CHAPTER

5

Software Design

Learning Outcomes

In this chapter you will learn:

- Basic terminology
- Recurring concepts in software design
- Producing highly-cohesive and lightly-coupled modules
- Assessing and enhancing intramodule cohesiveness and intermodule coupling
- Transforming specifications into design using a top-down refinement process
- Documenting a software design
- Reviewing and validating a design
- Designing usable and user-friendly graphical user interfaces

Design follows analysis

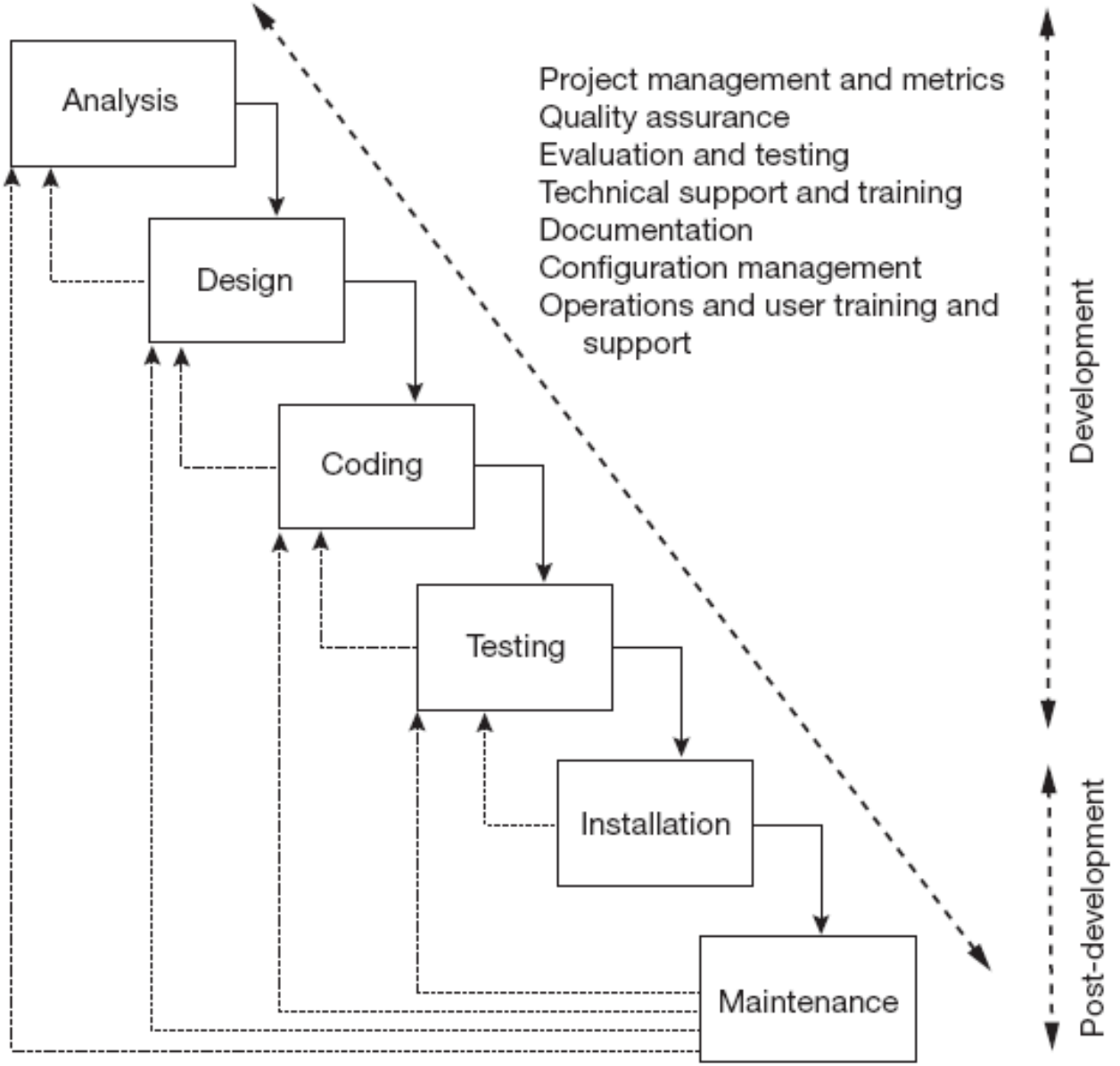


Figure 2.1 Waterfall model phases and its ongoing umbrella activities

Mapping specification to design

- How the specification can be implemented?
- Deliverables:
  - High level / architectural design
  - Detailed design, Database design, GUI design
  - Test plans

Table 5.1 Mapping of specification outcomes to design outcomes

Specification outcomes	Design outcomes
Data model	Database design
Process model and use case model	High-level design/object-oriented design Graphical user interface design
Behavioral model	Detailed design
Non-functional requirements	High-level and detailed design

Recurring concepts in design

- Abstraction
- Top down decomposition
- Information hiding
- Separation of concerns
- Interfaces
- Modularity
- Divide and conquer

Abstraction

- Hiding details for the purpose of simplifying and managing the design of complex software.
- A software module, as a black box, is an abstraction of the control and data flows that exist inside it.
- Specification is an abstraction of the design, and the design as an abstraction of the implementation.
  - Reducing the level of abstraction means moving towards a more concrete realization of the current level.

Top down decomposition

- Also called **stepwise refinement** is a software analysis and design procedure by which a process or an abstraction at a high level is broken into two or more lower level abstractions.
  - The decomposition procedure continues until a process or an abstraction is reached and cannot be decomposed any further.
- Top down decomposition was used to refine the context level DFDs until a low level diagram is obtained in which all processes in the diagram are not decomposable.

Information hiding

- Also called **encapsulation** is related to the hiding of internal design decisions related to the selected algorithm and data structures of a module from the outside world.
- Reduce the side effects of any future maintenance or modification of the design, and hence minimizing the effect on the other modules in the design.
- Modularity, information hiding and encapsulation help achieving the concept of **separation of concern**.

Separation of concerns

- Related to the partitioning of complex software into separate modules that are very lightly related or coupled.
- Modules can then be designed by different designers with very little interfacing among them.

Interfaces

- Interfaces are the points of accesses through which modules or systems communicate.
- Each abstraction should possess a well defined interface clearly describing the expected inputs to and outputs from the abstraction.

Modularity

- Modularity in design refers to the splitting of a large software system into smaller interconnected modules.
- Modules are interconnected through their interfaces. The interconnection should be as simple and little as possible to avoid side effects and costly maintenance.
- Two modules are **directly connected** if one module can call the other module.
- Two modules are **indirectly connected** if they share common files or global data structures.

Divide and conquer

- A concept used in developing a detailed design or an algorithm for a module.
- Based on recursively breaking the problem into smaller sub problems. Backtracking will occur when the lowest sub problems reached are solved hence contributing to the solution of the original problem.

Desirable design properties

- Adaptability and extendibility
- Fault-tolerance and error recoverability
- Maintainability
- Portability
- Reliability
- Reusability
- Scalability
- Security
- Testability, diagnosability
- Trustworthiness (fault-tolerant, reliable and secure)

‘contributes’ relationships

among design properties

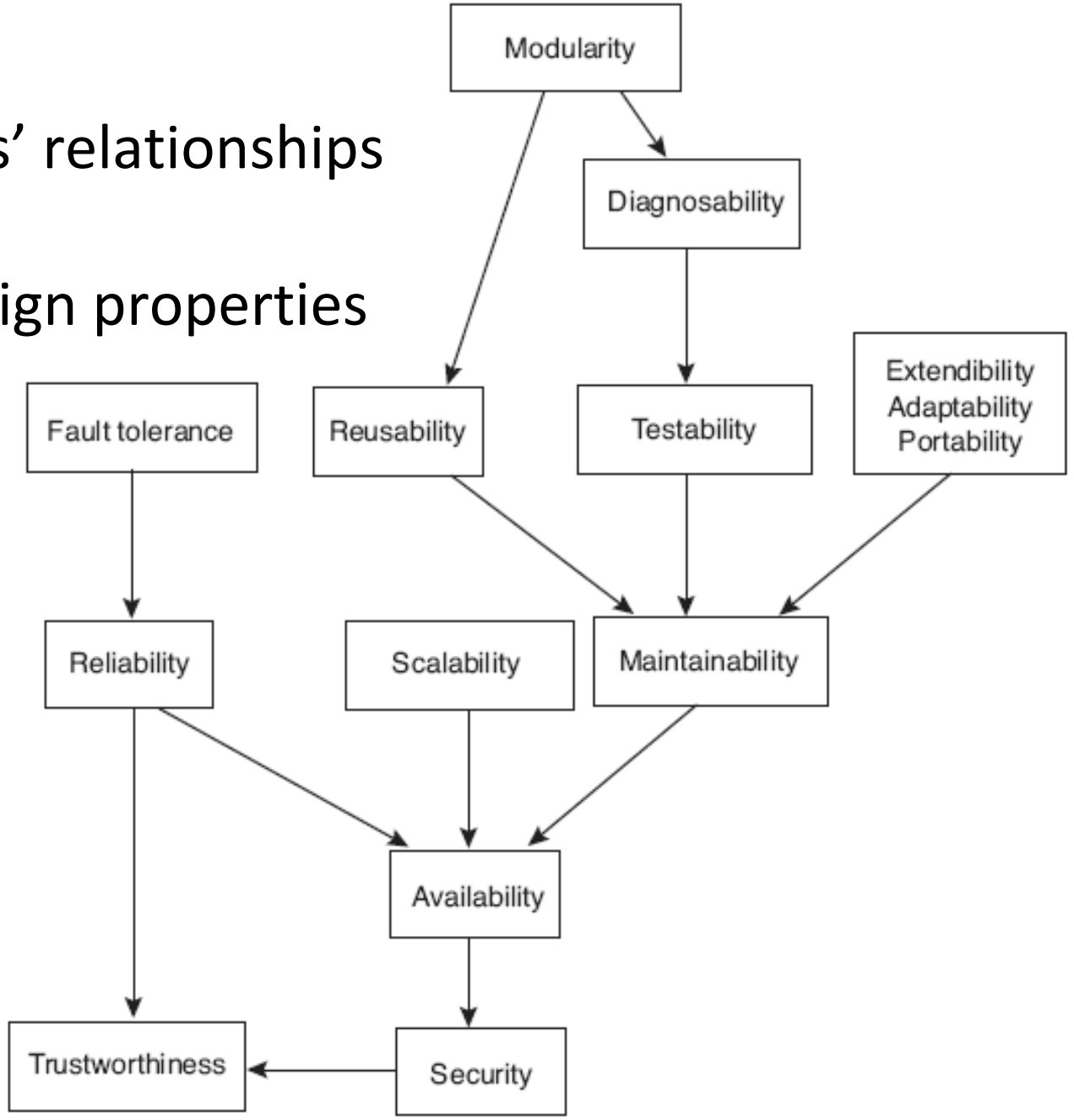


Figure 5.1 The *contributes* relationships among design properties

High level architectural design

- Very important for meeting the non-functional requirements and enforcing good design properties
- Interconnected modules
  - Fan in, fan out, height, width
  - Inter-module coupling
  - Intra-module cohesion

Design hierarchy

Fan out of M3 is 4

Fan in of M is 3

Height is 4, width is 5

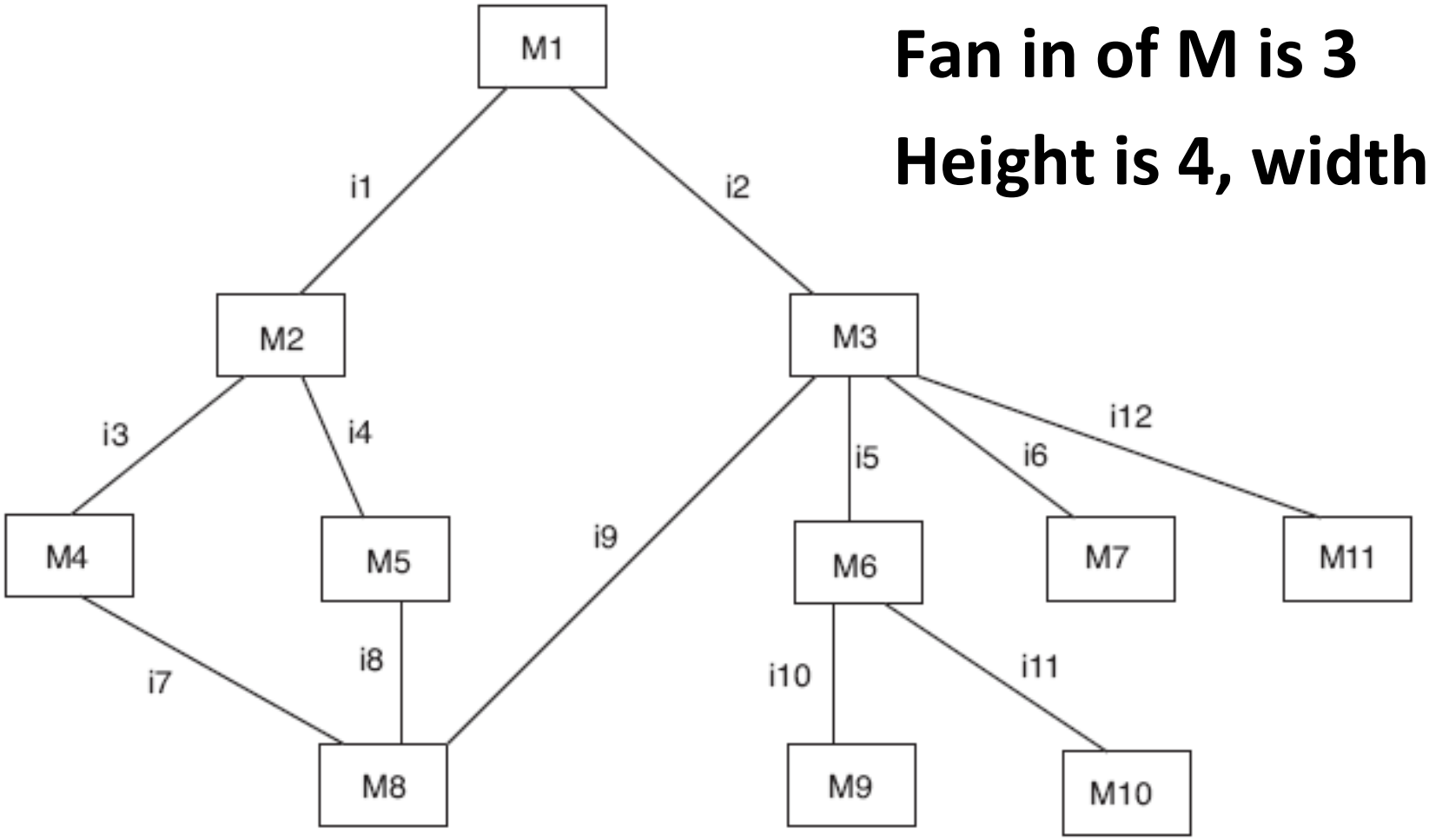


Figure 5.2 A high-level design hierarchy

Intermodule coupling

- Measure of relationship between two modules M1 and M2
  - Direct relationship: M1 is calling M2
  - Indirect relationship: M1 and M2 share common data or resources
- No coupling: no direct or indirect relationships between M1 and M2

Intermodule coupling

- **Data coupling**: M1 is passing the necessary data to M2
- **Stamp coupling**: M1 is passing a data structure to M2 - not all of its content is needed
- **Control coupling**: M1 is passing a flag to M2 – flag controls the execution inside M2
- **Common coupling**: M1 and M2 have indirect relationship
- **Content coupling**: M1 jumps inside M2 (go to in some programming languages)

Table 5.2 Summary of the levels of intermodule coupling

Level	Coupling	Brief description
0	No coupling	Module is called but does not share any data directly or indirectly with any other module
1	Data	Module is passing only the necessary data to another module
2	Stamp	Module is passing more data than needed to another module
3	Control	Module is passing a flag to control the flow of control in another module
4	Common	Modules are sharing common global data and data files
5	Content	Module branching out to a label in the other module

0 = Best 5 = Worst

Example

```
M1(boolean flag) { if (flag==true)call
      M6(studentRec); }
M2(int studentNum) { read studentRec from
      file A; call M1(studentRec.status); call
      M3(studentRec.studentName); }
M3(string stringToPrint) { print stringToPrint; }
M4(Record studentRec) { modify studentRec;
      update File A; call M3("record updated"); }
M5() { call M2(studentNum); call
      M4(studentRec); }
M6(studentRec) { sendEmailTo studentRec.
      studentName; }
```

Figure 5.3 Design hierarchy corresponding to the pseudocode in Example 5.2

```
M1(boolean flag) { if (flag==true)call
    M6(studentRec); }
M2(int studentNum) { read studentRec from
    file A; call M1(studentRec.status); call
    M3(studentRec.studentName); }
M3(string stringToPrint) { print stringToPrint; }
M4(Record studentRec) { modify studentRec;
    update File A; call M3("record updated"); }
M5() { call M2(studentNum); call
    M4(studentRec); }
M6(studentRec) { sendEmailTo studentRec.
    studentName; }
```

Table 5.3 Coupling among modules of the design in Figure 5.3

	M1	M2	M3	M4	M5	M6
M1						Stamp
M2	Control		Data	Common		
M3						
M4		Common	Data			
M5		Data		Data		
M6						

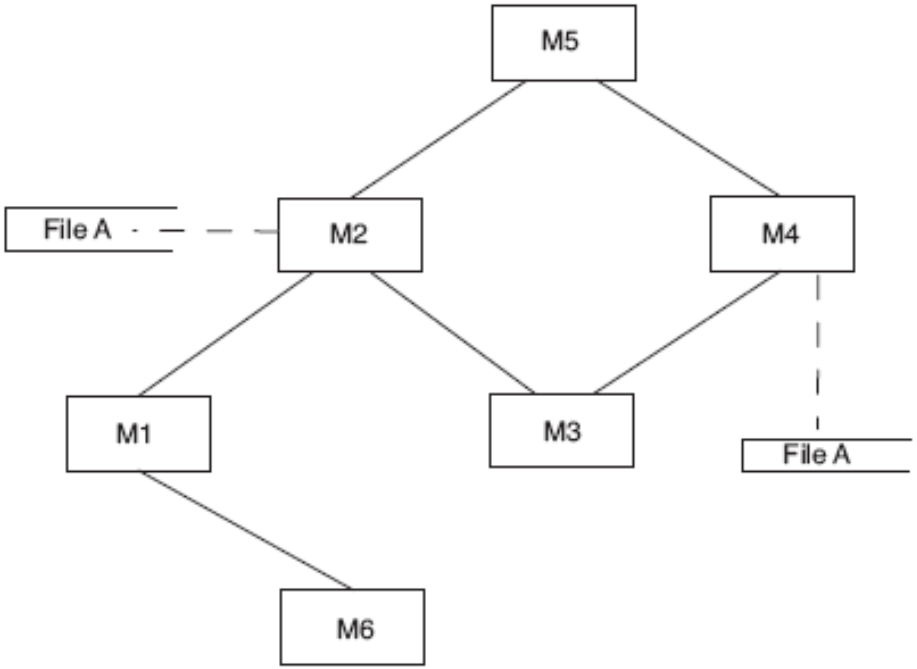


Figure 5.3 Design hierarchy corresponding to the pseudocode in Example 5.2

Figure 5.3 Design hierarchy corresponding to the pseudocode in Example 5.2

Table 5.3 Coupling among modules of the design in Figure 5.3

	M1	**M2**	**M3**	**M4**	**M5**	**M6**
M1						Stamp
M2	Control		Data	Common		
M3						
M4		Common	Data			
M5		Data		Data		
M6						

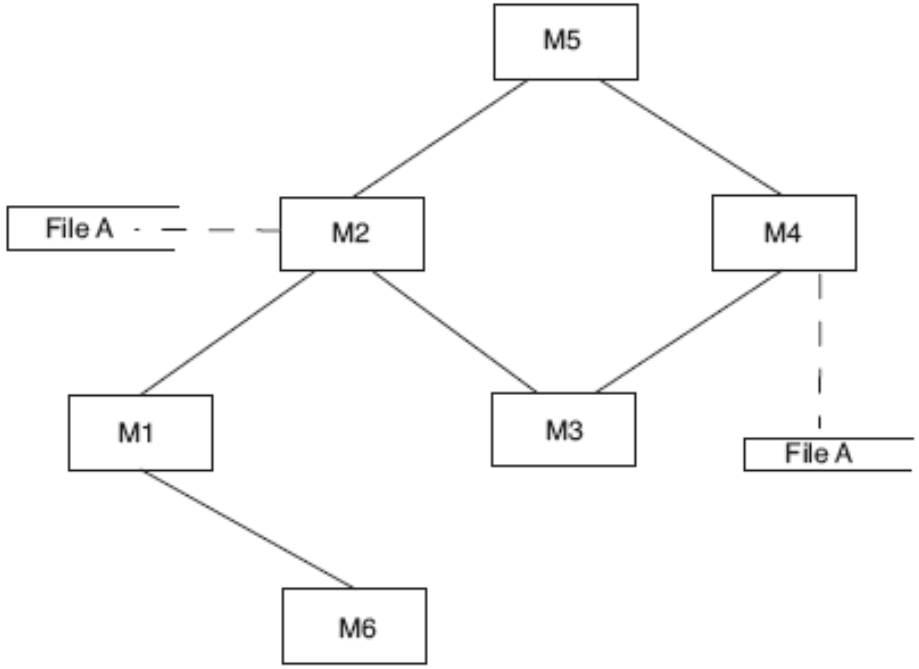


Figure 5.3 Design hierarchy corresponding to the pseudocode in Example 5.2

Structured design

- Transforming DFD into a design hierarchy
- DFD process into a module

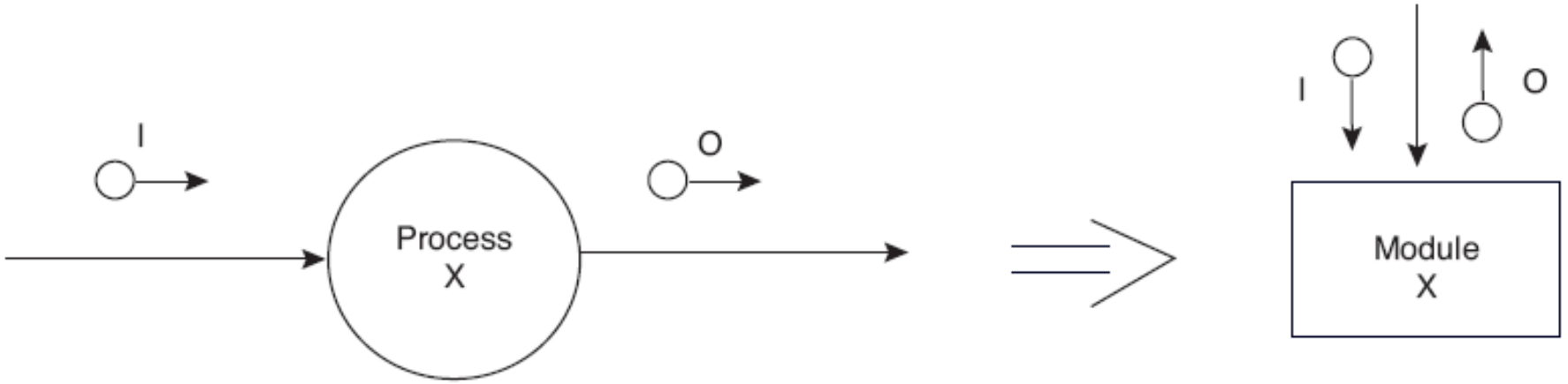


Figure 5.4 Transforming a process in a DFD into a module

Example

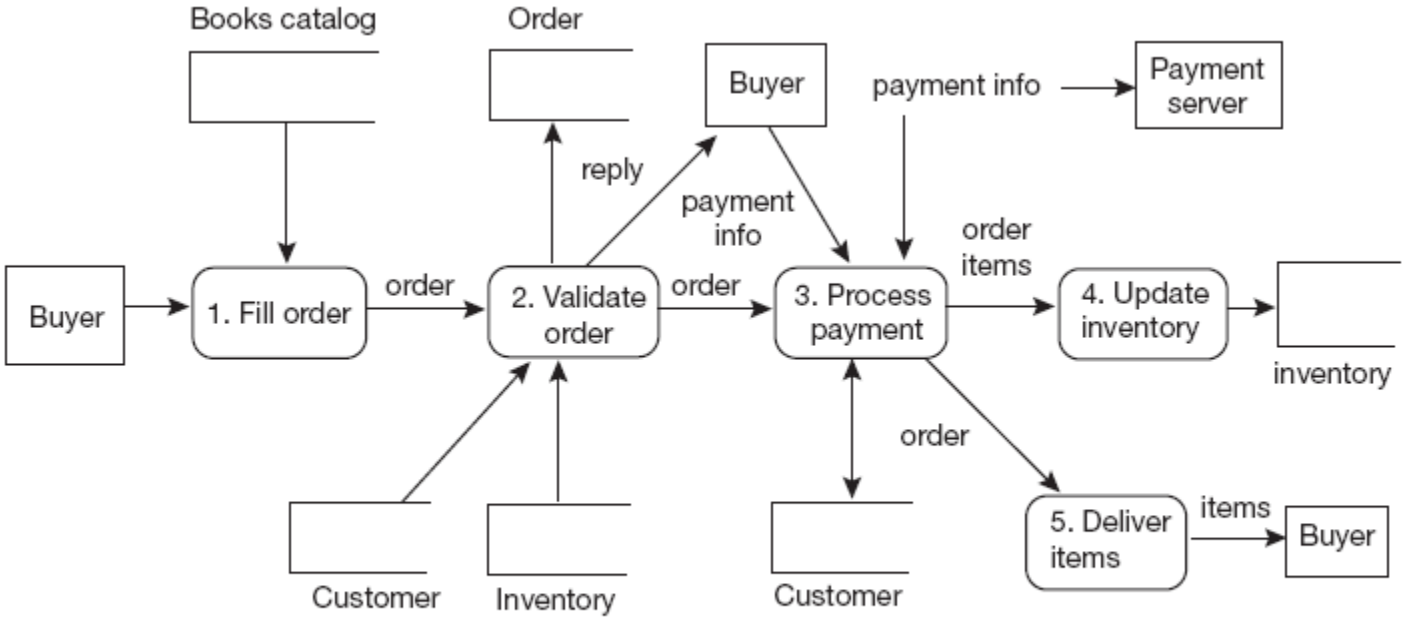


Figure 4.21 Level 0 DFD for the context level DFD in Figure 4.20

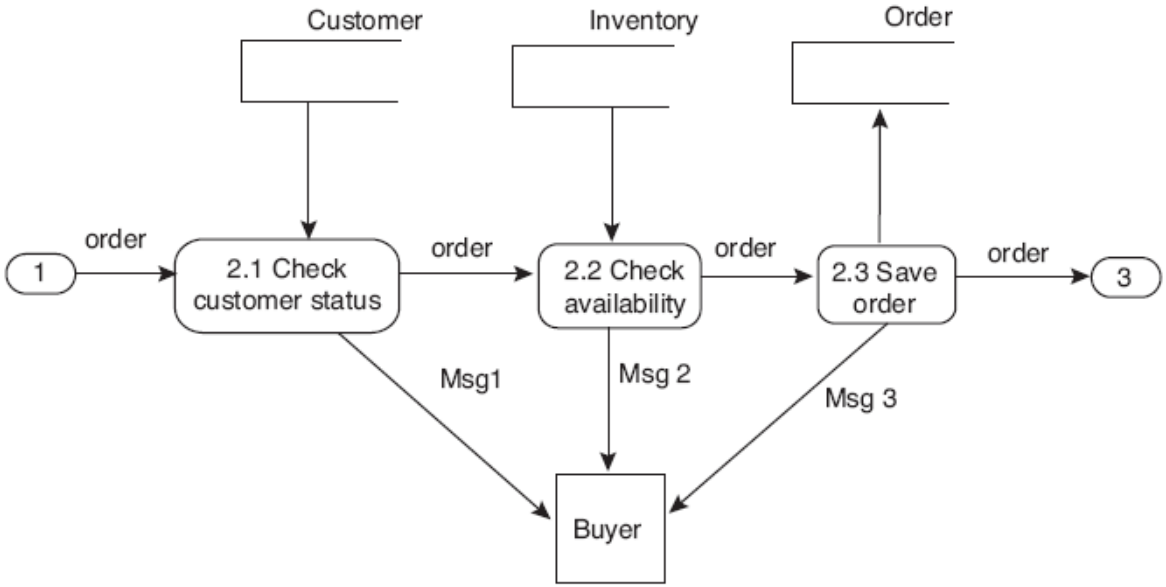


Figure 4.22 Decomposition of process validate order

Example

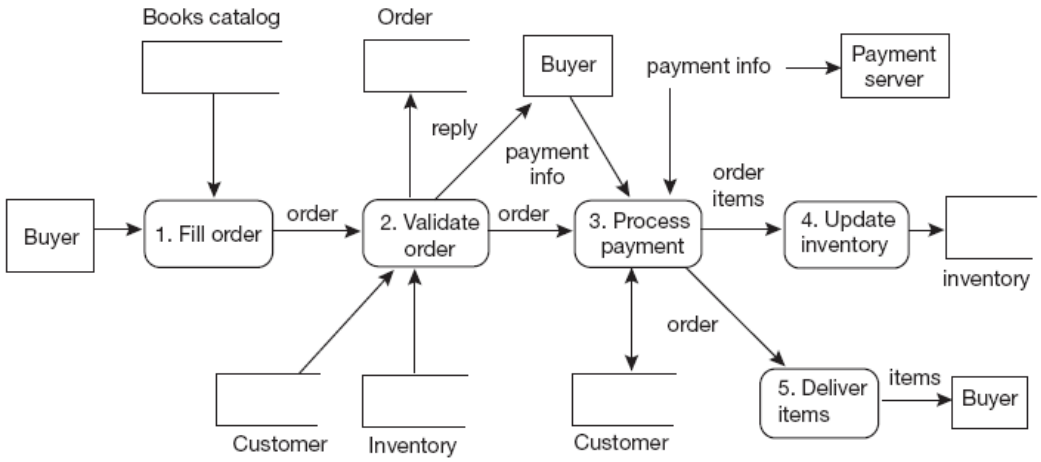


Figure 4.21 Level 0 DFD for the context level DFD in Figure 4.20

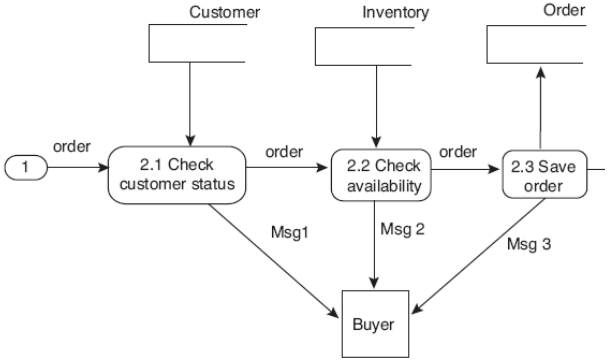


Figure 4.22 Decomposition of process validate order

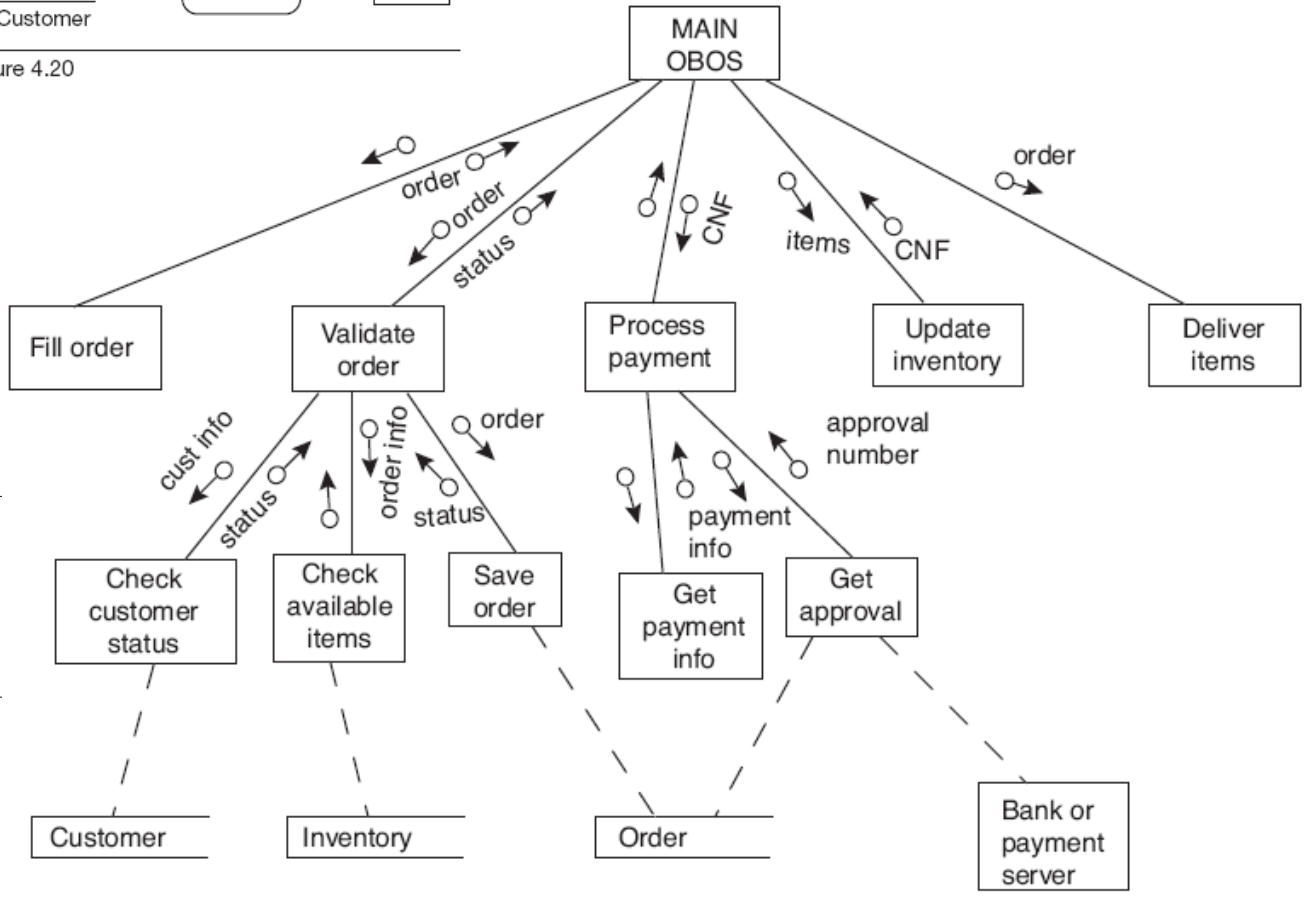


Figure 5.5 High-level design hierarchy corresponding to the DFDs in Example 4.10

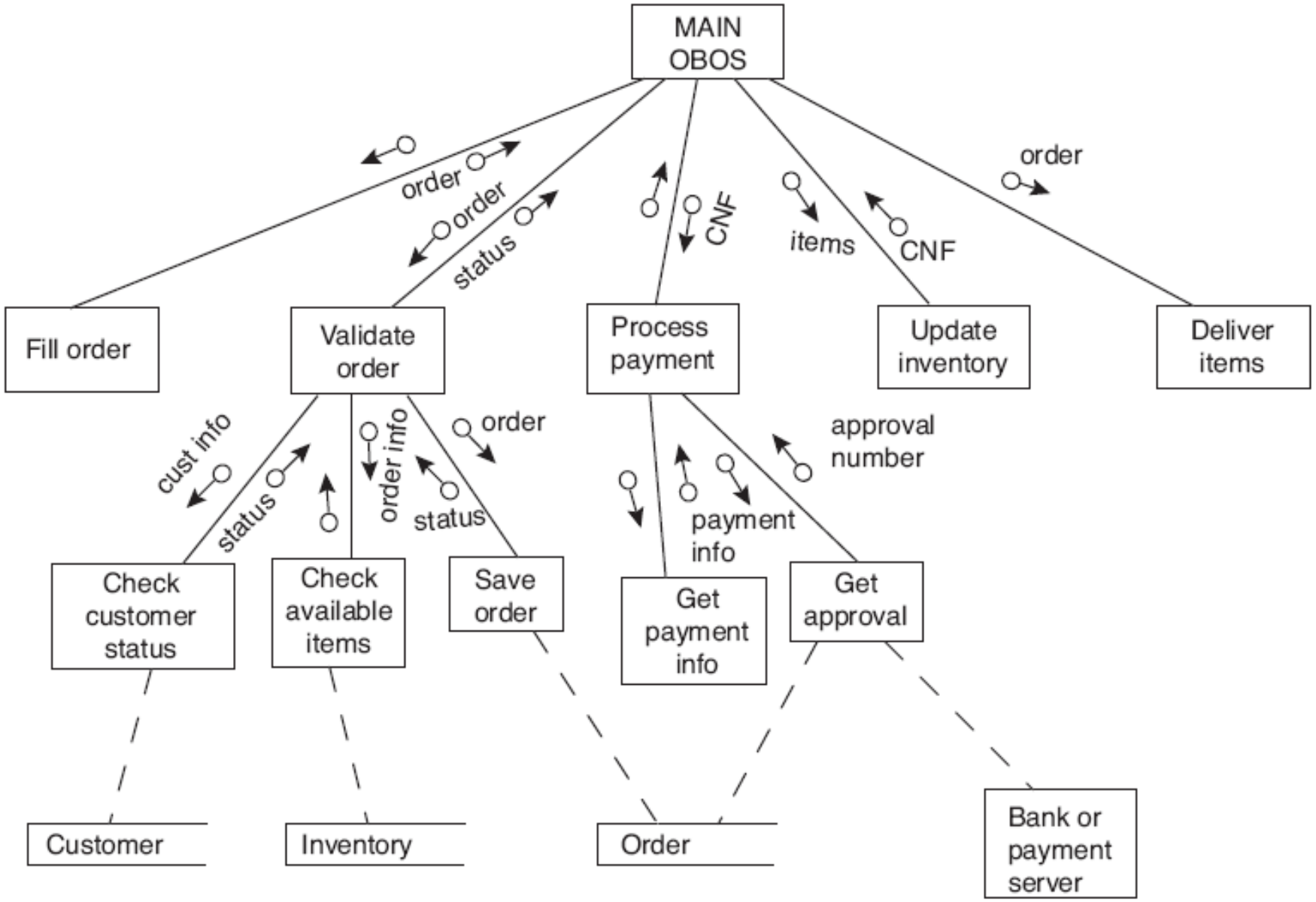


Figure 5.5 High-level design hierarchy corresponding to the DFDs in Example 4.10

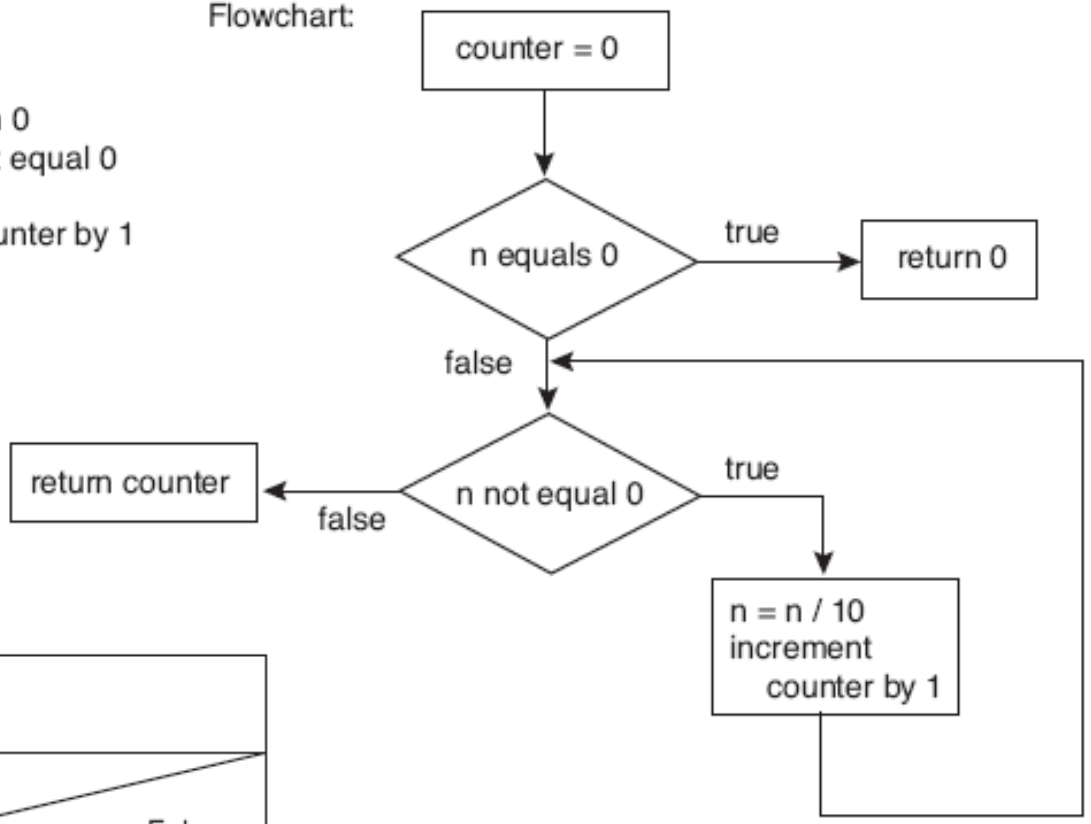
Detailed design

- Describing the module data (structures) and algorithm
- Pseudocode, flowchart, activity diagram, Nassi-Schneidermann, …
- Template for module description

Pseudocode:

```
int counter = 0
if (n equals 0) return 0
repeat while n is not equal 0
        n = n / 10
        increment counter by 1
return counter
```

Flowchart:



Nassi-Shneiderman Diagram:

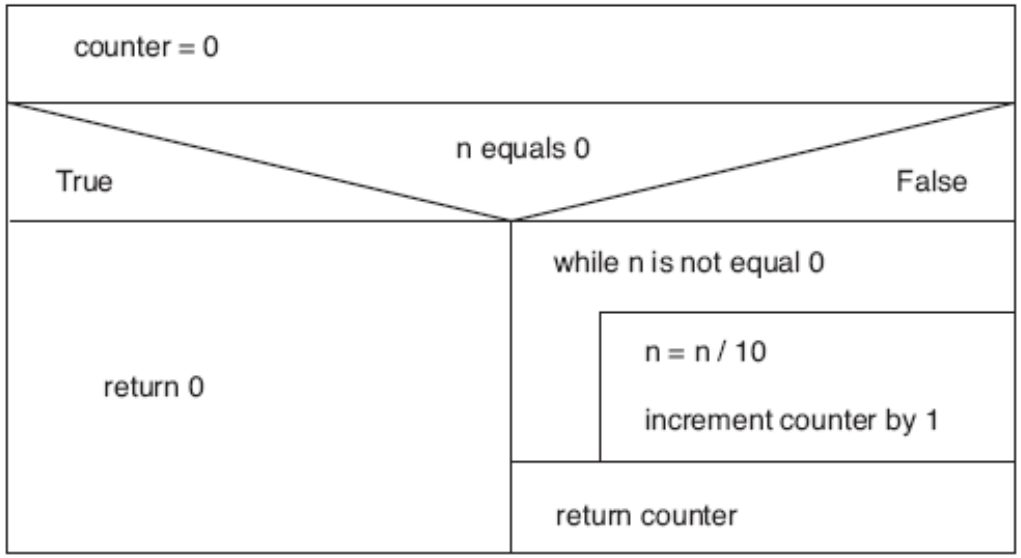


Figure 5.6 Alternative ways for describing an algorithm for a detailed design

Module detailed design template

Table 5.4 Module-detailed design template

Module Name:	
Author:	Created:
Revision history:	
Module description:	
Module interfaces:	
Module returned value(s):	
Called by module(s):	
Calls module(s):	
File(s) accessed:	
External system(s) accessed:	
Data structures and global data used:	
Algorithm in pseudocode:	

Table 5.5 Detailed design for module ValidateOrder

Module Name: ValidateOrder
Author: Kassem Saleh Created: January 12, 2006
Revision history: Revised by same on March 12, 2006
Module description: This module receives order information and checks whether the order can be filled. The returned value indicates the type of problem with the order, if any.
Module interfaces: Input: Order record
Module returned value(s): 0—Valid order 1—Bad customer standing 2—Item(s) not available 3—Cannot save order
Called by module(s): MAIN
Calls module(s): CheckCustomerStatus( ), CheckAvailableItems( ), and SaveOrder()
File(s) accessed: None
External system(s) accessed: None
Data structures and global data used: Order data structure filled by user

Algorithm in pseudocode:
code = CheckCustomerStatus(customerInfo);
if code != 0 return code
code = CheckAvailableItems(orderItems);
if code != 0 return code
return SaveOrder(order);

Table 5.6 Detailed design for module CheckCustomerStatus

Module Name: CheckCustomerStatus
Author: Kassem Saleh Created: January 13, 2006
Revision history: none
Module description: This module receives customer information and checks in the customer file to establish if the customer is in good standing
Module interfaces: Input: Customer information (part of the order information)
Module returned value(s): 0—Good standing 1—Bad standing
Called by module(s): ValidateOrder
Calls module(s): None
File(s) accessed: Customer file
External system(s) accessed: None
Data structures and global data used: customerInfo structure
Algorithm in pseudocode: Open file Customer for read only Query file for customerInfo.customerNumber If customerNumber does not exist return 0; —no record for new customer return foundCustomerRecord.status;

Intramodule cohesion

- Level of closeness or cohesion of steps in module's algorithm
- Higher cohesion leads to better software: higher maintainability, testability, reusability …
- Functional cohesiveness is the highest
- Coincidental cohesiveness is the lowest

Table 5.7 Summary of the levels of intramodule cohesiveness

Level	Cohesion	Brief description
0	Coincidental	Module performs unrelated activities that are included in the module by coincidence
1	Logical	Module performs one activity exclusively among many similar activities based on a control variable passed to the module
2	Temporal	Module includes unrelated steps that must be performed at the same time
3	Procedural	Module performs an unrelated set of activities in a specific order according to a business process or procedure
4	Communicational	Module performs many unrelated activities sharing the same input and output
5	Sequential	Module contains a sequence of activities—the output of one activity is the input needed to perform the successor activity in the sequence
6	Functional	Module contains strongly-related steps or statements that perform a single function

0 = Worst 6 = Best

Database design

- Mapping ER model into Database schema (or table) design
- 1 to 1: 1 table, 1 to m: 2 tables, m to m: 3 tables

Example

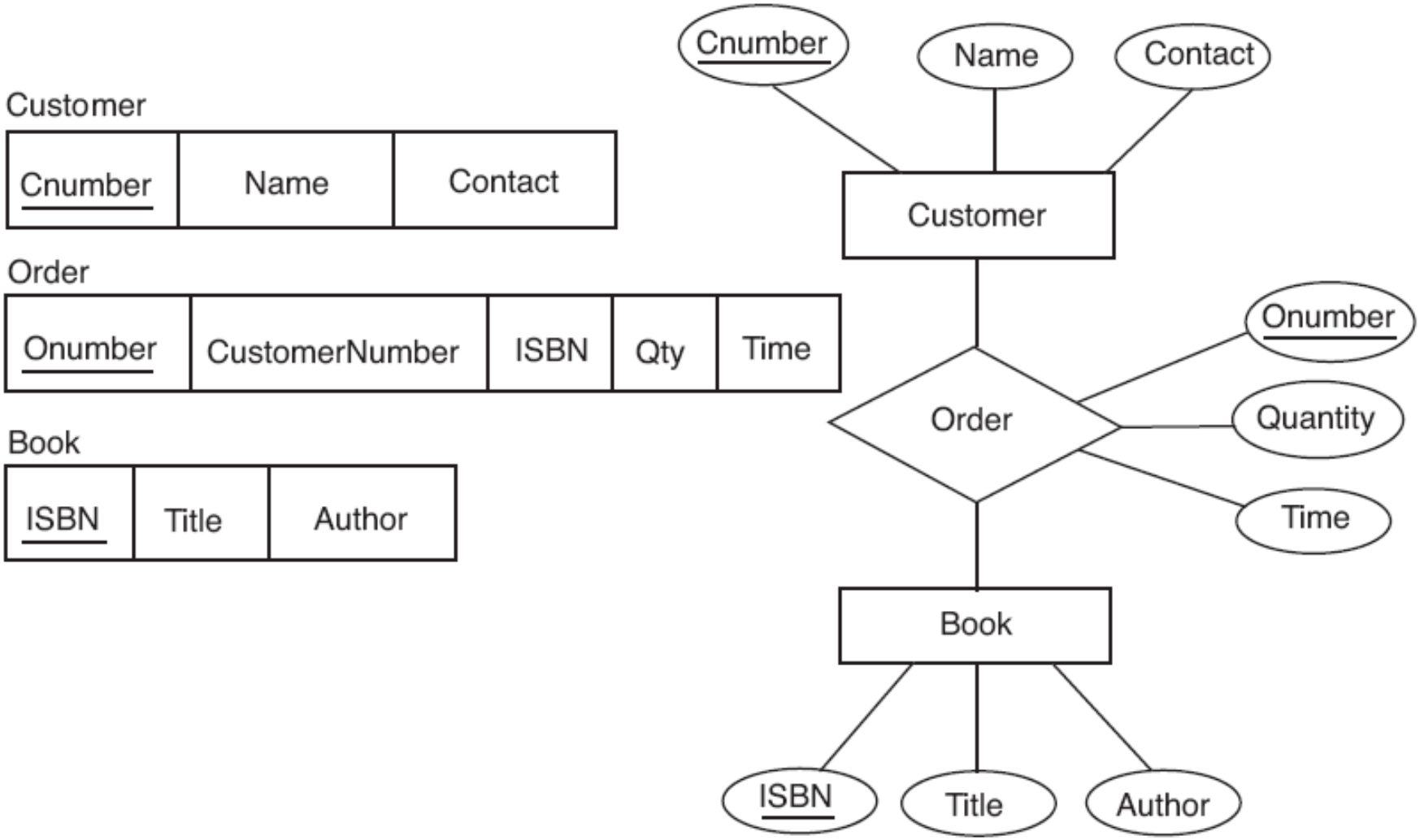


Figure 5.7 Schema design for Customer, Order, and Book relations

Graphical User Interface Design

- Details on the external interface specifications described in the SRS document
- Look of the interface artifacts, their sequencing and interrelationships
- GUI development team
  - Specification, design, implementation, testing
- Guidelines for designing GUI

Guidelines for good GUI

- Consistency
- Reusability
- Flexibility and efficiency
- Forgiveness and tolerance
- Readability, simplicity and clarity
- User friendliness
- Visibility

Consistency

- Consistency of the interface within the same application, and across similar applications.
  - the use of similar font sizes and styles in different screens, dialogs and forms,
  - the use of the same function keys and control keys to perform the same actions on different screens of the design like using F6 to get contextual help in different screens,
  - the use of the same format to display error messages, and
  - the use of the same icons to mean the same thing like using the error, warning, information icons.
- Similar things must be dealt with similarly
  - opening any file by double clicking on the file icon, regardless of the file type.

Reusability

- GUI designer should aim at reusing GUI artifacts and elements from existing and proven graphical interfaces.
- Reuse will allow consistency of the GUI across various applications, hence enhancing the usability and the user familiarity with the new interface, and hence reducing the need for future changes to the interface.
  - The menu bar in existing editing applications includes menus and menu items in an order the users are familiar with.
- GUI design should reuse the same GUI layout unless there is a proven benefit to deviate from familiar interface design

Flexibility and efficiency

- The interface should be flexible in allowing novice and expert users to use different paths to navigate the interface.
  - Expert and frequent users can use shortcuts to improve the efficiency of the navigation. Default settings can be used by novice users and should be easily modified if the user wishes to do so.
- The interface must be easily navigated by users.

Forgiveness and tolerance

- The GUI design should allow the user to confirm or undo critical actions.
- The interface should correct trivial user inputs errors and continue operating normally thereafter.
- Forgiveness and tolerance contribute to the user friendliness of the system.

Readability, simplicity and clarity

- The GUI design must include simple GUI artifacts
  - simple screens with clear messages without overcrowding them, use of appropriate colors, font sizes and styles that is convenient to the target users of the software.
- A usability questionnaire to understand the type of audience and their preferences.
  - if 90% of the users are novice users and ages are above 50, the interaction style must be mostly based on pointing devices, and font size used in text must be selected accordingly.
  - Displayed error, help and warning messages must be clear, concise and as simple as possible to assist the user in properly choosing the next action to be performed.

User friendliness

- GUI design must provide helpful, courteous and non-offending messages.
- For example, an error message displayed by the system should spell out exactly what is the error and how it can be avoided by the user.
- Displaying a system error number without any additional information is considered to be offending the user intellect and hence unfriendly.
- Help messages must be concise and contextual by providing only the needed and requested help.
- Adapting the user interface to different languages and cultures is very crucial in today's global context.
- Allowing the user to change the interface language could be an option to deal with this issue.

Visibility

- GUI should not include any hidden artifacts.
- UI components that should be available for particular user types should be disabled.
- The existence of non visible interface components may be considered harmful in terms of security and must be dealt with carefully during code reviews.
- GUI usability metrics included in the GUI design. These decisions do not involve the addition of hidden GUI components, but are reflected later in additional GUI related code including counter and probes to collect statistics.

Software design document

- IEEE 1016 – documenting software design
- Complex
  - The decomposition view shows the partitioning of the system into design entities. This view can be expressed using hierarchical diagram of components and their composite sub components, like a class diagram.
  - The dependency view shows the relationships among design entities. This view can be described using structured charts and data dependency diagrams.
  - The interface view includes the description of all design entity interfaces allowing their proper use. This view can be described by interface files and parameter lists.
  - The detailed view includes the internal detailed design of a design entity. This view can be described using flowcharts, pseudocode, Nassi-Shneiderman diagrams or detailed schema description.

Table 5.8 An alternative design document template

1. Introduction
   Purpose, scope, assumptions, and references
2. High-level design
   Transforming the data flow diagram into a structured chart or object-oriented design
   Detailed description of each module or method interface
3. Database design
   Mapping the entity relationship model into a database schema design
   Detailed description of the relational schema structure
4. Graphical user interface design
   Screen design and ordering of screen sequences
5. Detailed design
   Detailed description of each module or method listed in the high-level design section using a unified template

Design reviews

- IEEE standard 1028 on software reviews
  - Management review
  - Technical review
  - Inspection and walkthrough
  - Audit and quality assurance
- Last three types of reviews are applicable

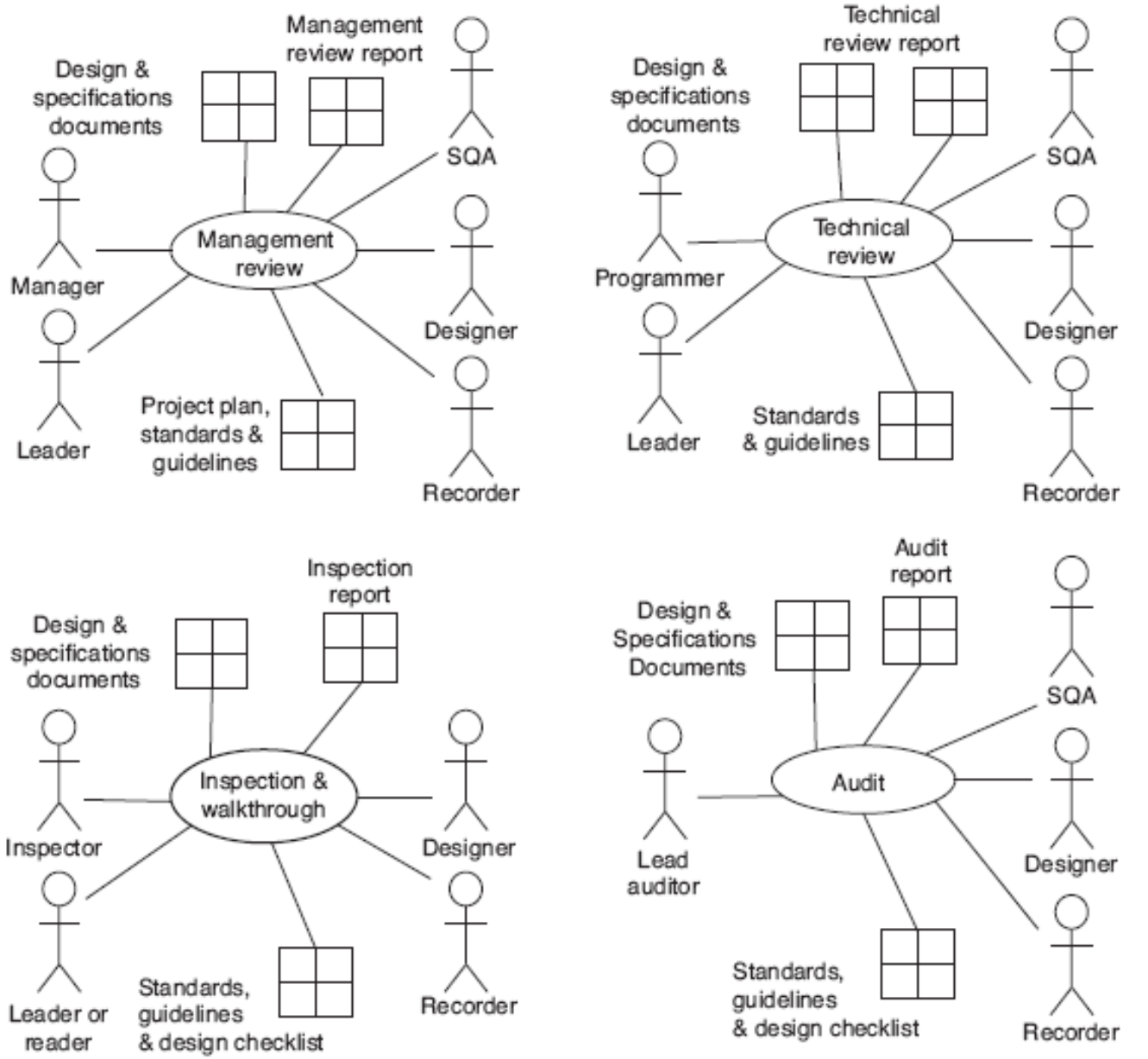


Figure 5.8 Five types of design reviews and their participants

Design inspection checklists

- Non-functional requirements
- GUI design checklist
- Desirable design features

Table 5.9 Inspection questions addressing some desirable design features

Completeness • Is each module in the design hierarchy described in details using a uniform template? • Are all functionalities covered by the design? • Are all data files described in detail? • Are all user functionalities covered by the graphical user interface design? • Are all module interfaces completely specified in terms of type and size? • Have all non-functional requirements been covered in the design?
Consistency • Are module interfaces consistent with respect to type compatibility? • Are the user interface artifacts consistent in size, color, format, and so on? • Are all modules described consistently using a common template? • Are all data consistently declared and defined?
Correctness • Is the algorithm described for each module correct? Is the logic correct? • Are the sequences of interactions in the user interface design correct according to the requirements specifications? • Is the design hierarchy correct? Does it have the appropriate levels of coupling and cohesion? • Is the database design correct? Are all relation schemas in the appropriate normal forms? • Are all data variables, data structures, and files properly initialized? • Have all non-functional requirements in the design been dealt with properly?

Non-ambiguity and readability • Is the logic of each module clear and simple? Does it lead to a single interpretation by the programmer? • Is the user interface clear and simple?
Modifiability and changeability • Is it easy to modify the high-level design hierarchy if new functionalities are added? Is the software to be ported to a different platform? • Is it easy to modify the database design to meet new data requirements? • Is it easy to modify the user interface design to deal with new or changed functionalities?
Robustness and fault tolerance • How is a module algorithm dealing with an exception of abnormal input values if applicable? • How is the user interface dealing with illegal inputs at different contexts?
Traceability • Are all requirements specifications covered in the design documents? • Can each module be referred to in order to address some elements of the requirements specifications?
Testability • Is each module easy to test on its own? • Can the modules be integrated easily and then tested? • Does critical code include probes to help in testing and diagnosing errors?

Design walkthrough

- Designer presents main ideas and driving principles of the design
- Designer walks through the design documents highlighting the design alternatives and the rationale for the selected design option
- Suggested changes are recorded
- Review outcome: accept as is, accept with minor changes, accept with major changes and a follow up review, or redesign.
- Typically conducted after a design inspection review.

Audit review

- Unbiased assessment to check the conformance of the design document to applicable standards, guidelines and regulations.
- Conducted by the Software Quality Assurance (SQA)
- May use the services of an external audit organization