

Agile practices, used in the maintenance process itself, are likely to be effective, whether or not an agile approach has been used for system development. Incremental delivery, design for change and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process of software evolution.

However, the main difficulty after software delivery is likely to be keeping customers involved in the process. Although a customer may be able to justify the full-time involvement of a representative during system development, this is less likely during maintenance where changes are not continuous. Customer representatives are likely to lose interest in the system. Therefore, it is likely that alternative mechanisms, such as change proposals, discussed in Chapter 25, will be required to create the new system requirements.

The other problem that is likely to arise is maintaining continuity of the development team. Agile methods rely on team members understanding aspects of the system without having to consult documentation. If an agile development team is broken up, then this implicit knowledge is lost and it is difficult for new team members to build up the same understanding of the system and its components.

Supporters of agile methods have been evangelical in promoting their use and have tended to overlook their shortcomings. This has prompted an equally extreme response, which, in my view, exaggerates the problems with this approach (Stephens and Rosenberg, 2003). More reasoned critics such as DeMarco and Boehm (DeMarco and Boehm, 2002) highlight both the advantages and disadvantages of agile methods. They propose a hybrid approach where agile methods incorporate some techniques from plan-driven development may be the best way forward.

3.2 Plan-driven and agile development

Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity. Figure 3.2 shows the distinctions between plan-driven and agile approaches to system specification.

In a plan-driven approach, iteration occurs within activities with formal documents used to communicate between stages of the process. For example, the requirements will evolve and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together, rather than separately.

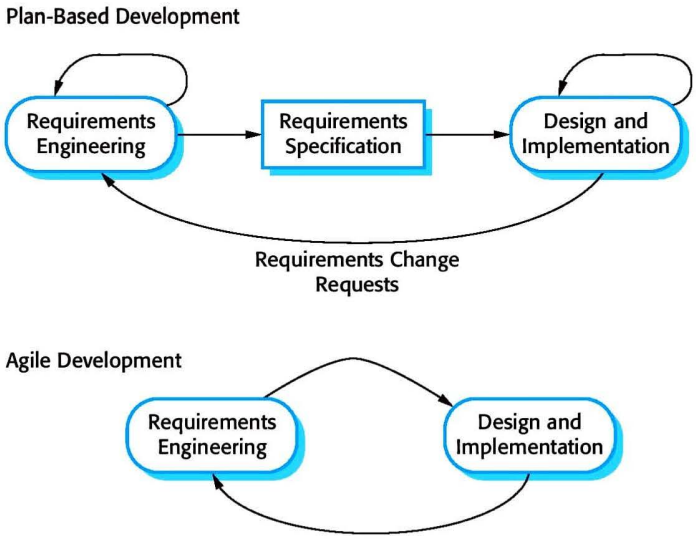


Figure 3.2 Plan-driven and agile specification

A plan-driven software process can support incremental development and delivery. It is perfectly feasible to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused and it may produce some design documentation. As I discuss in the following section, the agile development team may decide to include a documentation 'spike', where, instead of producing a new version of a system, the team produce system documentation.

In fact, most software projects include practices from plan-driven and agile approaches. To decide on the balance between a plan-based and an agile approach, you have to answer a range of technical, human, and organizational questions:

1. Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
2. Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
3. How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.
4. What type of system is being developed? Systems that require a lot of analysis before implementation (e.g., real-time system with complex timing requirements) usually need a fairly detailed design to carry out this analysis. A plan-driven approach may be best in those circumstances.
5. What is the expected system lifetime? Long-lifetime systems may require more design documentation to communicate the original intentions of the system

developers to the support team. However, supporters of agile methods rightly argue that documentation is frequently not kept up to date and it is not of much use for long-term system maintenance.

6. What technologies are available to support system development? Agile methods often rely on good tools to keep track of an evolving design. If you are developing a system using an IDE that does not have good tools for program visualization and analysis, then more design documentation may be required.
7. How is the development team organized? If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams. You may need to plan in advance what these are.
8. Are there cultural issues that may affect the system development? Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering. This usually requires extensive design documentation, rather than the informal knowledge used in agile processes.
9. How good are the designers and programmers in the development team? It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code. If you have a team with relatively low skill levels, you may need to use the best people to develop the design, with others responsible for programming.
10. Is the system subject to external regulation? If a system has to be approved by an external regulator (e.g., the Federal Aviation Authority [FAA] approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.

In reality, the issue of whether a project can be labelled as plan-driven or agile is not very important. Ultimately, the primary concern of buyers of a software system is whether or not they have an executable software system that meets their needs and does useful things for the individual user or the organization. In practice, many companies who claim to have used agile methods have adopted some agile practices and have integrated these with their plan-driven processes.

3.3 Extreme programming

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods. The name was coined by Beck (2000) because the approach was developed by pushing recognized good practice, such as iterative development, to 'extreme' levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated and tested in a day.

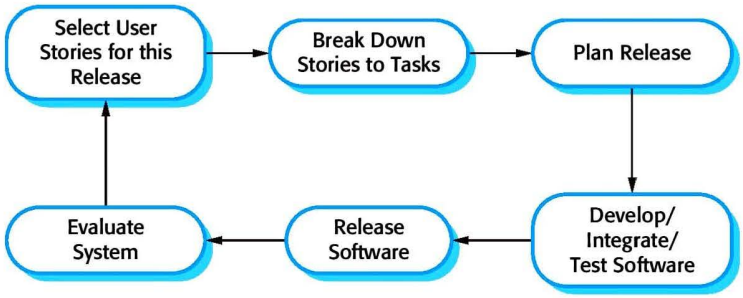


Figure 3.3 The extreme programming release cycle

In extreme programming, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system. Figure 3.3 illustrates the XP process to produce an increment of the system that is being developed.

Extreme programming involves a number of practices, summarized in Figure 3.4, which reflect the principles of agile methods:

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.
4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.
5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In an XP process, customers are intimately involved in specifying and prioritizing system requirements. The requirements are not specified as lists of required system functions. Rather, the system customer is part of the development team and discusses scenarios with other team members. Together, they develop a 'story card' that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the mental

Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Figure 3.4 Extreme programming practices

health care patient management system is shown in Figure 3.5. This is a short description of a scenario for prescribing medication for a patient.

The story cards are the main inputs to the XP planning process or the 'planning game'. Once the story cards have been developed, the development team breaks these down into tasks (Figure 3.6) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

Prescribing Medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose. If she wants to change the dose, she enters the dose and then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose and then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose and then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A 'prescribing medication' story.

Sometimes, during the planning game, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. In XP terms, this is a 'spike', an increment where no programming is done. There may also be 'spikes' to design the system architecture or to develop system documentation.

Extreme programming takes an 'extreme' approach to incremental development. New versions of the software may be built several times per day and releases are delivered to customers roughly every two weeks. Release deadlines are never slipped; if there are development problems, the customer is consulted and functionality is removed from the planned release.

When a programmer builds the system to create a new version, he or she must run all existing automated tests as well as the tests for the new functionality. The new build of the software is accepted only if all tests execute successfully. This then becomes the basis for the next iteration of the system.

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn't worth taking time to add generality to a program to cope with change. The changes anticipated often never materialize and completely different change requests may actually be made. Therefore, the XP approach accepts that changes will happen and reorganize the software when these changes actually occur.

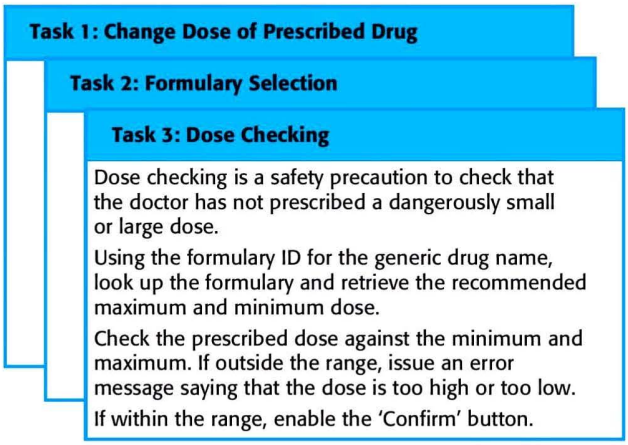


Figure 3.6 Examples of task cards for prescribing medication.

A general problem with incremental development is that it tends to degrade the software structure, so changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system.

Extreme programming tackles this problem by suggesting that the software should be constantly refactored. This means that the programming team look for possible improvements to the software and implement them immediately. When a team member sees code that can be improved, they make these improvements even in situations where there is no immediate need for them. Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of code with calls to methods defined in a program library. Program development environments, such as Eclipse (Carlson, 2005), include tools for refactoring which simplify the process of finding dependencies between code sections and making global code modifications.

In principle then, the software should always be easy to understand and change as new stories are implemented. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require the architecture of the system to be modified.

In practice, many companies that have adopted XP do not use all of the extreme programming practices listed in Figure 3.4. They pick and choose according to their local ways of working. For example, some companies find pair programming helpful; others prefer to use individual programming and reviews. To accommodate different levels of skill, some programmers don't do refactoring in parts of the system they did not develop, and conventional requirements may be used rather than user stories. However, most companies who have adopted an XP variant use small releases, test-first development, and continuous integration.

3.3.1 Testing in XP

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

To avoid some of the problems of testing and system validation, XP emphasizes the importance of program testing. XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system.

The key features of testing in XP are:

1. Test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

Test-first development is one of the most important innovations in XP. Instead of writing some code and then writing tests for that code, you write the tests before you write the code. This means that you can run the test as the code is being written and discover problems during development.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. This approach can be adopted in any process in which there is a clear relationship between a system requirement and the code implementing that requirement. In XP, you can always see this link because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation. The adoption of test-first development in XP has led to more general test-driven approaches to development (Astels, 2003). I discuss these in Chapter 8.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of 'test-lag'. This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

User requirements in XP are expressed as scenarios or stories and the user prioritizes these for development. The development team assesses each scenario and breaks it down into tasks. For example, some of the task cards developed from the story card for prescribing medication (Figure 3.5) are shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

Test 4: Dose Checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose × frequency is too high and too low.
4. Test for inputs where single dose × frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 3.7 Test case description for dose checking

The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. As I discuss in Chapter 8, acceptance testing is the process where the system is tested using customer data to check that it meets the customer's real needs.

In XP, acceptance testing, like development, is incremental. The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs. For the story in Figure 3.5, the acceptance test would involve scenarios where (a) the dose of a drug was changed, (b) a new drug was selected, and (c) the formulary was used to find a drug. In practice, a series of acceptance tests rather than a single test are normally required.

Relying on the customer to support acceptance test development is sometimes a major difficulty in the XP testing process. People adopting the customer role have very limited available time and may not be able to work full-time with the development team. The customer may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be standalone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Massol and Husted, 2003) is a widely used example of an automated testing framework.

As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually results in a large number of tests being written and executed. However, this approach does not necessarily lead to thorough program testing. There are three reasons for this:

1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.

2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
3. It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so remain untested.

Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests written after development, then undetected bugs may be delivered in the system release.

3.3.2 Pair programming

Another innovative practice that has been introduced in XP is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. However, the same pairs do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process.

The use of pair programming has a number of advantages:

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's (1971) idea of egoless programming where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (covered in Chapter 24) are very successful in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Although pair programming is a less formal process that probably doesn't find as many errors as code inspections, it is a much cheaper inspection process than formal program inspections.
3. It helps support refactoring, which is a process of software improvement. The difficulty of implementing this in a normal development environment is that effort in refactoring is expended for long-term benefit. An individual who practices refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as

two individuals working alone. There have been various studies of the productivity of paid programmers with mixed results. Using student volunteers, Williams and her collaborators (Cockburn and Williams, 2001; Williams et al., 2000) found that productivity with pair programming seems to be comparable with that of two people working independently. The reasons suggested are that pairs discuss the software before development so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers (Arisholm et al., 2007; Parrish et al., 2004) did not replicate these results. They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

3.4 Agile project management

The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project. They supervise the work of software engineers and monitor how well the software development is progressing.

The standard approach to project management is plan-driven. As I discuss in Chapter 23, managers draw up a plan for the project showing what should be delivered, when it should be delivered, and who will work on the development of the project deliverables. A plan-based approach really requires a manager to have a stable view of everything that has to be developed and the development processes. However, it does not work well with agile methods where the requirements are developed incrementally; where the software is delivered in short, rapid increments; and where changes to the requirements and the software are the norm.

Like every other professional software development process, agile development has to be managed so that the best use is made of the time and resources available to the team. This requires a different approach to project management, which is adapted to incremental development and the particular strengths of agile methods.

The Scrum approach (Schwaber, 2004; Schwaber and Beedle, 2001) is a general agile method but its focus is on managing iterative development rather than specific technical approaches to agile software engineering. Figure 3.8 is a diagram of the Scrum management process. Scrum does not prescribe the use of programming practices such as pair programming and test-first development. It can therefore be used with more technical agile approaches, such as XP, to provide a management framework for the project.

There are three phases in Scrum. The first is an outline planning phase where you establish the general objectives for the project and design the software architecture.

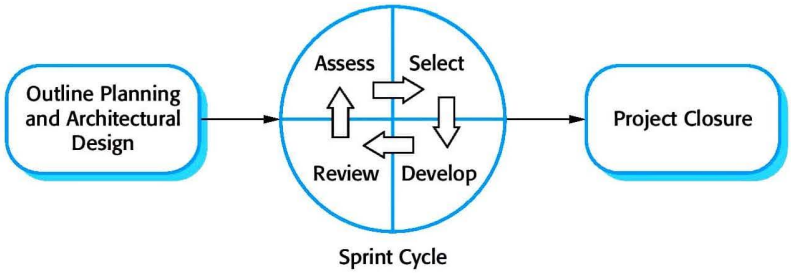


Figure 3.8 The Scrum process

This is followed by a series of sprint cycles, where each cycle develops an increment of the system. Finally, the project closure phase wraps up the project, completes required documentation such as system help frames and user manuals, and assesses the lessons learned from the project.

The innovative feature of Scrum is its central phase, namely the sprint cycles. A Scrum sprint is a planning unit in which the work to be done is assessed, features are selected for development, and the software is implemented. At the end of a sprint, the completed functionality is delivered to stakeholders. Key characteristics of this process are as follows:

1. Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
2. The starting point for planning is the product backlog, which is the list of work to be done on the project. During the assessment phase of the sprint, this is reviewed, and priorities and risks are assigned. The customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sprint.
3. The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.
4. Once these are agreed, the team organizes themselves to develop the software. Short daily meetings involving all team members are held to review progress and if necessary, reprioritize work. During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'. The role of the Scrum master is to protect the development team from external distractions. The way in which the work is done depends on the problem and the team. Unlike XP, Scrum does not make specific suggestions on how to write requirements, test-first development, etc. However, these XP practices can be used if the team thinks they are appropriate.
5. At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

The idea behind Scrum is that the whole team should be empowered to make decisions so the term 'project manager', has been deliberately avoided. Rather, the

'Scrum master' is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog, and communicates with customers and management outside of the team.

The whole team attends the daily meetings, which are sometimes 'stand-up' meetings to keep them short and focused. During the meeting, all team members share information, describe their progress since the last meeting, problems that have arisen, and what is planned for the following day. This means that everyone on the team knows what is going on and, if problems arise, can replan short-term work to cope with them. Everyone participates in this short-term planning—there is no top-down direction from the Scrum master.

There are many anecdotal reports of the successful use of Scrum available on the Web. Rising and Janoff (2000) discuss its successful use in a telecommunication software development environment, and they list its advantages as follows:

1. The product is broken down into a set of manageable and understandable chunks.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything and consequently team communication is improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works.
5. Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams with team members located in different places around the world. Consequently, there are various experiments going on to develop Scrum for distributed development environments (Smits and Pshigoda, 2007; Sutherland et al., 2007).

3.5 Scaling agile methods

Agile methods were developed for use by small programming teams who could work together in the same room and communicate informally. Agile methods have therefore been mostly used for the development of small and medium-sized systems. Of course, the need for faster delivery of software, which is more suited to customer needs, also applies to larger systems. Consequently, there has been a great deal of interest in scaling agile methods to cope with larger systems, developed by large organizations.

Denning et al. (2008) argue that the only way to avoid common software engineering problems, such as systems that don't meet customer needs and budget overruns, is to find ways of making agile methods work for large systems. Leffingwell (2007) discusses which agile practices scale to large systems development. Moore and Spens (2008) report on their experience of using an agile approach to develop a large medical system with 300 developers working in geographically distributed teams.

Large software system development is different from small system development in a number of ways:

1. Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.
2. Large systems are 'brownfield systems' (Hopkins and Jenkins, 2008); that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.
3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.
4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.
5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
6. Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

There are two perspectives on the scaling of agile methods:

1. A 'scaling up' perspective, which is concerned with using these methods for developing large software systems that cannot be developed by a small team.

2. A 'scaling out' perspective, which is concerned with how agile methods can be introduced across a large organization with many years of software development experience.

Agile methods have to be adapted to cope with large systems engineering. Leffingwell (2007) argues that it is essential to maintain the fundamentals of agile methods—flexible planning, frequent system releases, continuous integration, test-driven development, and good team communications. I believe that the critical adaptations that have to be introduced are as follows:

1. For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation. The software architecture has to be designed and there has to be documentation produced to describe critical aspects of the system, such as database schemas, the work breakdown across teams, etc.
2. Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress. A range of communication channels such as e-mail, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
3. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible when several separate programs have to be integrated to create the system. However, it is essential to maintain frequent system builds and regular releases of the system. This may mean that new configuration management tools that support multi-team software development have to be introduced.

Small software companies that develop software products have been amongst the most enthusiastic adopters of agile methods. These companies are not constrained by organizational bureaucracies or process standards and they can change quickly to adopt new ideas. Of course, larger companies have also experimented with agile methods in specific projects, but it is much more difficult for them to 'scale out' these methods across the organization. Lindvall, et al. (2004) discuss some of the problems in scaling-out agile methods in four large technology companies.

It is difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software

tools (e.g., requirements management tools) and the use of these tools is mandated for all projects.

3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Change management and testing procedures are examples of company procedures that may not be compatible with agile methods. Change management is the process of controlling changes to a system, so that the impact of changes is predictable and costs are controlled. All changes have to be approved in advance before they are made and this conflicts with the notion of refactoring. In XP, any developer can improve any code without getting external approval. For large systems, there are also testing standards where a system build is handed over to an external testing team. This may conflict with the test-first and test-often approaches used in XP.

Introducing and sustaining the use of agile methods across a large organization is a process of cultural change. Cultural change takes a long time to implement and often requires a change of management before it can be accomplished. Companies wishing to use agile methods need evangelists to promote change. They must devote significant resources to the change process. At the time of writing, few large companies have made a successful transition to agile development across the organization.

KEY POINTS

- Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads, and producing high-quality code. They involve the customer directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system.
- Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement, and customer participation in the development team.
- A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.

3.5.1 Adaptive Software Development (ASD)

WebRef

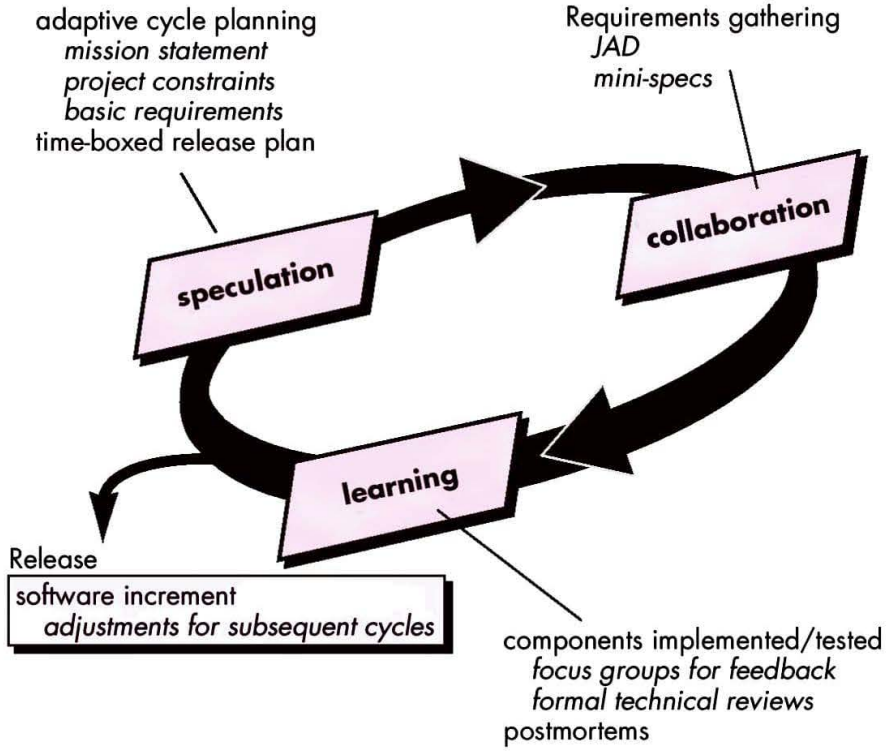
Useful resources for ASD can be found at **www.adaptivesd.com**.

Adaptive Software Development (ASD) has been proposed by Jim Highsmith [Hig00] as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

Highsmith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" (Figure 3.3) that incorporates three phases, speculation, collaboration, and learning.

FIGURE 3.3

Adaptive software development



12 See http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

During *speculation,* the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

Effective collaboration with your customer will only occur if you jettison any "us and them" attitudes.

No matter how complete and farsighted the cycle plan, it will invariably change. Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working.

Motivated people use *collaboration* in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

ASD emphasizes learning as a key element in achieving a "self-organizing" team.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on "learning" as much as it is on progress toward a completed cycle. In fact, Highsmith [Hig00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups (Chapter 5), technical reviews (Chapter 14), and project postmortems.

The ASD philosophy has merit regardless of the process model that is used. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

3.5.3 Dynamic Systems Development Method (DSDM)

WebRef

Useful resources for DSSD can be found at **www.dsdm.org**.

The *Dynamic Systems Development Method* (DSDM) [Sta97] is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" [CCS02]. The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The DSDM Consortium (**www.dsdm.org**) is a worldwide group of member companies that collectively take on the role of "keeper" of the method. The consortium has defined an agile process model, called the *DSDM life cycle* that defines three different iterative cycles, preceded by two additional life cycle activities:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

DSDM is a process framework that can adopt the tactics of another agile approach such as XP.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer. (Note: All DSDM prototypes are intended to evolve into the deliverable application.) The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration—revisits prototypes built during *functional model iteration* to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.

Implementation—places the latest software increment (an "operationalized" prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

DSDM can be combined with XP (Section 3.4) to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments. In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

3.5.5 Feature Driven Development (FDD)

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues [Coa99] as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing [Pal02] have extended and improved Coad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

WebRef

A wide variety of articles and presentations on FDD can be found at: **www.featuredrivendevelopment.com/**.

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means. FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits (Chapter 16), the collection of metrics, and the use of patterns (for analysis, design, and construction).

In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" [Coa99]. The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues [Coa99] suggest the following template for defining a feature:

<action> the **<result>** **<by | for | of | to>** a(n) **<object>**

where an **<object>** is "a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)." Examples of features for an e-commerce application might be:

Add the product to shopping cart

Display the technical-specifications of the product

Store the shipping-information for the customer

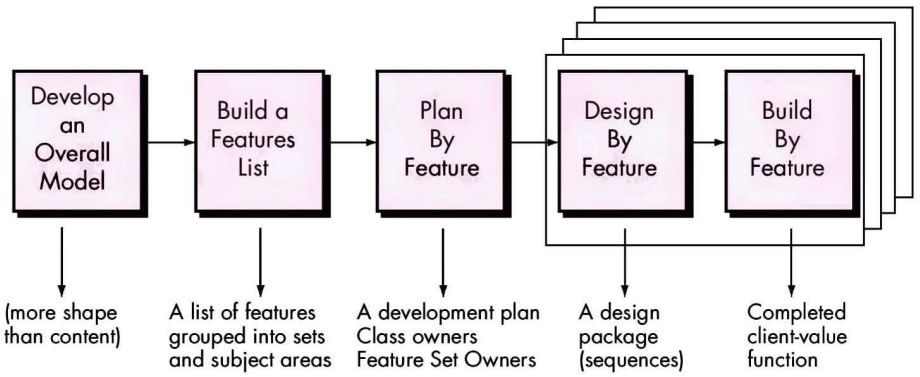


FIGURE 3.5
Feature Driven Development [Coa99] (with permission)

A feature set groups related features into business-related categories and is defined [Coa99] as:

<action><-ing> a(n) **<object>**

For example: *Making a product sale* is a feature set that would encompass the features noted earlier and others.

The FDD approach defines five "collaborating" [Coa99] framework activities (in FDD these are called "processes") as shown in Figure 3.5.

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and other stakeholders to understand project status—what accomplishments have been made and problems have been encountered. If deadline pressure is significant, it is critical to determine if software increments (features) are properly scheduled. To accomplish this, FDD defines six milestones during the design and implementation of a feature: "design walkthrough, design, design inspection, code, code inspection, promote to build" [Coa99].

3.5.6 Lean Software Development (LSD)

Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized ([Pop03], [Pop06a]) as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,* and *optimize the whole.*

Each of these principles can be adapted to the software process. For example, *eliminate waste* within the context of an agile software project can be interpreted to mean [Das05]: (1) adding no extraneous features or functions, (2) assessing the cost and schedule impact of any newly requested requirement, (3) removing any superfluous process steps, (4) establishing mechanisms to improve the way team members find information, (5) ensuring the testing finds as many errors as possible,

(6) reducing the time required to request and get a decision that affects the software or the process that is applied to create it, and (7) streamlining the manner in which information is transmitted to all stakeholders involved in the process.

For a detailed discussion of LSD and pragmatic guidelines for implementing the process, you should examine [Pop06a] and [Pop06b].