Appendix A

Software Requirements Specification Template
To Be Used with NSE

This document is an outline for specifying software requirements with NSE, adapted from the IEEE Guide to Software Requirements Specifications (Std 830-1998 and Std 830-1993). The corresponding file name of this document in the provided disk (or virtual “disk” held on a Web site) of this book is NSE_SRS.doc with bookmarks inserted, so that the information can be easily used in the description part of the related test cases for establishing automated and self-maintainable traceability among all the related documents and test cases and the source code (see Chap. 9 for guidance and examples). It is recommended for the customer and the software development organization to work closely together to write and maintain the software requirement specification document.

With NSE, it is recommended to use the HAETVE (Holistic, Actor–Action and Event–Response driven, Traceable, Visual, and Executable) technique (see Chap. 11) through dummy programming using dummy modules (only providing the dummy source code is good enough, providing both the source code and corresponding J-Chart is better) to replace the Use Case approach in the requirement specification.

NSE Software Requirements Specification Template
For
<Project>
Version 1.0 approved
Prepared by <author>
<organization>
<date created>
Revision History
Table of Contents

1 Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, Acronyms, and Abbreviations
   1.4 References
   1.5 Overview

2 The Overall Description
   2.1 Product Perspective
      2.1.1 System Interfaces
      2.1.2 User Interfaces
      2.1.3 Hardware Interfaces
      2.1.4 Software Interfaces
      2.1.5 Communications Interfaces
      2.1.6 Memory Constraints
      2.1.7 Operations
      2.1.8 Site Adaptation Requirements
   2.2 Product Functions
   2.3 User Characteristics
   2.4 Constraints
   2.5 Assumptions and Dependencies
   2.6 Apportioning of Requirements

3 Specific Requirements
   3.1 External Interfaces
   3.2 Functions
   3.3 Performance Requirements
   3.4 Logical Database Requirements
   3.5 Design Constraints
      3.5.1 Standards Compliance
   3.6 Software System Attributes
      3.6.1 Reliability
      3.6.2 Availability
      3.6.3 Security
      3.6.4 Maintainability
      3.6.5 Portability
   3.7 Organizing the Specific Requirements
      3.7.1 System Mode
      3.7.2 User Class
      3.7.3 Objects
1 Introduction

The introduction of the SRS should provide an overview of the entire SRS. It should contain the following subsections:

(a) Purpose
(b) Scope
(c) Definitions, acronyms, and abbreviations
(d) References
(e) Overview

1.1 Purpose

This subsection should

(a) Delineate the purpose of the SRS
(b) Specify the intended audience for the SRS

1.2 Scope

In this subsection:

(a) Identify the software product(s) to be produced by name
(b) Explain what the software product(s) will, and, if necessary, will not do
(c) Describe the application of the software being specified, including relevant benefits, objectives, and goals
(d) Be consistent with similar statements in higher-level specifications if they exist
1.3 Definitions, Acronyms, and Abbreviations

This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

1.4 References

In this subsection:

1. Provide a complete list of all documents referenced elsewhere in the SRS
2. Identify each document by title, report number (if applicable), date, and publishing organization
3. Specify the sources from which the references can be obtained

This information may be provided by reference to an appendix or to another document.

1.5 Overview

In this subsection:

1. Describe what the rest of the SRS contains
2. Explain how the SRS is organized

2 The Overall Description

This section of the SRS should describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in Sect. 3 of the SRS, and makes them easier to understand. This section usually consists of six subsections as follows:

(a) Product perspective
(b) Product functions
(c) User characteristics
(d) Constraints
(e) Assumptions and dependencies
(f) Apportioning of requirements
2.1  **Product Perspective**

This subsection of the SRS should put the product into perspective with other related products. If the product is independent and totally self-contained, it should be so stated here. If the SRS defines a product that is a component of a larger system, as frequently occurs, then this subsection should relate the requirements of that larger system to functionality of the software and should identify interfaces between that system and the software. A block diagram showing the major components of the larger system, interconnections, and external interfaces can be helpful.

This subsection should also describe how the software operates inside various constraints. For example, these constraints could include

(a)  **System interfaces**
(b)  **User interfaces**
(c)  **Hardware interfaces**
(d)  **Software interfaces**
(e)  **Communications interfaces**
(f)  **Memory**
(g)  **Operations**
(h)  **Site adaptation requirements**

2.1.1  **System Interfaces**

This should list each system interface and identify the functionality of the software to accomplish the system requirement and the interface description to match the system.

2.1.2  **User Interfaces**

This should specify the following:

(a)  The logical characteristics of each interface between the software product and its users. This includes those configuration characteristics (e.g., required screen formats, page or window layouts, content of any reports or menus, or availability of programmable function keys) necessary to accomplish the software requirements.
(b)  All the aspects of optimizing the interface with the person who must use the system.

This is a description of how the system will interact with its users. Is there a GUI, a command line or some other type of interface? Are there special interface requirements? If you are designing for the general student population, for instance, what is the impact of ADA (American with Disabilities Act) on your interface?
2.1.3 Hardware Interfaces

This should specify the logical characteristics of each interface between the software product and the hardware components of the system. This includes configuration characteristics (number of ports, instruction sets, etc.). It also covers such matters as what devices are to be supported, how they are to be supported, and protocols. For example, terminal support may specify full-screen support as opposed to line-by-line support.

2.1.4 Software Interfaces

Specify the use of other required software products and interfaces with other application systems. For each required software product, include:

1. Name
2. Mnemonic
3. Specification number
4. Version number
5. Source

For each interface, provide:

1. Discussion of the purpose of the interfacing software as related to this software product
2. Definition of the interface in terms of message content and format

2.1.5 Communications Interfaces

Specify the various interfaces to communications such as local network protocols, etc. These are protocols you will need to directly interact with.

With NSE, for improving the communication capability and efficiency, besides the developer’s internal Web site, it is recommended to share an extra Project Web Site and the BBS held in the customer site or the developer site, then make the Web pages and the index pages of the BBS traceable to the requirements and the source code (see an application example shown in Fig. 8.17).

List:

1. The URL for the index page of the shared Web site
2. The URL for the index page of the BBS
3. The URL for the feedback page
4. Other important URLs

2.1.6 Memory Constraints

Specify any applicable characteristics and limits on primary and secondary memory. Do not just make up something here. If all the customer’s machines have only
128 KB of RAM, then your target design has got to come under 128 KB so there is an actual requirement. You could also cite market research here for shrink-wrap type applications “Focus groups have determined that our target market has between 256 and 512 MB of RAM, therefore the design footprint should not exceed 256 MB.” If there are no memory constraints, so state.

With NSE, memory leak and usage violation should be checked and reported (see Fig. 16.8).

### 2.1.7 Operations

Specify the normal and special operations required by the user such as:

1. The various modes of operations in the user organization
2. Periods of interactive operations and periods of unattended operations
3. Data processing support functions
4. Backup and recovery operations

(Note: This is sometimes specified as part of User Interfaces) If you separate this from the UI stuff earlier, then cover business process type stuff that would impact the design. For instance, if the company brings all their systems down at midnight for data backup that might impact the design. These are all the work tasks that impact the design of an application, but which might not be located in software.

### 2.1.8 Site Adaptation Requirements

In this section:

1. Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode
2. Specify the site or mission-related features that should be modified to adapt the software to a particular installation

If any modifications to the customer’s work area would be required by your system, then document that here. For instance, “A 100 KW backup generator and 10,000 BTU air conditioning system must be installed at the user site prior to software installation.”

This could also be software-specific like, “New data tables created for this system must be installed on the company’s existing DB server and populated prior to system activation.” Any equipment the customer would need to buy or any software setup that needs to be done so that your system will install and operate correctly should be documented here.

### 2.2 Product Functions

This subsection of the SRS should provide a summary of the major functions that the software will perform. For example, an SRS for an accounting program may use this
part to address customer account maintenance, customer statement, and invoice preparation without mentioning the vast amount of detail that each of those functions requires. Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product. Note that for the sake of clarity:

(a) The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.
(b) Textual or graphical methods can be used to show the different functions and their relationships. Such a diagram is not intended to show a design of a product, but simply shows the logical relationships among variables.

2.3 User Characteristics

This subsection of the SRS should describe those general characteristics of the intended users of the product including educational level, experience, and technical expertise. It should not be used to state specific requirements, but rather should provide the reasons why certain specific requirements are later specified in Sect. 3 of the SRS.

2.4 Constraints

This subsection of the SRS should provide a general description of any other items that will limit the developer’s options. These include

(a) Regulatory policies  
(b) Hardware limitations (e.g., signal timing requirements)  
(c) Interfaces to other applications  
(d) Parallel operation  
(e) Audit functions  
(f) Control functions  
(g) Higher-order language requirements  
(h) Signal handshake protocols (e.g., XON-XOFF and ACK-NACK)  
(i) Reliability requirements  
(j) Criticality of the application  
(k) Safety and security considerations

2.5 Assumptions and Dependencies

This subsection of the SRS should list each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software
but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption may be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change accordingly.

2.6 Apportioning of Requirements

This subsection of the SRS should identify requirements that may be delayed until future versions of the system.

3 Specific Requirements

This section of the SRS should contain all of the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. Throughout this section, every stated requirement should be externally perceivable by users, operators, or other external systems. These requirements should include at a minimum a description of every input (stimulus) into the system, every output (response) from the system, and all functions performed by the system in response to an input or in support of an output. As this is often the largest and most important part of the SRS, the following principles apply:

(a) Specific requirements should be stated in conformance with all the characteristics described in Sect. 4.3.
(b) Specific requirements should be cross-referenced to earlier documents that relate.
(c) All requirements should be uniquely identifiable.
(d) Careful attention should be given to organizing the requirements to maximize readability.

Before examining specific ways of organizing the requirements, it is helpful to understand the various items that comprise requirements as described in Sects. 3.1–3.7.

3.1 External Interfaces

This contains a detailed description of all inputs into and outputs from the software system. It complements the interface descriptions in Sect. 2 but does not repeat information there. Remember Sect. 2 presents information oriented to the customer/user while Sect. 3 is oriented to the developer.
It contains both content and format as follows:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy, and/or tolerance
- Units of measure
- Timing
- Relationships to other inputs/outputs
- Screen formats/organization
- Window formats/organization
- Data formats
- Command formats
- End messages

3.2 Functions

Functional requirements define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as “shall” statements starting with “The system shall…”

These include:

- Validity checks on the inputs
- Exact sequence of operations
- Responses to abnormal situations, including
  - Overflow
  - Communication facilities
  - Error handling and recovery
- Effect of parameters
- Relationship of outputs to inputs, including
  - Input/Output sequences
  - Formulas for input to output conversion

It may be appropriate to partition the functional requirements into subfunctions or sub-processes. This does not imply that the software design will also be partitioned that way.

3.3 Performance Requirements

This subsection specifies both the static and the dynamic numerical requirements placed on the software or on human interaction with the software, as a whole. Static numerical requirements may include:

(a) The number of terminals to be supported
(b) The number of simultaneous users to be supported
(c) Amount and type of information to be handled
Static numerical requirements are sometimes identified under a separate section entitled capacity.

Dynamic numerical requirements may include, for example, the numbers of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms.

For example,

95% of the transactions shall be processed in less than 1 s rather than,

An operator shall not have to wait for the transaction to complete.

(Note: Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function.)

3.4 Logical Database Requirements

This section specifies the logical requirements for any information that is to be placed into a database. This may include:

• Types of information used by various functions
• Frequency of use
• Accessing capabilities
• Data entities and their relationships
• Integrity constraints
• Data retention requirements

If the customer provided you with data models, those can be presented here. ER diagrams (or static class diagrams) can be useful here to show complex data relationships. Remember a diagram is worth a thousand words of confusing text.

3.5 Design Constraints

Specify design constraints that can be imposed by other standards, hardware limitations, etc.

3.5.1 Standards Compliance

Specify the requirements derived from existing standards or regulations. They might include:

1. Report format
2. Data naming
3. Accounting procedures
4. Audit tracing

For example, this could specify the requirement for software to trace processing activity. Such traces are needed for some applications to meet minimum regulatory or financial standards. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

3.6 Software System Attributes

There are a number of attributes of software that can serve as requirements. It is important that required attributes be specified so that their achievement can be objectively verified. The following items provide a partial list of examples. These are also known as nonfunctional requirements or quality attributes.

These are characteristics the system must possess, but that pervade (or cross-cut) the design. These requirements have to be testable just like the functional requirements. It is easy to start philosophizing here, but keep it specific.

3.6.1 Reliability

Specify the factors required to establish the required reliability of the software system at the time of delivery. If you have MTBF requirements, express them here. This does not refer to just having a program that does not crash. This has a specific engineering meaning.

3.6.2 Availability

Specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart. This is somewhat related to reliability. Some systems run only infrequently on-demand (like MS Word). Some systems have to run 24/7 (like an e-commerce Web site). The required availability will greatly impact the design. What are the requirements for system recovery from a failure? “The system shall allow users to restart the application after failure with the loss of at most 12 characters of input”.

3.6.3 Security

Specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to:
• Utilize certain cryptographic techniques
• Keep specific log or history data sets
• Assign certain functions to different modules
• Restrict communications between some areas of the program
• Check data integrity for critical variables

3.6.4 Maintainability

Specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices.

3.6.5 Portability

Specify attributes of software that relate to the ease of porting the software to other host machines and/or operating systems. This may include:

• Percentage of components with host-dependent code
• Percentage of code that is host dependent
• Use of a proven portable language
• Use of a particular compiler or language subset
• Use of a particular operating system

3.7 Organizing the Specific Requirements

For anything but trivial systems, the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organizing these in a manner optimal for understanding. There is no one optimal organization for all systems. Different classes of systems lend themselves to different organizations of requirements in Sect. 3. Some of these organizations are described in the following subclasses.

3.7.1 System Mode

Some systems behave quite differently depending on the mode of operation. When organizing by mode, there are two possible outlines. The choice depends on whether interfaces and performance are dependent on mode.

3.7.2 User Class

Some systems provide different sets of functions to different classes of users.
3.7.3 Objects

Objects are real-world entities that have a counterpart within the system. Associated with each object is a set of attributes and functions. These functions are also called services, methods, or processes. Note that sets of objects may share attributes and services. These are grouped together as classes.

3.7.4 Feature

A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. Each feature is generally described in a sequence of stimulus–response pairs.

3.7.5 Stimulus

Some systems can be best organized by describing their functions in terms of stimuli.

3.7.6 Response

Some systems can be best organized by describing their functions in support of the generation of a response.

3.7.7 Functional Hierarchy

When none of the above organizational schemes prove helpful, the overall functionality can be organized into a hierarchy of functions organized by either common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships between and among the functions and data.

With NSE, it is recommended to form a document hierarchy description table using bookmarks to indicate what requirements are related to what design documents and other documents and the test scripts as well as the test case numbers, so that when users want to modify a requirement or perform requirement validation, it is easy to locate the related test cases to perform forward tracing to identify the corresponding source code.

3.8 Additional Comments

Whenever a new SRS is contemplated, more than one of the organizational techniques given in Sect. 3.7 may be appropriate. In such cases, organize the specific
requirements for multiple hierarchies tailored to the specific needs of the system under specification.

There are many notations, methods, and automated support tools available to aid in the documentation of requirements. For the most part, their usefulness is a function of organization. For example, when organizing by mode, finite state machines or state charts may prove helpful; when organizing by object, object-oriented analysis may prove helpful; when organizing by feature, stimulus–response sequences may prove helpful; when organizing by functional hierarchy, data flow diagrams and data dictionaries may prove helpful.

In any of the outlines below, those sections called “Functional Requirement i” may be described in native language, in pseudo code, in a system definition language, or in four subsections titled: Introduction, Inputs, Processing, and Outputs.

4 Supporting Information

The supporting information makes the SRS easier to use. It includes the following:

(a) Table of contents
(b) Index
(c) Appendixes

4.1 Table of Contents and Index

The table of contents and index are quite important and should follow general compositional practices.

4.2 Appendixes

The appendixes are not always considered part of the actual SRS and are not always necessary. They may include

(a) Sample input/output formats, descriptions of cost analysis studies, or results of user surveys
(b) Supporting or background information that can help the readers of the SRS
(c) A description of the problems to be solved by the software
(d) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements

When appendixes are included, the SRS should explicitly state whether or not the appendixes are to be considered part of the requirements.
5 Change Management Process

Identify the change management process to be used to identify, log, evaluate, and update the SRS to reflect changes in project scope and requirements. How are you going to control changes to the requirements? Can the customer just call up and ask for something new? Does your team have to reach consensus? How do changes to requirements get submitted to the team? Formally in writing, email, or phone call?

6 Document Approvals

Identify the approvers of the SRS document. Approver name, signature, and date should be used.

7 Delivery

Indicate what should be delivered – with NSE, it is recommended to deliver:
1. The computer program (source code and executable product)
2. The data used
3. The documents traceable to and from the source code
4. The database built through static and dynamic measurement of the program
5. A set of Assisted Online Agents (AOA) to support testability, visibility, reliability, traceability, changeability, conformity, and maintainability, including a set of AOA of NSE if the product is developed with NSE

8 Appendix

This section is optional.

Appendices may be included if any, either directly or by reference, to provide supporting details that could aid in the understanding of the Software Requirements Specifications.
Outline for SRS Section 3
Organized by mode: Version 1

3 Specific Requirements
3.1 External Interface Requirements
   3.1.1 User Interfaces
   3.1.2 Hardware Interfaces
   3.1.3 Software Interfaces
   3.1.4 Communications Interfaces
3.2 Functional Requirements
   3.2.1 Mode 1
      3.2.1.1 Functional Requirement 1.1
         ...
      3.2.1.n Functional Requirement 1.n
   3.2.2 Mode 2
       ...
   3.2.m Mode m
      3.2.m.1 Functional Requirement m.1
         ...
      3.2.m.n Functional Requirement m.n
3.3 Performance Requirements
3.4 Design Constraints
3.5 Software System Attributes
3.6 Other Requirements

Outline for SRS Section 3
Organized by mode: Version 2

3 Specific Requirements
3.1 Functional Requirements
   3.1.1 Mode 1
      3.1.1.1 External Interfaces
         3.1.1.1.1 User Interfaces
         3.1.1.1.2 Hardware Interfaces
         3.1.1.1.3 Software Interfaces
         3.1.1.1.4 Communications Interfaces
      3.1.1.2 Functional Requirement
         3.1.1.2.1 Functional Requirement 1
         ...
         3.1.1.2.n Functional Requirement n
      3.1.1.3 Performance
   3.1.2 Mode 2
      3.1.m Mode m
3.2 Design Constraints
3.3 Software System Attributes
3.4 Other Requirements
Outline for SRS Section 3
Organized by user class (i.e. different types of users → System Administrators, Managers, Clerks, etc.)

3 Specific Requirements
3.1 External Interface Requirements
   3.1.1 User Interfaces
   3.1.2 Hardware Interfaces
   3.1.3 Software Interfaces
   3.1.4 Communications Interfaces

3.2 Functional Requirements
   3.2.1 User Class 1
      3.2.1.1 Functional Requirement 1.1
      ...
      3.2.1.n Functional Requirement 1.n
   3.2.2 User Class 2
      ...
   3.2.m User Class m
      3.2.m.1 Functional Requirement m.1
      ...
      3.2.m.n Functional Requirement m.n

3.3 Performance Requirements
3.4 Design Constraints
3.5 Software System Attributes
3.6 Other Requirements

Outline for SRS Section 3
Organized by object (Good if you did an object-oriented analysis as part of your requirements)

3 Specific Requirements
3.1 External Interface Requirements
   3.1.1 User Interfaces
   3.1.2 Hardware Interfaces
   3.1.3 Software Interfaces
   3.1.4 Communications Interfaces

3.2 Classes/Objects
   3.2.1 Class/Object 1
      3.2.1.1 Attributes (Direct or Inherited)
         3.2.1.1.1 Attribute 1
      ...
      3.2.1.n Attribute n
   3.2.1.2 Functions (Services, Methods, Direct, or Inherited)
      3.2.1.2.1 Functional Requirement 1.1
      ...
      3.2.1.m Functional Requirement 1.m
3.2.1.3 Messages (Communications Received or Sent)

3.2.2 Class/Object 2

…

3.2.p Class/Object \( p \)

3.3 Performance Requirements

3.4 Design Constraints

3.5 Software System Attributes

3.6 Other Requirements

Outline for SRS Section 3

Organized by feature (Good when there are clearly delimited feature sets)

3 Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

3.1.2 Hardware Interfaces

3.1.3 Software Interfaces

3.1.4 Communications Interfaces

3.2 System Features

3.2.1 System Feature 1

3.2.1.1 Introduction/Purpose of Feature

3.2.1.2 Stimulus/Response Sequence

3.2.1.3 Associated Functional Requirements

3.2.1.3.1 Functional Requirement 1

…

3.2.1.3.n Functional Requirement \( n \)

3.2.2 System Feature 2

…

3.2.m System Feature \( m \)

3.3 Performance Requirements

3.4 Design Constraints

3.5 Software System Attributes

3.6 Other Requirements

Outline for SRS Section 3

Organized by stimulus (Good for event-driven systems where the events form logical groupings)

3 Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

3.1.2 Hardware Interfaces

3.1.3 Software Interfaces

3.1.4 Communications Interfaces

3.2 Functional Requirements

3.2.1 Stimulus 1
3.2.1.1 Functional Requirement 1.1

...  
3.2.1.n Functional Requirement 1.n

3.2.2 Stimulus 2

...

3.2.m Stimulus m

3.2.m.1 Functional Requirement m.1

...

3.2.m.n Functional Requirement m.n

3.3 Performance Requirements

3.4 Design Constraints

3.5 Software System Attributes

3.6 Other Requirements

Outline for SRS Section 3
Organized by response (Good for event-driven systems where the responses form logical groupings)

3 Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

3.1.2 Hardware Interfaces

3.1.3 Software Interfaces

3.1.4 Communications Interfaces

3.2 Functional Requirements

3.2.1 Response 1

3.2.1.1 Functional Requirement 1.1

...

3.2.1.n Functional Requirement 1.n

3.2.2 Response 2

...

3.2.m Response m

3.2.m.1 Functional Requirement m.1

...

3.2.m.n Functional Requirement m.n

3.3 Performance Requirements

3.4 Design Constraints

3.5 Software System Attributes

3.6 Other Requirements
Outline for SRS Section 3
Organized by functional hierarchy (Good if you have done structured analysis as part of your design)

3 Specific Requirements
3.1 External Interface Requirements
   3.1.1 User Interfaces
   3.1.2 Hardware Interfaces
   3.1.3 Software Interfaces
   3.1.4 Communications Interfaces
3.2 Functional Requirements
   3.2.1 Information Flows
      3.2.1.1 Data Flow Diagram 1
         3.2.1.1.1 Data Entities
         3.2.1.1.2 Pertinent Processes
         3.2.1.1.3 Topology
      3.2.1.2 Data Flow Diagram 2
         3.2.1.2.1 Data Entities
         3.2.1.2.2 Pertinent Processes
         3.2.1.2.3 Topology
      ... 
   3.2.1.n Data Flow Diagram n
      3.2.1.n.1 Data Entities
      3.2.1.n.2 Pertinent Processes
      3.2.1.n.3 Topology
3.2.2 Process Descriptions
   3.2.2.1 Process 1
      3.2.2.1.1 Input Data Entities
      3.2.2.1.2 Algorithm or Formula of Process
      3.2.2.1.3 Affected Data Entities
   3.2.2.2 Process 2
      3.2.2.2.1 Input Data Entities
      3.2.2.2.2 Algorithm or Formula of Process
      3.2.2.2.3 Affected Data Entities
      ... 
   3.2.2.m Process m
      3.2.1.m.1 Input Data Entities
      3.2.1.m.2 Algorithm or Formula of Process
      3.2.1.m.3 Affected Data Entities
3.2.3 Data Construct Specifications
   3.2.3.1 Construct 1
      3.2.3.1.1 Record Type
      3.2.3.1.2 Constituent Fields
   3.2.3.2 Construct 2
      3.2.3.2.1 Record Type
3.2.3.2 Constituent Fields

...%
3.2.3.1 Construct \( p \)
3.2.3.1.1 Record Type
3.2.3.1.2 Constituent Fields

3.2.4 Data Dictionary
3.2.4.1 Data Element 1
3.2.4.1.1 Name
3.2.4.1.2 Representation
3.2.4.1.3 Units/Format
3.2.4.1.4 Precision/Accuracy
3.2.4.1.5 Range

3.2.4.2 Data Element 2
3.2.4.2.1 Name
3.2.4.2.2 Representation
3.2.4.2.3 Units/Format
3.2.4.2.4 Precision/Accuracy
3.2.4.2.5 Range

...%
3.2.4.q Data Element \( q \)
3.2.4.q.1 Name
3.2.4.q.2 Representation
3.2.4.q.3 Units/Format
3.2.4.q.4 Precision/Accuracy
3.2.4.q.5 Range

3.3 Performance Requirements
3.4 Design Constraints
3.5 Software System Attributes
3.6 Other Requirements

Outline for SRS Section 3

Showing multiple organizations (Can’t decide? Then glob it all together)

3 Specific Requirements
3.1 External Interface Requirements
3.1.1 User Interfaces
3.1.2 Hardware Interfaces
3.1.3 Software Interfaces
3.1.4 Communications Interfaces
3.2 Functional Requirements
3.2.1 User Class 1
3.2.1.1 Feature 1.1
3.2.1.1.1 Introduction/Purpose of Feature
3.2.1.1.2 Stimulus/Response Sequence
3.2.1.1.3 Associated Functional Requirements
3.2.1.2 Feature 1.2
  3.2.1.2.1 Introduction/Purpose of Feature
  3.2.1.2.2 Stimulus/Response Sequence
  3.2.1.2.3 Associated Functional Requirements

…

3.2.1.m Feature 1.m
  3.2.1.m.1 Introduction/Purpose of Feature
  3.2.1.m.2 Stimulus/Response Sequence
  3.2.1.m.3 Associated Functional Requirements

3.2.2 User Class 2

…

3.2.n User Class n

3.3 Performance Requirements
3.4 Design Constraints
3.5 Software System Attributes
3.6 Other Requirements

Outline for SRS Section 3
Organized by HAETVE Applications

3 Specific Requirements
  3.1 External Actor Descriptions
    3.1.1 Human Actors
    3.1.2 Hardware Actors
    3.1.3 Software System Actors
  3.2 HAETVE Application Descriptions
    3.2.1 (Dummy Source Code for HAETVE) 1
    3.2.2 (Dummy Source Code for HAETVE) 2
    …
    3.2.n (Dummy Source Code for HAETVE) n
  3.3 Performance Requirements
  3.4 Design Constraints
  3.5 Software System Attributes
  3.6 Other Requirements
Appendix B
An Example About How to Realize 100\% MC/DC (Modified Condition/Decision Coverage) for a Program Unit

In this appendix, an example is used for illustrating the test coverage analysis metrics of Panorama C/C++ for Windows XP.

SUM_PRODUCT is a sample program which requests the input of three integers: LOW, HIGH, and MAX. The integers should not be negative, otherwise an error message will be given.

The source code of SUM_PRO.cpp is listed below:

```c
#include <stdio.h>

main(void)
// This program prints for each k in the range LOW to HIGH // k + k and k * k. No more than MAX number of k's are used.
{
    int low, high, max, k, n=0;
    scanf("%d %d %d", &low, &high, &max);
    printf(" LOW = %d HIGH = %d MAX = %d \n",low,high,max);
    if ( low >= 0 && high >=0 && max >= 0)
        for (k=low; k<=high; k++)
        {
            ++n;
            if (n > max)
                break;
            printf(" %d + %d = %d   %d * %d = %d\n", k, k+k, k, k, k*k);
        }
    else
        printf("Error! The input data are incorrect! \n");
}
```
Appendix B

Note: if it is for Panorama C, the file name SUM_PRO.cpp must be renamed to SUM_PRO.c.

A SUM_PRO.hsi file is generated from the Makefile of SUM_PRO.exe and loaded into the main menu of Panorama. Then, a.dbs file is created for SUM_PRO.exe. To capture the dynamic test coverage data, SUM_PRO.exe is executed with several groups of integers as listed below:

<table>
<thead>
<tr>
<th>LOW</th>
<th>HIGH</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>-2</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>8</td>
</tr>
<tr>
<td>-2</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

A series of J-Flow and J-Diagrams in OO-Diagrammer are listed to show the changes of accumulated test coverage each time when SUM_PRO.exe is executed.

Note: in this Appendix, the test coverage refers to the accumulated test coverage in order to show the result of all the executions.

Before the execution of SUM_PRO.exe, the test coverage of the code is zero. This is reflected in the Bar graph and diagrams given below (Figs. B.1–B.3):

Fig. B.1 Bar graph in OO-Diagrammer:
the test coverage data are all zeros (here, J-Coverage means MC/DC)
Fig. B.2  J-Flow in OO-Diagrammer:
all the elements are untested and highlighted
Fig. B.3  J-Diagram in OO-Diagrammer: accumulated test coverage: all the elements are untested and highlighted
To execute the sample program, type SUM_PRO.exe in the appropriate directory at the prompt:

C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: 2 8 0

LOW = 2 HIGH = 8 MAX = 0

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.

Then check the Bar graph, J-Flow, and J-Diagram in OO-Diagrammer. Select the accumulated test coverage on the corresponding Options dialog box, then click OK. The test coverage data are automatically updated (Figs. B.4–B.6):

![Bar graph](image)

Fig. B.4 Bar graph in OO-Diagrammer:

after the first execution of sum_pro.exe, the test coverage results are to be improved
Fig. B.5  J-Flow in OO-Diagrammer:
after the first execution of sum_pro.exe
Fig. B.6  J-Diagram in OO-Diagrammer: after the first execution of sum_pro.exe
Now, execute SUM_PRO.exe again. This time three integers 10, 20, and 12 are inputted. SUM_PRO.exe outputs, from 10 to 20, 11 groups of equations:

```
C: >\Func\SUM_PRO\sum_pro.exe
Enter positive integers LOW, HIGH, and MAX: 10 20 12
LOW = 10 HIGH = 20 MAX = 12
10 + 10 = 20 10 * 10 = 100
11 + 11 = 22 11 * 11 = 121
12 + 12 = 24 12 * 12 = 144
13 + 13 = 26 13 * 13 = 169
14 + 14 = 28 14 * 14 = 196
15 + 15 = 30 15 * 15 = 225
16 + 16 = 32 16 * 16 = 256
17 + 17 = 34 17 * 17 = 289
18 + 18 = 36 18 * 18 = 324
19 + 19 = 38 19 * 19 = 361
20 + 20 = 40 20 * 20 = 400
```

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Then check the Bar graph, J-Flow, and J-Diagram in OO-Diagrammer. Select the accumulated test coverage data on the corresponding Options dialog box, then click OK. The test coverage data on the diagrams are automatically updated (Figs. B.7–B.9):

![Test coverage graph](image)

**Fig. B.7** Bar graph in OO-Diagrammer:
the test coverage data have increased significantly
Fig. B.8  J-Flow in OO-Diagrammer:
accumulated test coverage after the second execution
Fig. B.9  J-Diagram in OO-Diagrammer: accumulated test coverage:
the number of unexecuted elements highlighted has been greatly decreased compared with the
diagrams before
Now, execute SUM_PRO.exe again to increase its test coverage further. This time integers 10, 1, 11 are inputted.

```
C: >\Func\SUM_PRO\sum_pro.exe
```

*Enter positive integers LOW, HIGH, and MAX:* **10 1 11**

**LOW = 10 HIGH = 1 MAX = 11**

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since Low =10 > High = 1, no equation is outputted this time.

Then check the Bar graph, J-Flow, and J-Diagram in OO-Diagrammer. Select the accumulated test coverage on the corresponding Options dialog box, then click OK. The test coverage data are automatically updated (Figs. B.10–B.12):

![Bar graph in OO-Diagrammer: accumulated test coverage](image)

**Fig. B.10** Bar graph in OO-Diagrammer: accumulated test coverage: compared with Fig. B.7, one more branch and one more segment are tested. Consequently, J-Coverage is increased by one too
Fig. B.11 J-Flow in OO-Diagrammer: accumulated test coverage: compared with Fig. B.8, one more segment (branch) is tested.
Fig. B.12 J-Diagram in OO-Diagrammer: accumulated test coverage: compared with Fig. B.9, one more segment (branch) is tested.
Now, carefully observe the J-Flow or J-Diagram, you may find out that the condition test coverage should be increased. Since Condition True has reached 100% coverage, the Condition False needs to be increased.

C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: 2 8 -2

LOW = 2 HIGH = 8 MAX = -2

Error! The input data are incorrect!
The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since a negative integer is inputted, an error message is given this time.

Then check the Bar graph, J-Flow, and J-Diagram in OO-Diagrammer. Select the accumulated test coverage on the corresponding Options dialog box, then click OK. The test coverage data are automatically updated (Figs. B.13–B.15):

![Bar graph in OO-Diagrammer: the accumulated test coverage of SC0 branch has reached 100%. J-Coverage is increased too](image)
Fig. B.14  J-Flow in OO-Diagrammer:
accumulated test coverage: only two conditions are untested
The program prints for each k in the range LOW to HIGH // k + k and k * k. No more than MAX number of k's are used.

```c
main(void)
{
    int low, high, max, k, n=0;
    printf("Enter positive integers LOW, HIGH, and MAX: ");
    scanf("%d %d %d", &low, &high, &max);
    printf(" LOW = %d HIGH = %d MAX = %d\n", low, high, max);

    if (low >= 0 && high >= 0 && max == 0)
    {
        for (k=low; k<=high; k++)
        {
            ++n;
            if (n>max)
            {
                break;
            }
            if_invseg s4(1)
            {
                printf(" %d + %d = %d %d * %d = %d\n", k, k, k+k, k, k, k*k);

                for_invseg s6(2)
                for_invseg s7(3)
            }
        }
    }
    else
    {
        printf("Error! The input data are incorrect\n");
    }
}
```

**Fig. B.15** J-Diagram in OO-Diagrammer:
accumulated test coverage: only two conditions are untested
To increase the coverage of Condition False, run SUM_PRO.exe again and input another group of integers. This time, integer High is negative.

C: >\Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: 2 -2 8

LOW=2 HIGH=-2 MAX=8

Error! The input data are incorrect!

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since a negative integer High is inputted, an error message is given too.

Then check the Bar graph, J-Flow, and J-Diagram in OO-Diagrammer. Select the accumulated test coverage in the corresponding Options dialog box, then click OK. The test coverage data are automatically updated (Figs. B.16–B.18):

Fig. B.16 Bar graph in OO-Diagrammer:
J-Coverage has been increased
Fig. B.17  J-Flow in OO-Diagrammer: accumulated test coverage: only one False condition is untested
Fig. B.18 J-Diagram in OO-Diagrammer:
accumulated test coverage: only one False condition is untested
To cover all the conditions, run SUM_PRO.exe again and input another group of data with negative Low integer.

C: \Func\SUM_PRO\sum_pro.exe

Enter positive integers LOW, HIGH, and MAX: \textbf{-2 2 8}

LOW=\textbf{-2} \textit{HIGH}=\textbf{2} \textit{MAX}=\textbf{8}

Error! The input data are incorrect!

The bold characters above are typed in at the prompts, while the italic characters are displayed by the sample program SUM_PRO.exe.

Since a negative integer Low is inputted, an error message is given too.

Then check the Bar graph, J-Flow, and J-Diagram in OO-Diagrammer. Select the accumulated test coverage on the corresponding Options dialog box, then click OK. All the conditions should have been covered (Figs. B.19–B.21):

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig_b19.png}
\caption{Bar graph in OO-Diagrammer: accumulated test coverage: all the test coverage metrics have been reached 100\%}
\end{figure}
Fig. B.20  J-Flow in OO-Diagrammer:
accumulated test coverage: the program sum_pro.exe is completely tested
main(void)
// This program prints for each k in the range LOW to HIGH
// k + k and k * k. No more than MAX number of k's are used.
{
int low, high, max, k, n=0;
printf("Enter positive integers LOW, HIGH, and MAX: ");
scanf("%d %d %d", &low, &high, &max);
printf(" LOW = %d HIGH = %d MAX =%d\n",low,high,max);

if (low>=088&&high>=088&&max>=0)
  for (k=low;k<high;k++)
  {
    ++n;
    if (n>max)
      break;
    if_inv.seg s4(1)
      printf("%d + %d = %d  %d * %d = %d\n", k, k, k+k, k, k*k);
      for_inv.seg1 s6(2)
      for_inv.seg2 s7(3)
  }
else
  printf("Error! The input data are incorrect!\n");
From the example above, it is clearly shown how test coverage data are displayed on J-Flows and J-Diagrams and how the result shown may help you to increase the coverage of your program.

Similarly, other tools of Panorama C/C++, such as the structure charts, software metrics diagrams, reports, ActionPlus diagrams, etc., can also show the dynamic test data vividly and help you successfully plan any further testing.
Appendix C
How to Control/Simulate the Return Values of a Program Unit Being Tested

There are two ways provided: (1) using a global/static variable to control the return values from a called program unit and (2) using files to control the return value of a program unit according to the execution times.

1. The source code (here the statements shown in bold are inserted for getting/controling the return value in a function call statement)

(1) main.c
#include <stdio.h>
extern int sub1();
extern int sub2();

void main(argc, argv)
int argc;
char **argv;
{
int i, returned_value1, returned_value2;
if(argc == 1)
{
    printf("Order returned_value \n");
    for (i=1; i<7; i++)
    {
        returned_value1=sub1();
        printf("%d \t %d
", i, returned_value1);
    }
}
else
{
    returned_value2=sub2();
    // printf ("#############################################
";
    printf("The retuned value from a unit called by the unit being tested: %d\n", returned_value2);
}
printf("\n");
}

(2) trouble.c (a sample application program)
/ * trouble.c */
#include <stdio.h>

static int flag = 0;
int send_value1()
{
++ flag;
switch (flag)
{
    case 1: return -1;
    case 2: return 0;
    case 3: return 1;
    case 4: return 32797;
    case 5: return 32798;
    case 6: return 32799;
}
/* The original statements. */
}

int send_value2()
{
    char order_file_name[]="order_file.txt";
    char data_file_name[]="data_file.txt";
    char line_buf[1024];
    FILE *f_order, *f_data;
    int order, new_order, line_number, value;
    if ((f_order = fopen(order_file_name, "r")) == NULL)
    {
        printf("Error in opening the file of %s \n", order_file_name);
        return -1;
    }
    if((fgets(line_buf, sizeof (line_buf), f_order))!=NULL)
    {
        sscanf(line_buf, "%d", &order);
        fclose (f_order);
        new_order = order +1;
        if((f_order = fopen(order_file_name, "w")) == NULL)
        {
            printf("Error in reading the file of %s \n", order_file_name);
            return -1;
        }
        fprintf(f_order, "%d", new_order);
        fclose (f_order);
    }
    else
    {
        printf("Error in reading data from the file of %s \n", order_file_name);
        return -1;
    }
    if((f_data = fopen(data_file_name,"r")) == NULL)
    {
        printf("Error in reading the data file of %s \n", data_file_name);
return -1;

line_number = 1;
while ((fgets(line_buf, sizeof(line_buf), f_data)) != NULL)
{
    if (line_number == order) /* It is the right value to be used. */
    {
        sscanf(line_buf, "%d", &value);
        fclose(f_data);
        return value;
    }
    ++line_number;
}
fclose(f_data);
/* The original code statements. */
}
int sub1()
{
    /* Original code statements */
    return send_value1();
}
int sub2()
{
    /* Original code statements */
    return send_value2();
}

2. Other files
(1) order_file.txt
1
(2) save_order_file.txt
1
(3) data_file.txt
-1
0
1
32797
32798
32799

3. The Makefile
# Some n make macros for building Win32 applications
CPU=i386
cc=cl
link=link
cflags=-c

all: trouble.exe
OBJJS=main.obj trouble.obj
trouble.obj: trouble.c  
$(cc) $(cflags) $*.c  
main.obj: main.c  
$(cc) $(cflags) $*.c  
panounit.exe: $(OBJJS)  
 $(link) -out:panounit.exe -subsystem:console main.obj trouble.obj libc.lib kernel32.lib

4. The batch file: run_panounit.bat  
copy save_order_file.txt order_file.txt  
panounit  
panounit 1  
panounit 2  
panounit 3  
panounit 4  
panounit 5  
panounit 6
Appendix D
Hints for Answering the “Points and Questions to Ponder” in Each Chapter

*Note:* hints are not real answers, but something to help you make your answers.

Chapter 1

*Points and Questions to Ponder*

(a) What are the major differences between the traditional software definition and the new one defined with NSE? Do you think it is necessary to provide a software product to the customer (not the end-user) with the database built through static and dynamic measurement of the product, and a set of Assistant Online Agents? Why?

**Hints:**
Traditional: *Software* = program + data + document, but the program and the documents are separated without traceability that is established automatically.

With NSE: *Software* = program + data + documents traceable to and from the source code, plus the *database* built through static and dynamic measurement of the program, and a set of *Assisted Online Agents* (automated and intelligent tools working with the program and the database) for handling the issue of complexity and supporting the testability, visibility, changeability, conformity, reliability, and traceability – making the software product adaptive and truly maintainable in the new working environment at the customer site, and that the requirement validation and the acceptance testing can be done dynamically in a fully automated way with mouse clicks only.

(b) Are today’s software products sufficiently engineered? Why?

**Hints:** Today’s software products are not sufficiently engineered, because the old-established software engineering paradigm is based on reductionism and the superposition principle that the whole of a complex system is the sum of its components, so that almost all software engineering tasks and activities are performed partially and locally; many critical problems exist such as … (Please complete it).

(c) What are the common limitations existing with current software process models?
Hints: About the common limitations with the current software process models, refer to Section 1.4.5.2.

(d) For efficiently supporting software maintenance, what conditions do you think a process model or software development approach should satisfy?

Hints: The following conditions:

1. Being able to help users perform software maintenance holistically and globally.
2. Being able to greatly reduce the amount of defects introduced into the software product and propagated to the software maintenance phase through defect prevention and defect propagation prevention performed from the first step to the entire software development process.
3. Being able to help users prevent the side effects for the implementation of requirement changes or code modifications.
4. Being able to provide the necessary means to help users greatly reduce the time, resources requested, and cost in regression testing after the implementation of requirement changes or code modifications, such as the capability for test case efficiency analysis and test case minimization, or automated, efficient, and intelligent test case selection.
5. Being able to help the customer side to maintain a software product with almost the same conditions as if the software product is maintained by the product development side.

(e) Although the software engineering paradigm itself is a complex system consisting of many related parts which are connected closely and interactively, some people still believe that only improving one or two parts of the software engineering paradigm without improving its other parts can still dramatically improve the overall characteristics, performance, behavior, and the problem-solving capability of the software engineering paradigm – do you agree with their conclusion? Why?

Hints: No. According to the Holism principle of complexity science, the characteristics and behaviors of the whole of a complex system emerge from the interaction of its parts, and cannot be inferred simply from the behavior of its individual components …(please complete it).

Chapter 2

(a) How is a successful project defined?

Hints: The definition of a successful project is one that completed within 10% or so of its committed cost and schedule and delivered all of its intended functions.

(b) What is the root cause that about 70% of software projects are failures?
Hints:

1. The foundation of the old-established software engineering paradigm is based on linear thinking, reductionism, and the superposition principle.
2. The old-established software engineering paradigm is outdated including the process models, the software development methodologies, the software testing paradigm, the quality assurance paradigm, the maintenance paradigm, etc. (please read Sects. 2.5–2.12, write down your notes, then close this book and make your answer)

Chapter 3

(a) What is complexity science?

Hints:

Read Sect. 3.1, write down your notes, then close this book and make your answer.

(b) What are the major differences between Reductionism and Holism?

Hints:

Compare the difference: the superposition principle and the Holism principle (about how to handle the relationship between the whole of a complex system and its parts).

(c) What are the essential principles of complexity science? How are they related to the establishment of NSE?

Hints:

Read Sect. 3.2, write down your notes, then close this book and make your answer.

The essential principles of complexity science are the foundation for establishing NSE.

Chapter 4

(a) What are the major differences between Hall’s framework and FDS?

Hints:

Consider the differences in the four aspects:
1. The objectives
2. The phases being performed; follow or do not follow a linear order
3. The contents of the axes
4. The use of computer simulation
(b) Why is it recommended to apply complexity science to solve the problems of a complex system in an industry through two major steps (the first one is to complete the paradigm shift by the organization performing the tasks or a tool vendor, then the second one is to handle the detailed tasks by applying the corresponding new paradigm established in the first step)?

Hints:

The “Sunlight” of complexity science cannot directly “Reach” the target without removing the big “Umbrella” in the middle – the old-established paradigm… (please complete it)

Chapter 5

(a) What are the major problems existing with today’s software development? Why are those problems so hard to solve?

Hints:

1. Consider the issues of quality, productivity, cost, risk, missed schedules, blown budgets, and flawed products …
2. Those problems cannot be solved by a single development, in either technology or management technique – they are caused by the entire existing software engineering paradigm based on reductionism and superposition principle including the linear process models …

(b) Why does today’s software maintenance take 75% or more of the total effort and total cost in software product development?

Hints:

1. With the linear process models, huge amounts of defects will be introduced into a software product
2. The defects easily propagate down to the maintenance phase
3. The implementation of requirement changes and code modifications is performed partially and locally without the means to prevent the side-effects …
4. The process and the result in software maintenance are invisible.

(c) What is NSE?

Hints:

Consider:
1. The objectives
2. The foundation
3. The major features

(Read Sect. 1.7, write down your notes, then close this book and make your answer.)
Chapter 6

(a) What are the driving forces for the establishment of NSE (Nonlinear Software Engineering paradigm)? Describe them in as much detail as possible.

Hints:

Describe each technique listed in Fig. D.1:

(b) Which principles do the techniques introduced in this chapter comply with? Why?

Hints:

The essential principles of complexity science, including the Nonlinearity principle, the Holism principle, the … (read Chap. 3 for the detailed description of the principles).
Chapter 7

(a) What are the major differences between the NSE software visualization paradigm and the traditional software visualization paradigm?

Hints:

Briefly compare Sects. 7.1 and 7.2.

(b) What are the major benefits of virtually existing charts and diagrams without storing hard copies in the hard disk and the memory of a computer?

Hints:

Consider: (1) the space saved; (2) the time spent in loading the graphics and in displaying the graphics.

(c) Point out the reasons why a system-level call graph or diagram should be made interactive and traceable.

Hints:

Consider:

1. The difference between static graphics and dynamic graphics – which one is more useful?
2. About traceability, compare the following Figs. D.2 and D.3, then make your answer
Fig. D.2 A call graph shown in J-Chart
Fig. D.3 A module and the related modules highlighted
(d) Write three small programs for generating the three charts shown in Fig. D.4 separately through dummy programming, then compile them and run the executable programs to correct possible defects.

Hints:

Refer to Fig. D.5, design your three dummy programs, then use S_Panorama for C/C++ or S_Panojava for Java (see Preface) to generate the call graphs and correct possible bugs:
Chapter 8

(a) About the software process model, “There has to be upstream movement” – why?

Hints:
Without upstream movement, defects introduced in the upper phases will easily propagate to the lower phases, and the defect removal cost will increase tenfold several times – now software maintenance takes 75% or more of the total effort and total cost of software development.

(b) Why is there no upstream movement at all in all the existing software process models (excluding the NSE process model)?

Hints:
Please consider what is the foundation of those models.

(c) Why should software maintenance be performed globally and holistically? How can software maintenance be performed globally and holistically?

Hints:
1. Find out what are called “Butterfly-effects”
2. Consider what kinds of traceability are needed for performing software maintenance holistically and globally.

(d) Is a modified waterfall model with feedback as shown in the following figure (Fig. D.6) a linear model or not? Why?
Appendix D

Hints:

Read Sect. 1.4 for answering this question.

(e) List the drawbacks of a linear life cycle model without upstream movement.

Hints:

Read Sects. 2.5 and 8.2 carefully, then make your answer.

(f) What are the major differences between the NSE process model and the existing process models?

Hints:

Read Sect. 8.10, then make your answer.

Chapter 9

(a) Why is software traceability, particularly requirement traceability, so important?

Hints:

Please read


After that, make your answer.

(b) Why should a bookmark be used to open a related document that is traced automatically?
Hints:

1. Usually a requirement specification file will include the descriptions for all requirements, so that if we do not use bookmarks, all requirements traced will be shown from the beginning location of the requirement specification file – it will cause confusion about exactly which requirement is traced.
2. Using bookmarks, we can open the related document and show it from the location indicated by a bookmark.
3. Bookmarks will not affect the contents.
4. When the contents of a document are modified, in most cases the bookmarks will automatically point to the new locations without manual modification.
5. Try to set some bookmarks in a document, then modify the contents, and view the documents again using the bookmarks to see what happens.

(c) What are the benefits to use Time Tags for implementing the bidirectional traceability between the test cases and the source code?

Hints:

Consider the following points to make your answer:

1. Automation
2. Accuracy
3. Self-maintainability

(d) What are the major features of this automated and self-maintainable traceability?

Hints:

Read Sect. 9.5 and the article “Software Requirements Traceability Remains a Challenge” [Kan09], then make your answer.

(e) Where do you think this automated and self-maintainable traceability can be efficiently used in software engineering?

Hints:

Consider the following points to make your answer:

1. Identify and fix the inconsistency defects among documents, and between documents and source code
2. Prevent side effects in the implementation of requirement changes and code modification
3. Perform regression testing efficiently
4. Validate and verify the product efficiently
5. Make the documents generated by third party tools also traceable
6. Automate the acceptance testing

(f) How can this automated and self-maintainable traceability be used to make a document produced by a third party tool traceable with the requirements of a project being developed using this technique and tools?
Hints:

Using batch files – try your own examples before making your answer.

Chapter 10

(a) What are the differences in software development methodology between that based on Constructive Holism and that based on Generative Holism?

Hints:

Consider the following points to make your answer:

1. The whole and its parts of a complex system, which one comes first?
2. With the software development method based on Generative Holism, we can begin user testing very early, and we can adopt a build-to-budget strategy that protects absolutely against schedule or budget overrun (at the cost of possible functional shortfall).

(b) What are the major differences between RUP (Rational Unified Process) and the NSE software development methodology?

Hints:

Compare Figs. 1.57 and 1.58, and Sect. 23.7, then make your answer.

(c) How can the NSE software testing paradigm be dynamically used in upstream quality assurance for defect prevention and defect propagation prevention?

Hints:

Read Chap. 16, then consider the following points to make your answer:

1. Having an output is no longer a condition to dynamically use the Transparent-box method for software testing.
2. In the case where there is no output in the execution of a test case, we can specify the expected execution path in the test case description part, and then check whether the real execution path covers the expected path to find logic defects.
3. With the HAETVE technique (see Chap. 11), the requirement development work products and graphic design documents are generated from dummy programs which are executable.
4. After the execution of the test cases, the bidirectional traceability facility will be established for checking the consistency among the documents and test cases and source code.

(d) How can the NSE software visualization paradigm be used in software defect prevention, defect propagation prevention, software understanding, testing, and maintenance?
Hints:

Read Chap. 7 and see Figs. 10.10 and 10.11 to make your answer.

Chapter 11

(a) What are the major differences between the Use Case approach and the HAETVE technique?

Hints:

Consider the following points to make your answer:

1. Holism
2. Visibility
3. Maintainability
4. Traceability for static review and defect removal
5. Execution for dynamic defect prevention and defect propagation prevention
6. Whether it is suitable for Event–Response type applications

(b) Why should the graphical result of the function decomposition of the functional requirements of a product be made traceable?

Hints:

Consider the following points to make your answer:

1. Visibility
2. Big Picture for program understanding
3. Static review for defect removal
4. Support for incremental development (to assign orders for incremental unit coding and testing)

(c) Why do we need not only static review, but also dynamic testing in the software requirement development phase?

Hints:

1. Read Capers Jones’ article “Software quality in 2002: a survey of the state of the art” (Six Lincoln Knoll Lane, Burlington, Massachusetts 01803 http://www.SPR.com 23 July 2002) to know
   • Usually how many percent of the defects are introduced in the requirement development phase
   • The impact of those defects introduced in the requirement development phase
   • The difficulty to remove the defects introduced in the requirement development phase
2. Many defects are hard to find without dynamic execution of the program.
3. Without dynamic testing, it is impossible to establish automated and self-maintainable traceability for detecting the inconsistent defects among the related documents and test cases and source code.

Chapter 12

(a) What are the major problems with today’s software design?

Hints: Read Sect. 12.1 carefully to make your answer.

(b) What are the benefits to use the Software Synthesis Design (and Incremental growing up) technique for software design?

Hints: Read Sects. 12.2 and 12.5 carefully to make your answer.

(c) Complete the dummy program for generating the top-down design result shown in Fig. D.7:

![Call Graph Through Top-Down Design](image)

Hints: Read Sect. 12.3 carefully, particularly the corresponding dummy programs, to complete your dummy program design, then use the S_Panorama or S_Panojava (see Preface) tools to verify your dummy program design.

Chapter 13

(a) What are the major problems existing with today’s software programming?

Hints: Read Sect. 13.1 carefully (write down your notes, then close this book) to make your answer.
(b) Why, with NSE, does software design become precoding and coding become further design?

Hints: With NSE, the dummy programs used for design to generate graphical design documents can be directly extended to become the source code in the coding process, and the extended source code can be directly used to generate the graphical design documents to update the design.

(c) Why should all commercial software products satisfy 100% MC/DC test coverage?

Hints: Read Sects. 13.4 and 16.4 carefully, and consider the following points to make your answer:

1. If the MC/DC test coverage is low, or if we only obtain high branch-level test coverage or high statement-level test coverage, many logic paths may not be tested.
2. Once the execution conditions are satisfied for those untested paths in the customer site, software disasters may happen.
3. As introduced in Appendix B, with NSE it is not very difficult to get 100% MC/DC test coverage result for program units, because NSE visualization tools can highlight any untested branches and conditions graphically to help users design the necessary test cases.

Chapter 14

(a) Why should a software product be tested before its application?

Hints: Read Sect. 14.1 carefully (write down your notes and then close this book) to make your answer.

(b) How many kinds of tests are needed?

Hints: Read Sects. 14.3–14.13 carefully (write down your notes and then close this book) to make your answer.

(c) Can a software product be tested manually only, without using tools?

Hints: Consider the following points to make your answer:

1. Can structural testing be performed manually?
2. Can performance testing be performed manually?
3. Can memory leak be completely checked manually?
4. Can the execution path of a runtime error be traced manually?
5. Can GUI test operations be captured and played back manually?
6. Can load testing for Web applications be performed manually?

(d) Who should test a software product – the product developers, other teams or groups but not the developers, or both? Why?
Hints: Consider the following points to make your answer:

1. The types of defects
2. The required knowledge to test a software product
3. The timing and the difference of the cost

Chapter 15

(a) What is a test case?

Hints: Read Sect. 15.1 then make your answer.

(b) How many basic test case design methods are used today?

Hints: Read Sect. 15.2, write down your notes, then close this book and make your answer.

(c) How can the NSE software testing paradigm and the NSE software visualization paradigm help users design efficient test cases?

Hints: Read Sect. 15.3, write down your notes, then close this book and make your answer.

(d) Describe the simple rules for writing test cases for using the Transparent-box software testing method and tools.

Hints: Read Sect. 15.6, write down your notes, then close this book and make your answer.

Chapter 16

(a) Why is it that the existing software testing methods, techniques, and tools cannot be dynamically used in the software requirement development phase and the software design phase?

Hints:

1. With the old-established software engineering paradigm, there is nothing executable in the requirement development phase and the design phase – people will think there is no need to perform software testing dynamically.
2. The existing software testing methods, techniques, and tools are mainly based on Black-box testing which is used to compare the output with the expected value, so that it can only be used dynamically after production.

(b) Why are software functional testing and structural testing performed separately with today’s software testing paradigm?
Hints: Consider

1. Today, in functional testing people mainly use the Black-box testing method.
2. Most people think the purposes of the two kinds of testing are different.
3. It is difficult to combine them together without automatic and visual tools.

(c) Software disasters happen often – is it related to the drawbacks of the existing software testing methods, technologies, and tools? Why?

Hints: Consider

1. Today, in what phases is software testing dynamically performed?
2. Why did NIST (National Institute of Standards and Technology) conclude that “Briefly, experience in testing software and systems has shown that testing to high degrees of security and reliability is from a practical perspective not possible. Thus, one needs to build security, reliability, and other aspects into the system design itself and perform a security fault analysis on the implementation of the design.” (“Requiring Software Independence in VVSG 2007: STS Recommendations for the TGDC,” November 2006 http://www.vote.nist.gov/DraftWhitePaperOnSIinVVSG2007-20061120.pdf)?

(d) Why can and should the Transparent-box testing method and the corresponding tools be dynamically used in the requirement development phase and the design phase?

Hints:

1. Different from the old-established software engineering paradigm with which there is nothing executable in the requirement development phase and the design phase (except for prototype design), with NSE the executable dummy programs are used for requirement development and product design, so that dynamic testing is needed for ensuring the quality of the product.
2. To the Transparent-box testing method, having an output is no longer a condition to dynamically use the method and the corresponding tool – in the cases where there is no output, it will check whether the execution path covers the expected path specified using J-Flow notations for helping users remove logic defects, and it can also establish the bidirectional traceability facility to help users remove inconsistent defects.
3. The forward and backward tracing processes are supported by a set of visual tools.

(e) What are the key points in designing test cases for software testing using the Transparent-box method?

Hints: Read Chap. 9 and Sect. 15.6 carefully (then close this book) to make your answer.

(f) What are the major differences between the old-established software testing paradigm and the NSE software testing paradigm?
Hints: Read Sect. 16.5 carefully (and then close this book) to make your answer.

Chapter 17

(a) What is the root cause that regarding software product quality, “Over the last 50 years there has been very little improvement?”

Hints: Read Sects. 2.3 and 17.1 carefully, write down your notes, then close this book and make your answer.

(b) What is defect prevention? Why should it be performed in the entire software development lifecycle from the first step down to the retirement of a software product?

Hints: Read Sect. 17.3.4, then consider the following points to make your answer:

1. The timing
2. The cost savings
3. The support for changeability
4. The possibility to extend the life time of a software product
5. “An ounce of prevention is worth a pound of cure!”

(c) What are the major differences between the old-established software quality assurance paradigm and NSE-SQA?

Hints: Compare Sects. 17.1 and 17.2, write down your notes, then close this book and make your answer.

Chapter 18

(a) Why does software maintenance take 75% or more of the total effort and total cost in software product development today?

Hints: Read Sect. 18.1, write down your notes, then close this book and make your answer.

(b) What are the major differences between the old-established software maintenance paradigm and the NSE software maintenance paradigm?

Hints: Compare Sects. 18.1 and 18.2, write down your notes, then close this book and make your answer.

(c) How can the side effects in the implementation of requirement changes or code modifications be prevented?
Hints:
1. Perform software maintenance holistically and globally.
2. Use various traceabilities.
3. Read Sect. 18.4, write down your notes, then close this book and make your answer.

(d) When a software product is made through outsourcing development, what should be provided with the product? Why?

Hints: Read Sect. 1.1, particularly about the new definition of software, write down your notes, then close this book and make your answer.

Chapter 19

(a) What are the major issues existing with the old-established software documentation paradigm?

Hints: Read Sect. 19.1, write down your notes, then close this book and make your answer.

(b) Is source code the best documentation for a program? Why?

Hints: Consider the following points to make your answer:

1. Text and graphics – which one do people like more for understanding a complex system?
2. Traceable and not traceable – which documents are more useful in software understanding?

(c) What are the major differences between the old-established software documentation paradigm and the NSE software documentation paradigm?

Hints: Compare Sects. 19.1 and 19.2, write down your notes, then close this book and make your answer.

Chapter 20

(a) What are the benefits of combining the project management process and product development process together to make their work products traceable?

Hints: Consider the following points to make your answer:

1. Timing in finding the possible problems
2. Timing in solving the problems found
3. The importance of getting first hand information
(b) Why should a project Web site and BBS be established and the related Web pages or BBS title pages be made traceable with the related requirements and test cases and source code?

Hints: Consider the following points to make your answer:

1. Besides regular short meetings, how can the members of the product management team and product development team communicate efficiently?
2. In case unexpected events occur, how can everybody contribute to solving the problems quickly?
3. In case a project is developed by a distributed network of teams, how can they share information better?

Chapter 21

(a) Why are software algorithms so important?

Hints: According to the popular algorithms textbook Introduction to Algorithms (Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein), “an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.” In other words, algorithms are like road maps for accomplishing a given, well-defined task (lbackstrom, Algorithm Tutorial, http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=importance_of_algorithms).

So the efficiency of problem-solving greatly depends on the algorithm used.

(b) What is a hash table? Where do we need to use hash tables?

Hints: A hash table or hash map is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person’s name) to their associated values (e.g., their telephone number) (Wikipedia, Hash Table, http://www.en.wikipedia.org/wiki/Hash_table).

Hash tables are used for sorting items such as the names of students.

Chapter 22

(a) Why do we need to use software tools?

Hints: Please consider the following points to make your answer:

1. Can every software engineering task be performed manually?
2. If a task can be done by both people and tools, which one can be used to save time and resources?
Appendix D

(b) Why should software tools be automated?

**Hints:** Please consider the following points to make your answer:

1. The efficiency
2. The maintainability of the work products

Chapter 23

(a) Why is it that “The important thing is that one model is enough – either the code or the diagrams. They should be reproducible from one another”?

**Hints:** Read Harry Sneed’s article, “The Drawbacks of Model driven Software Evolution”, IEEE CSMR 07-workshop on model-driven software evolution (MoDSE2007) Amsterdam, 20 March 2007

http://www.sciences.univ-nantes.fr/MoDSE2007/


(b) How to realize that “One model is enough – either the code or the diagrams. They should be reproducible from one another”?

**Hints:** Consider the following points to make your answer:

1. Diagrams and source code – which one is easier to change?
2. From code to diagram, and from diagram to code – which one is easier to do precisely?
3. What is offered by NSE?
4. Do you have a better solution? If you do, describe your solution in detail.

(c) Complete a small software project with NSE and the NSE support platform Panorama++.

**Hints:** It is important for you to test yourself how well you have learned from this book and how well you can apply NSE in practice. Please use the learning versions of the Panorama++ and Panojava product available to handle your small projects – try the provided application example (see Chap. 1) first.

Chapter 24

(a) Can any single development, in either technology or management technique efficiently solve the critical problems existing with today’s software development: low quality and productivity, high cost and risk? Why?
**Hints:** Read Sects. 24.5 and 24.6, write down your notes, then close this book and make your answer.

(b) Does a qualified “Silver Bullet” which is able to slay software “Werewolf” (a monster of missed schedules, blown budgets, and flawed products) mean a complete revolution in software engineering through a paradigm shift from the old one based on reductionism and the superposition principle to a revolutionary one based on complexity science? Why?

**Hints:**

1. I recommend reading one or two books introducing complexity science, or visit the Web site of complexity science map, http://www.art-science-factory.com/complexity-map_feb09.html.
2. Read Sect. 24.6, write down your notes, then close this book and make your answer.

(c) What are the major differences between the old-established software engineering paradigm and NSE?

**Hints:** Read Sect. 24.7.4, write down your notes, then close this book and make your answer.

---

**MACRO Representation in Diagrams**

For code inspection and walk through, users often prefer to having the code of a class/function diagrammed with the original source code locations (line numbers) shown before preprocessing. Panorama C/C++/OO-Browser and Panorama C/C++/OO-Diagrammer provide the logic and control flow diagrams of a class/function, with or without MACRO definition, before preprocess, thus to satisfy these users’ requirements.

For code test coverage analysis, users often want to have the code being diagrammed after preprocessing and have the unexecuted logic elements highlighted. Panorama C/C++/OO-Test provides the control flow diagram of the class/function after preprocessing with the unexecuted elements highlighted, thus to satisfy those users’ demands.

If in the case that the existence of some macro functions in the code makes it hard to identify the corresponding program logic before preprocessing, a group of lines of the code with macro functions used will be merged into one block in the diagrams.
### Glossary

<table>
<thead>
<tr>
<th><strong>Block</strong></th>
<th>A group of contiguous computer program statements that are treated as a unit.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Both</strong></td>
<td>See “Condition Both.”</td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>(1) A computer program construct in which one of two or more alternative sets of program statements is selected for execution; or (2) a point in a computer program at which one of two or more alternative sets of program statements is selected for execution; or (3) any of the alternative sets of program statements in (1).</td>
</tr>
<tr>
<td><strong>Branch testing</strong></td>
<td>Testing designed to execute each outcome of each decision point in a computer program.</td>
</tr>
<tr>
<td><strong>Class test coverage</strong></td>
<td>Class test coverage is defined as the ratio of the tested classes to the total number of classes. When one function within a class is tested, this class is considered to be tested.</td>
</tr>
<tr>
<td><strong>Condition (predicate) coverage</strong></td>
<td>Condition (predicate) coverage is defined as the percentage of both simple and compound conditions that have been tested. It is further defined as follows:</td>
</tr>
</tbody>
</table>

- **Condition True**: the percentage of true conditions and function entry points that have been tested. It can also be presented as a ratio.
- **Condition False**: the percentage of false conditions and function entry points that have been tested. It can also be presented as a ratio.
- **Condition Both**: the percentage of true and false conditions and function entry points (2 for each function) that have been tested. It is equal to the ratio of the sum of the numerators for Condition True and Condition False Coverage to the sum of their denominators.

*Note*: Function entry points are used in calculating condition coverage, because for an executed function with no conditions, its condition coverage should be 1/1 or 100%.
Control level

Program control is leveled. Level 1 begins from an entry of a program and ends at the end of the program. Each time that the control flow reaches a decision statement and its condition is satisfied, a new control level is begun. This new control level ends upon reaching the end of the executable part of the decision statement.

Cyclomatic complexity

Panorama uses a complexity metric adapted from Cyclomatic complexity metrics. The algorithm is as follows:

- Each function has a base complexity of 1 and each decision or loop statement adds a complexity of 1 to the base complexity.
- For switch statements, the user has the option of including or not including case statements as part of the calculation. When case statements are included each N-way switch will add a complexity of \((N - 1)\). When case statements are not included, the added complexity is 2.

Empty segment

(1) If there is no statement between two conjunctive nodes and the second node is a label, we define that there is an empty segment between the two nodes. (2) If the statement preceding the case statement is not an unconditional escape statement, we define that there is an empty segment between the case statement label and the conditional part of the case statement.

False

See “Condition False.”

High-level logic

The high-level logic is the table presentation of all functions or all classes within a program. The high-level logic can be displayed with the detailed logic and serves as a table in the diagrams.

Invisible segment

For each decision statement, if there is no executable statement associated with the decision statement when its condition is unsatisfied, we define there should be an invisible segment next to the decision statement (e.g., any if statement lacking an else part has an invisible segment by definition). For each repetition statement, there are two invisible segments. One of them will be executed when the program control reaches the statement but its condition is never satisfied. The other will be executed if the condition is satisfied at least once, and the program control exits the repetition body normally when the condition is no longer satisfied (vs. exiting directly from the body).

J-Coverage (condition-segment coverage)

J-Coverage metric is defined as the ratio of the number of executed visible and invisible segments plus executed outcomes of conditions to the number of all visible and invisible segments plus all outcomes of conditions in a program or program module. It is equal to MC/DC test coverage.

J-Complexity (test complexity)

JC0 (Block Test Complexity): Block Test Complexity is defined as the minimal instrumentation points required for recording all block (visible segment) test coverage data.
JC1 (Segment Test Complexity): Segment Test Complexity is defined as the minimal instrumentation points required for recording all segment test coverage data; the value of the Segment Test Complexity is equal to the sum of visible segments (including empty segments) plus invisible segments.

JC2 (Condition-Segment Test Complexity): Condition-Segment Test Complexity is defined as the minimum instrumentation points required for recording all Condition-Segment Test Coverage (J-Coverage) data. The value of JC2 is equal to the sum of all outcomes of conditions in all decision statements plus all visible segments and invisible segments. J-Complexity used alone usually denotes the Condition-Segment Test Complexity (JC2).

Node
The condition part of a decision statement, an else clause, a junction such as a label, or an entry or exit point of a program unit is called a node.

Path
A sequence of instructions that may be performed in the execution of a computer program.

Path condition
A set of conditions that must be met in order for a particular program path to be executed.

Path testing
Testing designed to execute all or selected paths through a computer program.

SC0
The Basic Segment Test Coverage (also known as the Block Test Coverage). A set of test cases of a program satisfies SC0 if all nodes and visible segments of the program have been executed at least once.

SC1
The Standard Segment Test Coverage. A set of test cases of a program satisfies SC1 if it satisfies SC0 coverage and all invisible segments of the program have been executed at least once.

SC1+
The Standard Segment Test Coverage. A set of test cases of a program satisfies SC1+ if it satisfies SC1 coverage and all the low-end invisible segments of the loops in the program have been executed at least once.
Index

A
Abstraction, 64, 238
Acceptance testing, 2, 3, 17, 57, 112, 224, 238, 252, 255, 383, 550, 699, 710
Action
diagram, 569
type, 282
Activity
business, 92, 433, 607
diagram, 323, 324, 500
linear, 199
processing, 660
prototyping, 274
upstream, 51, 97, 200, 612
Actor
descriptions, 671
external, 671
special, 280
Adaptation, 86, 229–230
Adaptive software development, 88
Agile
manifesto, 40
methods, 42
models, 57
processes, 41
software development, 40–41, 88, 230, 587–593
Agility, 41
Agent-oriented software engineering, 88, 236
Analysis
class, 192, 389
complexity, 126, 194, 378, 502, 540–541, 633, 635
control flow, 111, 120, 188, 194, 617
correspondence, 148
data, 443, 502
dynamic, 177, 500, 504
function cross reference, 120, 502
logic, 111, 120, 182, 188, 617
memory leak, 127, 216, 413, 417, 477
object-oriented, 64, 274, 666–667
path, 123, 148, 183, 232, 309, 391, 477, 633
performance, 120, 123, 127, 132, 185, 216, 219, 341, 413, 418, 454, 459, 484, 503, 536, 537, 539, 540, 542, 632
requirement, 31, 33, 34, 273
risk, 68, 131, 208, 269, 477, 607
source code, 533
static, 500, 540, 541, 543, 548, 567
structured, 62, 63, 669–670
systems, 188
test case efficiency, 3, 12, 16, 30, 76, 111, 125–126, 134, 219, 232, 233, 373, 378, 413, 481, 483, 533–537, 573, 617, 633, 635, 700
variable, 120, 127, 188, 506

731
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>standards, style</td>
<td>369, 232, 422</td>
</tr>
<tr>
<td>Complex adaptive system (CAS)</td>
<td>49, 84, 92, 611, 618</td>
</tr>
<tr>
<td>Complexity arises from simple rules principle</td>
<td>87, 95, 203</td>
</tr>
<tr>
<td></td>
<td>generative Holism principle of, 104, 201, 259, 277, 314, 316, 336, 342, 435, 491, 630, 640</td>
</tr>
<tr>
<td></td>
<td>holism principle of, 49, 87, 104, 201, 259, 271, 577, 611, 618</td>
</tr>
<tr>
<td></td>
<td>manifestation of the Essential Principles of, 225–226</td>
</tr>
<tr>
<td>Communication</td>
<td>63, 69, 76, 110, 126, 207, 223, 271, 380, 519, 521, 524, 582, 590, 617, 654, 658, 661</td>
</tr>
<tr>
<td>Component</td>
<td>important, 199, 237, 257, 409</td>
</tr>
<tr>
<td></td>
<td>reusable, 229, 259, 594</td>
</tr>
<tr>
<td></td>
<td>reuse, 229, 633</td>
</tr>
<tr>
<td>Computer aided software engineering (CASE)</td>
<td>31, 62, 63, 558, 568, 573</td>
</tr>
<tr>
<td>Computer software</td>
<td>92, 374, 594</td>
</tr>
<tr>
<td>Concept</td>
<td>29, 74, 75, 83, 86, 258, 314, 383, 469, 585, 587, 593, 597</td>
</tr>
<tr>
<td>Condition outcomes</td>
<td>140, 148, 422</td>
</tr>
<tr>
<td>Conditions combination method</td>
<td>389–390</td>
</tr>
<tr>
<td>Configuration management</td>
<td>32, 38</td>
</tr>
<tr>
<td>Constraints</td>
<td>68, 83, 86, 652–656, 659–660</td>
</tr>
<tr>
<td>Construction</td>
<td>31, 42, 51, 75, 97, 200, 593</td>
</tr>
<tr>
<td>Conversion</td>
<td>389, 513, 658</td>
</tr>
<tr>
<td>Cost estimation</td>
<td>62, 109, 207, 213, 223, 229, 251, 517, 519, 520, 524, 525, 615, 630</td>
</tr>
<tr>
<td>Coupling</td>
<td>132, 138, 317, 367, 568</td>
</tr>
<tr>
<td>CRC</td>
<td>99</td>
</tr>
<tr>
<td>Critical issues</td>
<td>92, 105, 216, 229, 239, 267, 317, 342, 494–495, 518, 577, 578, 585</td>
</tr>
<tr>
<td>Critical path</td>
<td>8, 341, 342, 347, 348</td>
</tr>
<tr>
<td>Cyclomatic complexity measurement</td>
<td>111, 140, 162, 170, 175, 176, 181, 182, 232, 343, 617</td>
</tr>
<tr>
<td>D</td>
<td>Data dictionary, 662, 663</td>
</tr>
<tr>
<td></td>
<td>Data element, 670</td>
</tr>
<tr>
<td></td>
<td>Data flow diagram, 662, 663</td>
</tr>
<tr>
<td></td>
<td>Data types, 372</td>
</tr>
<tr>
<td></td>
<td>Debugging, 72, 137, 138, 148, 179, 180, 263, 264, 276, 277, 279, 311, 418, 420, 423, 431, 437, 450, 452</td>
</tr>
<tr>
<td>Design</td>
<td>approach, 316, 406</td>
</tr>
<tr>
<td></td>
<td>automation, 93</td>
</tr>
<tr>
<td></td>
<td>changes, 263</td>
</tr>
<tr>
<td></td>
<td>constraints, 656, 659–660</td>
</tr>
<tr>
<td></td>
<td>defect, 105, 250, 410</td>
</tr>
</tbody>
</table>
Entity, 659, 662
Equivalence class partition, 388–389
Estimation, 62, 68, 109, 207, 213, 223, 229, 251, 317, 519, 520, 524, 525, 615, 630
Extreme programming (XP), 40, 41, 589–593, 595
F
Feasibility, 213
Feedback, 32, 33, 35, 82, 87, 100, 106, 131, 224, 235, 267, 520, 582, 654, 709
Floating license, 574
Flow chart, 142, 145, 146
Forms, 62, 82, 83, 560, 636
Formal methods, 62, 63
Foundation of modern civilization, 1, 57
Fractal, 82
Fractal dimension, 82
Framework
adaptable, 97
general, 101
Hall’s, 92, 93, 96, 97, 101, 701
systems engineering, 92–93
Function call graph, 138, 162, 172, 178, 190, 568
Function points, 105, 277, 336
Fundamental problem, 29, 76, 200, 469, 598, 600, 601
G
Genetic algorithm, 85–86
Group, 42, 46, 84, 86, 246, 371, 372, 378, 388, 655, 673, 679, 688, 691, 721
H
Hall’s systems engineering framework, 92, 93
Hardware, 373, 377, 380, 381, 600, 653, 654, 656, 657, 659
Hierarchical design, 210, 212, 213, 321, 328, 581
Hindsight, 557–558, 560
Holism principle of complexity science, 49, 87, 104, 201, 259, 271, 577, 611, 618
I
Idea, 51, 74, 81, 87, 97, 117, 200, 258, 315, 377, 531, 532, 534, 537–538, 540, 585, 587, 590, 593, 612
Incremental and iterative development, 51, 70, 77, 201, 612
Incremental unit coding, 8, 132, 341, 712
Index, 569, 654, 663
Industrial revolution, 91–101
Information
flows, 79, 669
hiding, 61
processing, 63, 552
sharing, 233
systems, 236
technology, 274
Inheritance, 3, 8, 111, 138, 162, 172, 188, 323, 340, 367, 568, 585, 586, 617
Index

Initial Condition Sensitivity principle, 87, 88, 95, 203
Innovation, 79, 93, 136, 199, 263–265, 521
Installation, 3, 560–565, 655
Instrumentation, 241, 242, 374–375, 532, 533, 538, 724, 725
Integration, 52, 63, 71, 131–133, 214, 215, 231, 275, 313, 341, 348, 384, 434, 565, 581, 582, 592, 593, 600, 612, 630
Integration testing, 123, 132, 216, 219, 232, 341, 349–353, 373, 378, 413
Integrity, 659, 661
Integer, 86, 371, 372, 388, 543, 672, 673, 676, 679, 682, 685, 686, 691
Intelligent agents, 2, 88, 469
Intelligent test case selection, 22, 30, 76, 109, 219, 615, 700
Intelligent version comparison, 32, 121–122, 134, 222, 233, 634, 635
Interface testing, 535
Interoperability, 372, 380
Invisible segment, 381, 422, 724, 725
ISO 9001:2000, 237

J
Java, 64, 280, 309, 378, 498, 560, 575, 579, 583–586, 648, 707
3J graphics (J-Chart, J-Diagram, and J-Flow), 100, 117, 138, 155–160, 195, 497, 584
Joint Application Design (JAD), 437

K
Keywords, 237, 241, 246, 254, 261, 264, 318, 387, 399, 412, 470, 540, 541, 573, 581, 642
Knowledge, 67, 84, 97, 101, 208, 371, 375, 715

L
Lateral thinking, 86, 87
Li-Yorke Theorem, 83
Linear system, 44, 81, 104, 201, 226
Load testing, 72, 377, 380, 381, 714
LOC, 367

M
Manpower, 76, 111, 373, 600, 617
Maintainability, 3, 76, 104, 115, 203, 469, 475, 477, 521, 650, 661, 664, 712, 720
Maintenance approach, 97, 235
cost, 29, 471, 598, 601, 602
ingeering, 467–473, 475–477, 485, 487
eexample, 472–474
Iceberg, 467, 470
issue, 588
model, 30, 58, 471, 476, 485, 700
problems, 274, 584
site, 106
support, 57, 222
tasks, 468, 475, 476, 487
technique, 133, 134, 219, 230
visualization,

MDA, 274, 578, 583

Measurement
branch execution frequency, 111, 617
Cyclomatic complexity, 111, 140, 162, 170, 175, 176, 181, 182, 232, 343, 617
database, 12, 261
memory leak, 12, 232, 353, 477, 582, 632, 649
performance, 3, 4, 6, 11, 111, 162, 176, 232, 341, 377, 477, 582, 584, 594, 617, 649
productivity, 349, 524, 526
program, 1, 3, 181, 224, 649
reports, 12, 194, 377
size, 194
support, 349, 353–362, 526
test case efficiency, 391, 395

Meta-synthesis, 84
Metrics, 262, 367, 374, 421, 424, 533–536, 567, 570, 672, 691, 694, 723, 724
Milestone, 267, 269, 635
Model
capability maturity, 39, 63
computer-independent (CIM), 274, 583
incremental, 75, 105, 595
integration, 32
iteration, 99, 105
cycle, 36, 57, 105, 235, 709
linear, 32, 33, 70, 201, 227, 235, 708
micro-waterfall, 35, 42, 43, 45, 92, 99
modified waterfall, 33
NSE, 37, 99, 227
original waterfall, 33
platform-specific, 274, 583
prototype, 36, 37, 57
quality, 278, 279
requirements, 51, 612
RUP, 39
sequential, 75, 599
SPIN, 61
spiral, 36, 37, 57
transformations, 274, 583
XP, 40, 41
Model-driven tools, 274, 583, 584
Modularity, 661
Monster of missed schedules, blown budgets, and flawed products, 49, 597, 600, 605, 607, 609, 610, 641, 642, 721
Multi-agent systems, 201

N
NIST (National Institute of Standards and Technology), 73, 104, 257, 383, 410, 463, 608, 716
No Silver Bullet (NSB), 63, 74–76, 96, 599–602
Nonlinear complex system, 69, 81, 88, 93, 94, 99, 100, 226, 238
Nonlinear system, 81, 83, 87, 129, 594
Nonlinear thinking, 30, 50, 57, 75, 76, 80, 86–88, 91–94, 97, 101, 137, 162, 234, 314, 411, 609, 611, 612
Notations
class, 287
control, 399
event-response relationship, 280
function, 287
HAETVE, 279, 497, 498
J-Chart, 3, 53, 118, 128, 132, 138, 139, 190, 232, 478, 524, 539, 543
J-Flow, 3, 148, 150, 189, 215, 225, 246, 405, 450, 541, 578, 716
mapping, 284, 398, 499
UML, 497
use case, 282, 397
use cases analysis, 498

NSE
applications, 270, 577–595
components of, 107–109, 614–615
documentation paradigm, 107, 109, 112, 260, 489–515, 615, 616, 633, 718
establishment of, 75, 79, 89, 101, 105–107, 115, 134, 409, 531, 701, 703
software development methodology, 107, 112, 183, 257–273, 279, 315, 337, 368, 439, 471, 491, 592, 614, 640, 711
software engineering visualization paradigm, 135–196
software quality assurance paradigm, 433–464
structure of, 107, 108
support techniques, 107, 109, 112, 183, 234, 615
support tools, 107, 109, 112, 557–575, 615
NSE-CMMI, 585–589
NSE-RUP, 593, 594
NSE-UML, 584–585, 587
NSE-XP, 592–593

O
Object Oriented analysis and design, 64, 274
Object-Oriented modeling, 64
Object-Oriented software engineering, 64
Object oriented technologies, 63
Obstacles, 92, 94
Open source software development, 63, 64
Openness principle, 88, 95, 96, 203
Operating system, 377, 380, 657, 661
Outsourcing, 2, 230, 487, 718

P
Package, 64, 258
Pair programming, 339, 437, 589, 591
Panorama product, 170, 558
Panorama++ product, 378, 414, 560–565
Paradigm
existing software engineering, 1, 29, 32–46, 57, 59, 77, 103–105, 702
nonlinear software engineering, 2, 48, 57, 89, 101, 107, 134, 199, 560, 565, 597, 601, 610, 619–627, 703
software maintenance, 44, 54, 56, 73–74, 103–105, 235, 304, 467–487, 578, 609, 614, 717
software project management, 44, 54, 74, 103, 104, 235, 433, 517–519, 528, 529, 614
software quality assurance, 44, 55, 72–73, 104, 105, 234, 433–464, 578, 717


Patterns, 131

Payoff, 29, 200, 471, 598, 601, 602

Performance measurement, 3, 4, 6, 11, 111, 162, 176, 232, 377, 477, 582, 584, 594, 617, 649

Portability, 579, 650, 661

Prediction, 80, 91–101

Preliminary design, 117, 214, 215, 266, 269, 317


Procedure, 62, 316, 660, 683, 719

Process
  business, 68, 655
coding, 124, 216, 340, 342, 344, 368, 714
collection, 51, 97, 200, 612
design, 179, 215, 318, 342, 344, 591
improvement, 44, 105, 600
leveling, 640
management, 32, 51, 104, 109, 110, 223, 229, 235, 251, 434, 436, 441, 517–520, 523, 524, 528, 529, 578, 580, 593, 609, 612, 615, 616, 651, 664, 718
nonlinear, 30, 32, 33, 101, 109, 315, 342, 368, 470, 592
playback, 571, 572
quality assurance, 434, 436, 463
review, 477, 591, 592
solution review, 477
testing, 111, 216, 218, 271, 361, 382, 617
tracking, 716
unified, 29, 42, 274, 593, 618

Production, 43, 46, 48, 55, 104, 105, 123, 434, 438, 463, 578, 589, 608, 610, 715

Productivity
  measurement, 349, 524, 526
  precise, 524, 526
  programming, 340, 348
  research, 463
  software, 463, 522, 575, 600, 633
  software development, 47


Project failure rate, 67–69


Project manager, 230–231

Project plan, 207, 223, 235, 582, 628, 649

Project planning, 263, 517, 520

Project scope, 42, 664

Property, 50, 87, 89, 106, 131, 309, 372, 584, 611

Prototyping
  activity, 274
design and evaluation, 206
document, 303, 328, 331
model, 36
  technologies, 62

Q

Quality assurance
  defect-prevention, 711
driven, 565
methodology, 433
metrics, 570
paradigm, 44, 49, 52, 55, 69, 72–73, 76, 88, 94, 103–105, 107, 109, 112, 234, 433–464, 578, 605, 608, 611, 613, 615, 717
principles, 55, 279, 311, 328, 409, 463
process, 434, 436, 463
reports, 570
requirements, 353
results, 434, 436
standards, 92, 97
strategy, 260, 261, 270, 279, 311, 314, 315, 337, 439–460
technique, 123–125, 133, 437, 438
visibility, 433
Rapid prototyping, 62, 231
Rational unified process (RUP), 29, 39, 42, 43, 57, 272, 593–595, 618, 711
Re-engineering, 138, 565
Realization, 200, 227, 598
Record, 28, 243, 253, 532, 533, 536–539, 571, 574
Relationship, 20, 70, 97, 98, 138, 190, 194, 280, 282, 318, 323, 389, 478, 518, 568, 569, 611, 618, 649, 656, 658, 659, 662, 701
Requirement
analysis, 31, 33, 34, 273
conflicts, 50, 220, 611, 636
decomposition, 132, 231, 286, 294, 303, 317, 445
documents, 303–306
elicitation/gathering, 31, 178, 182
engineering, 62, 273–311
management, 265
modeling, 31
priority, 132, 341, 631
traceability, 208, 238–240, 255, 709
validation and verification, 3, 12, 18, 51, 238, 240, 242, 254, 255, 304, 604, 612
Resources, 30, 34, 38, 47, 49, 57, 76, 83, 111, 373, 391, 473, 521, 580, 582, 591, 593, 602, 610, 617, 700, 719
Retirement, 434, 435, 438, 439, 464, 528, 592, 717
Reusable components, 229, 259, 594
Reverse engineering, 8, 110, 137, 138, 155, 179, 207, 271, 277, 340, 348, 476, 558, 560, 579, 584, 613, 616, 617, 633, 635
Revolutionary solution, 137–138, 202, 259–263, 276–279, 470, 518
Index

Risk analysis, 68, 131, 208, 269, 477, 607
Risk management, 274, 517, 520
Road map, 72, 107, 199, 276, 311, 719
Root cause for software disasters, 67–69
RUP. See Rational unified process (RUP)

S
Scheduling, 34, 36, 75, 517, 520, 599
Scope, 42, 388, 589, 591, 650, 651, 664
Security, 73, 257, 258, 410, 463, 608, 650, 656, 660–661, 663, 716
Self-adaptation principle, 88, 95, 97, 203, 529
Self-documenting, 191–194, 320–321, 598, 600, 601
Self-organization principle, 88, 95, 203, 529
Semi-automated test case design, 156, 217, 390–391
Sensitivity to Change principle, 88, 95, 203
Silver bullet, 75, 106, 107, 270, 361, 557, 597–650, 721
Simultaneous users, 658
Six sigma, 437
Smalltalk, 64
Software architectures, 31, 137, 138, 178, 227, 313, 598
Software definition
  traditional, 58, 699
  new, 2, 30
Software configuration management, 32
Software deliverables, 438
Software design engineering, 31, 215, 313, 337, 341
Software development methodologies
  test-driven, 52, 71, 100, 104, 257, 612
  defect prevention driven, 52, 123–125, 133, 230, 234, 364–366, 460, 613
Software disasters, 47, 65–69, 374, 434, 436, 607, 714, 716
Software engineering, 1, 61, 80, 92, 103–105, 115, 135, 199, 239, 257, 275, 313, 339, 383, 410, 433, 517, 560, 577, 597, 699
Software inspection, semi-automated, 166
Software lifecycle, 50, 611, 618
Software maintenance paradigm, 44, 54, 56, 73–74, 103–105, 235, 304, 467–487, 578, 609, 614, 717
Software metrics, 694
Software process improvement, (SPI), 55, 63, 104, 585, 587, 607
Software process model, 44, 57, 58, 63, 199, 200, 202, 235, 699, 700, 708
Software project management, 44, 46, 54, 74, 103, 104, 235, 251, 270, 271, 433, 517–519, 523, 528, 529, 609, 614
Software requirements engineering, 31
Software requirements specification, 649–671s
Software testing. See also Testing
  embedded, 381–382, 418
  engineering, 409
  object-oriented, 378–380
  objectives, 406
  performance, 3, 111, 429, 454, 468, 715
  phase, 270, 716
  support, 219
  system, 100
  tasks, 383
testing, 100, 123, 216, 218, 219, 340, 342, 353, 378, 384, 410, 582, 649
theory, 79, 83

T
Technical forum, 76, 110, 223, 617, 628, 630

Technique
assisted test case design, 128, 129, 203, 204, 419
comprehensive software testing, 122–123, 133, 203, 204
defect-prevention driven quality assurance, 123–125, 133, 364–366, 565
holistic and dynamic traceability, 122, 133, 203, 204, 253, 306, 318, 363, 531, 549–554, 602
holistic and intelligent version comparison, 121–122, 134, 222, 233
holistic, virtual, and traceable diagram generation, 117–119, 133, 203, 204, 214, 231
intelligent regression test case selection, 109, 128–130, 134, 203, 204
intelligent version comparison technique, 32, 121–122, 134, 203, 233
synthesis design and incremental growing up, 96, 98, 100, 131–133, 203, 214, 313–324, 336–339, 341–349, 367, 368, 441, 565, 581, 592, 630, 649, 713
test case efficiency analysis, 3, 12, 16, 76, 111, 125–126, 134, 204, 219, 232, 233, 373, 378, 413, 481, 483, 533–537, 573, 617, 633, 635
test case minimization, 3, 12, 17, 30, 76, 109, 111, 125–126, 134, 204, 219, 232, 373, 378, 391–396, 413, 472, 473, 477, 481, 484, 531, 533–536, 538, 571–573, 615, 617, 633, 635, 700
virtual and traceable documentation, 119–121, 134, 203, 204, 231

Test case
analysis, 148, 219, 572
analyser, 572
collection, 636
efficiency, 3, 12, 16, 76, 111, 125–126, 134, 204, 219, 232, 233, 373, 378, 391, 395, 413, 481, 483, 533–537, 573, 617, 633, 635
execution, 128, 237, 254, 387, 400, 405–406, 535
generation, 232, 349, 352
inputs, 373
manager, 373
numbers, 241, 304, 662
parameter, 225, 250
playback, 570–571
selection, 22, 30, 76, 109, 128–130, 134, 203, 204, 219, 405–406, 485, 534, 615, 638, 700
set, 111, 618
window, 246, 478
Test plan, 633, 635
Testability, 2, 3, 76, 115, 203, 229, 234, 270, 469, 475, 477, 521, 664, 699
Testing. See also Software testing
approach, 100, 123–125, 231, 232, 261, 294
black-box, 52, 117, 231, 234, 373, 375, 410, 578, 613, 715, 716
efficiency, 573
engineer, 232–233
load, 377, 381, 714
operations, 550, 582, 602
order, 124, 156, 183, 216, 232, 349
performance, 376–377, 380, 384, 455, 714
phase, 260, 270, 406, 450, 582
plan, 378
process, 111, 216, 218, 271, 361, 382, 617
purpose, 371–373
reliability, 377
resource, 76, 83, 311, 391, 464, 580, 602
requirements, 373, 643
result, 381
stress, 72, 377, 384
support, 219, 232, 349–353
technique, 122–123, 133, 203, 204
tool, 578
white-box, 52, 373–375, 389, 409, 429, 434, 613
web, 380–381
The mythical man-month, 29, 74–76, 97, 200–201, 597–599
Thinking
linear, 44–45, 49, 50, 57, 66, 69, 72, 74–76, 80, 86–88, 91–94, 97,
Index

Nonlinear, 30, 50, 57, 75, 76, 80, 86–88, 91–94, 97, 101, 137, 162, 234, 314, 411, 609, 611, 612


Tool
Adobe, 156, 303
analysis, 422
bar, 567–572, 574
chain, 137, 138, 179
development, 549
freeware, 381
function/capability, 564
integration, 63, 137, 138, 179, 564
modeling, 276, 583
OO-Analyzer, 3, 12, 563, 564, 567, 570, 572
OO-Browser, 3, 5, 155, 563, 564, 567–569, 649, 721
OO-CodeDiff, 3, 22, 563, 564
OO-DefectTracer, 3, 563, 564, 574
OO-Diagrammer, 3, 9, 160, 563, 564, 567, 569, 649, 673–693, 721
OO-MemoryChecker, 574
OO-MiniCase, 3, 15, 563, 564
OO-Test, 3, 559, 563, 564, 567, 568, 573, 649, 721
OO-Validate, 553, 567, 572, 649
Panounit, 216–219, 349, 352, 353, 698
performance, 11
playback, 3, 123, 189, 232, 382–383, 413, 454, 455, 563, 564, 570–572, 582, 617
software engineering, 137, 138, 179
support, 43, 237–239, 518, 523–524, 618
third party, 106, 251, 252, 255, 305, 564, 602, 710
vendor, 94, 97, 98, 101, 702
version comparison, 575, 634, 635

Traceability
backward, 51, 111, 189, 193, 219, 221, 227, 228, 248, 250, 271, 378, 440, 441, 472, 473, 477, 480, 482, 485, 572, 582, 612, 618, 635, 636
code, 447
facility, 117, 208, 231, 238, 239, 711, 716
forward, 31, 227, 247, 304, 440, 477, 572, 612
requirement, 208, 238–240, 255, 305
software, 75, 122, 255, 602, 604

Transaction, 659

U

UML
charts, 138
design, 275, 584
diagrams, 494, 584
notations, 497
Upstream movement, 33, 51, 70, 97, 103, 200, 201, 235, 268, 275, 276, 340, 434, 435, 468, 489, 578, 593, 598, 600, 601, 612
Usability, 44, 159
Use case
analysis, 286, 287
approach, 31, 130, 133, 274, 282, 413, 649
drawbacks, 274
notations, 282
User interface, 303, 382, 482
User manual, 237, 489, 649

V
Viewpoints, 100, 237
Virtual function, 414
Visibility, 2, 3, 68, 73, 75, 76, 238, 270, 433, 469, 475, 477, 503, 509, 510, 521, 602, 664, 699

W
Waterfall model, 32–36, 42, 43, 45, 51, 75, 92, 105, 599, 608, 612, 618
Waterfall model with feedback, 32, 33, 708, 709
Web engineering, 63
Werewolves, 597, 600, 605–609, 628–647
White box testing, 52, 375, 389, 409, 434, 613
Work flow, 206, 267, 268
Work product, 641–643, 649
Workflows, 97, 101

X
XP. See Extreme programming

Y
Y2K, 66