LECTURE NOTES[1]
Software: a fine art

Anthony A. Aaby

An Open Source Text
DRAFT Version: β 1.0
Edited: October 2, 2006

[1]A review of current knowledge, essays, and philosophical observations.

WWC Software Engineering Students

Contents

List of Figures

List of Tables

Before you begin ...

> This is a work in progress. Feedback to aabyan@wwc.edu is appreciated.

This is a collection of formal and informal notes, ideas, and essays on software quality [1].

The questions that drive this work are:

1. What is quality software?
2. How do we design quality software?
3. How do we build quality software?

Quality is a complex and multifaceted concept that may be described from five different perpsectives [33]. The transcendental view sees quality as something that can be recognized but not defined. The user view sees quality as fitness for the user's purpose. The manufacturer sees quality as conformance to a specification. The product view sees quality as an inherent characteristic of the product. The value view sees quality as dependent on what the customer is willing to pay for it.

The software quality characteristics found in the ISO Standard 9126 (Figure 1) provde a partial answer to the first quesition. They are used to organize the discussion of software quality. The second question is one of process. It has to do with the problem, its analysis, and available design elements. The third question is one of craftmanship, of working syle, of fit and finsh. I suggest that the answers to these questions should be articulated in terms of patterns.

The primary focus of this work is on the areas of design, construction, and quality. A secondary focus is on the areas of software engineering process, requirements, and testing.

The Software Engineering Processes

Software Engineering is the discipline of developing and maintaining software systems that behave reliably and efficiently, and are affordable to develop and

[1]It began to take formal form after reading Richard P. Gabriel's proposal for a Master of Fine Arts in Software `http://www.dreamsongs.com/MFASoftware.html`

Operational Qualities	Maintenance Qualities
• Functionality: suitability, accuracy, interoperability, security, compliance	• Maintainability: analyzability, changeability, stability, compliance
• Reliability: maturity, fault tolerance, recoverability compliance	• Portability: adaptability, installability, co-existence, replaceability, compliance
• Usability: understandability, learnability, operability, attractiveness, compliance	
• Efficiency: time behavior, resource utilization, compliance	

Figure 1: **ISO Standard 9126 Software Quality Characteristics**

maintain. *Software engineers* should be able to properly design and properly implement large-scale software systems.

The ten knowledge ares of software engineering and the eight related disciplines are displayed in Figure 2.

The *software engineering process* is concerned with the definition, implementation, assessment, measurement, management, change, and improvement of the software engineering process itself. *Software engineering management* is project management specialized to the needs of software development. It uses metrics appropriate for software engineering, to manage the activities and processes associated with

- Software Requirements
- Software Design
- Software Construction
- Software Testing

Computer systems and software have become central to organizational activities ...

There has been an. expansion of the commanding role of and budget for software ...

... increasingly important to have the ability to effectively acquire and integrate software . Later chapters in this part ...

This text is suitable for use in an undergraduate course in software engineering or for occasional lectures in other courses. It provides a high-level overview and is not intended to be complete as the text is more about a state of mind

Software Engineering Process				
Software maintenance	**Software engineering management**	**Software requirements**		**Configuration management** **Tools and methods** **Quality**
		Software acquisition	**Design** **Construction** **Testing**	
	Installation (and acceptance test), operation, and maintenance			
Releated disciplines: Computer Engineering, Computer Science, Management, Mathematics, Project Management, Quality Managment, Software Ergonomics, Systems Engineering				

Figure 2: **Software Engineering Knowledge Areas**

than a recipe to be followed. The exercises are an integral part of the text. They are designed to encourage students to agressively use abstraction, generalization, and metaphor to discover simularities and essential differences in the development of software. Through the exercises, readers elaborate the text by considering the implications of the concepts and reflecting on their experience. They are designed to get the reader involved in thinking and internalizing the concepts presented in the text. I hope that the readers of this work use it as a starting point for the development of their own personal philosophy of quality software.

The architect Louis Sullivan's famous dictum that form follows function suggests that the aesthetics of an utilitarian object are at least partly determined by the use or purpose of the object. Aesthetics are value judgements regarding the degree with which an object satisfies its purpose.

Software objects must satisfy various operational and maintenance characteristics. Verification and validation designate those activities that are used to determine whether a software object satisfies its operation and maintenance goals. Validation is the process that determines whether the right product was built (satisfies the user). Verification is the process that determines whether the product is built right (conforms to specifications).

Patterns (see Chapter 5) are used to organize solutions to recurring problems. There are several appendicies which provide summaries of patterns.

It is a challenge to organize this material in this text. The material is diverse and contains interrelationships that seem at times to defy organization.

The ideal teaching order may be different from the physical ordering of the chapters.

The first part of the text (Chapters 4, 5, and Appendix D) focuses on architecture, aesthetics, patterns, and quality. The concept of patterns is extended downward from architectural and design patterns to include the constructs in programming languages, programming idioms, data structures, and algorithms. The primary purpose of the patterns chapter is to engage students in formalizing their knowledge of programming so that they have a conscious awareness of the characteristics of software. architecture, aesthetics, quality, and patterns.

The second part of the text (Chapters 2, 6, and 7) focuses on analysis and the conceptual design processes.

The third part of the text (Chapters 8, 9, 10, and 12) focuses on design processes, design elements, and configurations found helpful in creating software with security, reliability, usability, maintainability, and portability quality characteristics.

The fourth part of the text (Chapter 13) is a single chapter on the software construction process.

The fifth part of the text (Chapters 14) focuses on the design issues of specialized architectures. It is a single chapter on the architecture of information systems.

Figure 3: **Domains**

Terminology

erminology needs to be
stently used throughout
ext.

Terminology is problemantic. Figure 3 provides a conceptual view of the domains involved in design. Architecture can refer to a profession, a process of design, a conceptual structure, a physical (or implementation) structure, or even an aesthetic value (as in great architecture). Design can refer to a process (activity), a plan, or an aesthetic value (as in a good design).

To distinguish between these various meanings, I will use the following:

Design elements - named components, interfaces, and connectors. of a design structure either conceptual or phyisical.

Pattern - a formalized description of a design element.

Design structure - may be either a conceptual or implementation plan.

Detailed design - a topological organization of a set of parameterized modules together with the inter-modular relationships.

Conceptual design - a topological organization of a set of parameterized components together with the inter-component relationships.

Design values - aesthetic values, design values, design principles, quality characteristics.

Design process - the activity that produces a design structure.

Architectural style - a vocabulary of design elements, a set of configuration constraints that must be satisfied by any design composed in the architectural style, and a semantic interpretation of the connectors between the design elements.

Notes to author

From [31]

1. Architectural styles

2. Families of programs and frameworks e.g. a partially complete software subsystem which can be extended by appropriately instantiating some specific plug-ins.

Part I

Background & Context

Chapter 1

Organizational & System Complexity

The complexity can be described along three dimensions: differentiation or variety, dynamic properties, and underlying causal mechanisms.

1.1 The Problem Space

Author note: I need to define ontological complexity, lexical complexity, and semantic complexity

Definition 1.1 *Let Σ be a state space. Given an initial state s_i in Σ and a specification σ on states, a* problem *is defined to be the task of finding a state transformation τ such that the state $\tau(s_i) = s_f$ satisfies σ. τ is said to be a* solution *to the problem.*

The state space may be discrete or continuous, static or dynamic. Examples of state spaces include polynomial equations, arrays of numbers, engineering knowledge and design elements etc. A state is an element of the state space and may consists of a configuration of more fundamental elements. Examples include: a polynomial equation pr all quadratic equations, an array of numbers, design elements for an automobile, the design of a specific automobile, etc. A state transformation is a mapping from the state space into the state space. Example include: the quadratic formula for solving quadratic equations, a sorting algorithm, a design methodology for roads, etc. A specification is a set of conditions that must be satisfied by a solution to the problem. Examples include: a method to solve quadratic formulas, a method to sort the elements of an array, a road for automobiles between points a and b.

Measures of complexity may be defined for the state space, for a problem, and for a solution.

Environmental complexity: the complexity of state s_i and specification σ.

An environment may be inherently complex because it is turbulent, hostile, diverse, technologically complex, or restrictive. Environments can be complex

1. **The tree** (recursively hierarchical model) – communication is largely vertical.
2. **The onion** (layered model) – each layer consists of a collection of agents which may communicate with any agent in the adjacent layer; a generalized hierarchical model.

   Let a_j^i designate agent j at layer i, then agent a_j^i communicates with $a_k^{i\pm1}$.
3. **The network** – independent and interdependent communicating modules.

Figure 1.1: **Organizational Structures**

simply because they are voluminous in their nature and demands, thus overwhelming organizational members with information. The external environment consists of suppliers, customers, markets, distribution networks, geographical location, competitors, and the institutions that shape what the organization must respond to. Without familiarity in the state space and commonly occurring problems, this may be difficult to measure.

Problem complexity: the measure of the difficulty of finding a suitable τ.

In engineering this is the complexity of the design task. Without familiarity in the state space and commonly occurring problems, this may be difficult to measure. Some general ideas are sketched out in Section 1.2.

Structural complexity: the complexity of expression of τ.

Contingency theory states that an organization structures itself and behaves in a particular manner as an attempt to fit with its environment. Organizational structure and complexity is a reaction to the complexity of the internal and external environment. An organization attempts to match its structure (complexity) to the differentiation of the environment through differentiation into similar units, differentiation of center/periphery, the differentiation conforming/deviant (official/unofficial, formal/informal), hierarchical differentiation, and functional differentiation.

The internal environment consists of the attributes of personnel, the core processes and technologies of the organization, the locus of authority and control.

While all structures may be described in terms of a graph, there are three general organizational models, tree, onion, and network (for graphs that are neither tree or onion structured) – See Figure 1.1.

A simple way to quantitatively measure organizational complexity is to count the number of different categories that exist for the variable in question. A raw count of the number of categories present does not take into account the fact that the distribution of these counts amongst the various categories may differ. Various entropy-based measures can take these distributional characteristics into account. The more uniformly spread the variation is amongst the different categories, the greater the measured complexity will be.

Organizational complexity can also be observed via differentiation in structure, authority and locus of control, and attributes of personnel, products, and technologies. Organizational complexity could be defined as the amount of variety, or differentiation, in the organization's core processes and technologies, customers and markets, products and product lines, distribution networks, suppliers, or geographical locations. Finally, diversity could exist in the causal processes underlying the work of the organization, leading to organizational complexity.

For software it is some measure of the mental effort required to understand a piece of software often for the purpose of quantifying the maintainability of the software.

Early attemps at measuring complexity focused on cohesion within a unit and coupling between units. *Cohesion* is a measure of how well the elements of an organization or subunit of an organization work together. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion". Organizations with high cohesion are associated with traits robustness, reliability, reusability, and understandability whereas low cohesion is associated traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand. *Coupling* or dependency is the degree to which each organizational subunit relies on each other subunit. Coupling is an ordinal type of measurement and is usually expressed as "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Low coupling means that one unit interacts with another unit through a well-defined and stable interface. Low coupling is a sign of a well structured organization. Coupling and cohesion are usually contrasted – low coupling often correlates with high cohesion, and vice versa.

Structural complexity can be measured using Cyclomatic complexity, Halstead measures, and function point analysis. It includes the following:

- Mathematical complexity: the number and diversity of mathematical operators and operands in τ.
- Cyclomatic complexity: the number of linearly-independent paths through τ.
- Maintainability complexity. Maintainability is the complexity of adapting an existing transformation to solve the new problem. It is measured in terms of the size of the transformation, the size of supporting documentation, and the complexity of the computation and structure of the transformation.

In engineering, it is the complexity of the resulting design. In business it is organizational complexity. There are many different measures for the structural

Complexity Measurement	**Primary Measure of**
Cyclomatic complexity	Structural complexity, measured by the number of paths through a module
Halstead Complexity Measures	Algorithmic complexity, measured by counting operators and operands
Henry and Kafura metric	Coupling between modules (parameters, global variables, calls)
Bowles metrics	Module and system complexity; coupling via parameters and global variables
Troy and Zweben metrics	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier metrics	Modularity of the structure chart

Figure 1.2: **Software Structural Complexity**

complexity of software. Some are found in the Table 1.2. Technical details for Halstead measures, Cyclometric complexity, and a Maintainability index are provided later in this section.

Computational complexity: for an algorithm, it is the space or time requirements required by τ to produce s_f expressed in terms of the size of s_i.

The space and time requirements of an algorithm to solve a problem define its computational complexity. Computer science studies the computational complexity of algorithms. Problems which require exponential time or space resources are said to be *computationally intractable.* Theorem proving is an example of a problem of exponential complexity while proof checking is of linear (polynomial) complexity. Some problems are unsolvable (uncomputable). The "halting problem" is an example of an unsolvable problem. Intractable and unsolvable problems are "solved" using approximate methods.

Halstead Measures

The Halstead measures assess the complexity of the code used to express computation by counting the number of operators and operands.

n_1	=	the number of distinct operators
n_2	=	the number of distinct operands
N_1	=	the total number of operators
N_2	=	the total number of operands

Measure	**Symbol**	**Formula**
Program Length	N	$N = N_1 + N_2$
Program vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N * \log_2 n$
Difficulty	D	$D = n^{1/2} * (N_2/n_2)$
Effort	E	$E = D * V$

Cyclomatic Complexity

Cyclomatic (also McCabe's) complexity measures structural complexity by counting the number of linearly-independent paths through a program module. Cyclomatic complexity (CC) is calculated from a connected graph of the topology of control flow within the module.

$$CC = E - N + p$$

where E = the number of edges in the graph; N = the number of nodes in the graph; and p = the number of connected components.

A common application of cyclomatic complexity is risk analysis.

Cyclomatic Complexity	**Risk Evaluation**
1-10	simple program; little risk
11-20	more complex program; moderate risk
21-50	complex program; high risk
greater than 50	untestable program; very high risk

Maintainability Index

MI =

$$171 - 5.2 * \log(aveV) - 23 * aveV(g\prime) - 16.2 * \ln(aveLOC) + 50 * \sin(\sqrt{(}2.4 * perCM))$$

where aveV = average Halstead Volume V per module; aveV(g') = average extended cyclomatic complexity per module; aveLOC = the average count of lines of code (LOC) per module; and, optionally perCM = average percent of lines of comments per module.

1.2 Problem Complexity

Computational complexity and structural complexity are measures that apply to the solution, i.e. to τ. Structural complexity could also be applied to the

$$\begin{tikzcd}
\text{Well-defined Problem} \arrow[d] & \text{Solution} \\
\text{Analogous standard problem} \arrow[r] & \text{Analogous standard solution} \arrow[u]
\end{tikzcd}$$

Figure 1.3: **Well-defined problems**

initial state or problem domain, s_i, and the specification, σ. Problem complexity is the measure of complexity of the initial state or problem domain, s_i, and the specification, σ.

Author note: Problem domains vary in complexity. There are several issues. One is the ease of query formulation. Another is the ease of the construction of a response to the query. Formalization helps with both issues. Need to look at lexical and semantic complexity.

Often there are competing requirements some of which are political. Competing and political requires arise in the interaction between team members and incompatible requirements of the various stakeholders. Problems with a political dimension require compromise for solution rather than appeal to physical properties.

At the simplest level, organization problems are complex because the people constituting the organization are complex. Individuals vary tremendously in their age, gender, their histories and experiences, and their beliefs and desires. People are capable of wearing many 'hats' within the organization, playing out various roles at different times. People are often 'matrixed' into both a functional and project team category, and most are fully capable of enacting the beliefs and norms associated with the given role at an appropriate time, even though the roles may actually conflict with one another–most people are capable of dealing effectively with such role conflict.

There are three broad categories of problems – well-defined problems, ill-defined problems and wicked problems. These are informal, qualitative descriptions rather than formal, quantitative descriptions.

Well-defined Problems

A well-defined problem is a problem that has a clear and precise definition. The solution is clearly specifiable and is clearly recognizable as a solution. Well-defined problems often have generally known solutions.

- They are solved using standard methods, methods of similar problems, or methods of analogous problems.

- Examples: puzzles, simple games, and lower level mathematics, science, and engineering.

The class of well-defined problems include uncomputable problems and computationally intractable problems where the solution requires more time and/or space than is available. Therefore, an alternative solution must be found which involves the use of heuristics and approximations. The alternate solution may be an example of an ill-defined problem.

Intractable Problem → Approximation → Solution

Figure 1.4: **Intractable problems**

Ill-defined Problem → Inventive/creative solution → Solution

Figure 1.5: **Ill-defined problems**

Ill-defined Problems

If it is not clear from the beginning what the problem is and thus, what a solution is, then the problem is an ill-defined problem. Therefore, finding a solution requires in addition, finding out what the real problem is. Solving and specifying the problem develop in parallel and drive each other. The solutions found are often such that they still could be improved and it is up to the problem solver to decide when enough is enough.

- Ill-defined problems include computationally intractable problems and uncomputable problems which require heuristic or approximate solutions.
- Unexplored problem domains often contain ill-defined problems.
- Ill-defined problems are solved with inventive and/or creative insight.
- Examples: AI

Wicked problems

The concept of "wicked problems" in design was originally proposed by H. J. Rittel and M. M. Webber (1984) in the context of social planning. Wicked problems are similar to ill-defined problems, just much worse. Furthermore solutions are very difficult, if at all, to recognize as such. Wicked problems have vague, incomplete, contradictory and changing requirements. Rittel and Webber pointed out that in solving a wicked problem, the solution of one aspect may reveal another, more complex problem. The following rules define the form of a wicked problem:

- There is no definitive formulation of a wicked problem. The existence of a discrepancy in representing a wicked problem can be explained in numerous ways. The choice of explanation determines the nature of the problem's resolution. Every wicked problem can be considered to be a symptom of another problem.
- Wicked problems do not have an enumerable (or an exhaustively describable) set of potential solutions, nor is there a well-described set of permissible operations that may be incorporated into the plan.

$$\text{Wicked Problem} \rightarrow \text{Inventive insight} \rightarrow \text{Political Solution}$$

Figure 1.6: **Wicked problems**

- Every wicked problem is essentially unique. So every solution to a wicked problem is a "one-shot operation"; because there is no opportunity to learn by trial-and-error, every attempt counts significantly.
- Wicked problems are often "solved" through the ongoing work of a group. Wicked Problem → Ongoing work of a group → Current solution
- requires an inventive/creative solution.
- Wicked problems have no stopping rule. There is no immediate and no ultimate test of a solution to a wicked problem.
- Solutions to wicked problems are not true-or-false, but good-or-bad. There is uncertainty if the offered solution is the best solution or is even a solution.
- The planner (designer) has no right to be wrong.
- Examples: environmental and economic problems. Any problem whose solution requires a group to change its behavior is likely to be a wicked problem.

These three categories of problems make clear that it is easier to find meaningful solutions for some types of problems than others. Most problem solving is done (and taught) from the perspective of a particular domain. So, for example, problems in the various branches of mathematics and the various academic disciplines with their own courses teaching their own problem solving methods. The distribution of statistics classes among academic departments illustrates both the felt need to focus on domain specific problems and the existence of general problem solving methods.

An important hypothesis in AI is that all intelligent problem solving can be characterized as a search process [59]. Problem solving methods (PSMs) are domain-independent reasoning components, which specify patterns of behavior which can be reused across applications.

1.3 Complex Adaptive System (CAS)

A *complex adaptive system* (CAS) is a system in which there is a significant interplay between the system and its environment. A social organization such as a business is an example of a CAS. Because complex systems and software take on many of the characteristics of a CAS, the material in this section applies both to social organizations and large software systems.

Agent An agent is an abstraction of a functional unit, an individual, project team, a division, or entire organization. The function of an agent is to transform (possibly empty) inputs to outputs. Human agents, faced with overwhelming and/or complex decisions, omit, tolerate errors, queue, filter, abstract, use multiple channels, escape, and chunk in order to deal effectively with the complexity.

Stakeholder A stakeholder is an agent which supplies a resource (to the organization) and expects a (different) resource in return.

Connectivity Connectivity is a link between agents through which information and resources flow. The information and resources are used for coordination (communication and synchronization), and control (command and assessment)[1].

Behavior Behavior of the CAS is induced by the simultaneous and parallel actions of agents within the system. The behavior is the result of numerous events occurring over extended periods of time.

Schema Schema are the norms, values, beliefs, and assumptions which are interpretive and behavioral and may be shared or individualistic.

Fitness Fitness – a complex aggregate of both global and local states within the system. Agents behave so as to increase the fitness of the system that they belong to either globally or locally.

Emergence Emergence is the arising of new, unexpected structures, properties, or processes in a self-organizing system.

Organization An organization is a collection of agents and links intentionally assembled to accomplish a common set of goals (scope, quality). The collection of agents may be homogeneous or heterogeneous.

Horizontal differentiation is differentiation by functional area (functional differentiation) and vertical differentiation is control differentiation. Organizational structure includes various mechanisms of coordination (communication and control).

Self-organizing A CAS is self-organizing if new emergent structures, patterns, and properties arise without being externally imposed on the system. Self-organization is usually distributed throughout the system.

[1]Various disciplines use different words for the same functions. Management science uses the words *communication*, *coordination*, and *control* The engineering disciplines use the words *feedback* and *control.* For sequential algorithms, the computing sciences use the phrases *control structures* and *data structures.* For parallel algorithms, they use the words *communication* and *synchronization.*

Control Control (and feedback) exists to ensure that the agents remain aligned to accomplish the goals of the organization.

The role of governance in a social organization is to assure the stakeholders that the operations are reliably effective and efficient in their use of the stakeholder provided resources. Control provides the necessary mechanisms to ensure reliably effective and efficient use of resources. The key governance questions are:

- Is the system effective?
- Is the system efficient in its use of resources?
- How is the system controlled

Control ... ... fault tolerant ... robust ... stabilizing ... self-aware ... Gödelian machine ...

There are four observable parameters (variables) that are necessary for effective control – time, cost, scope, and quality. Cost and time should be minimized while scope and quality should be maximized. External forces (customers and managers) get to pick the values of three of the variables, the production process gets to pick the value of the fourth variable. The customer's preference is the most important.

There are four generic control mechanisms:

1. Centralization – managers make the decision or require their approval of the decision prior to its implementation.
2. Formalization – rules, procedures, and guidelines that act to guide the decision making process.
3. Output control – established set of (measurable) goals and objectives that act as a criterion in the decision making process.
4. Cloning – an established set of shared values and expectations that act to guide the decision making process.

Cloning and Output control are used when clear measurable objectives are available, there are multiple effective means of achieving the goal, decision makers have expertise and access to necessary information, the proper process depends on factors that must be determined at the time of decision, or decision makers have access to knowledge of results which provide feedback necessary to modify future processes and decisions.

Centralization and Formalization (process control) are used when clear measurable objectives are unavailable, one best way exists and is known and there is a stable environment, decision makers lack expertise or necessary information, where an unfavorable outcome is unacceptable, or consistency is important.

A system is a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations. The specifications are the provisions of the contract and are called *assertions* and are expressed as boolean expressions. The specifications of the mutual obligations constitute a contract. The enforcement of the contract is accomplished using the notion of an assertion. An *assertion* is a boolean expression about the state of a software system. Assertions are inserted into the code and are checked during the system's execution. If any assertion evaluates to false, the system is invalid.

There are several kinds of expressions – preconditions, postconditions, invariants, and progress condition. The specification of each component includes a precondition and a postcondition. The *precondition* specifies the obligations that a user of the component must meet before the component may be used. The expression may include the system state and the arguments passed to the component. The failure of the precondition means that the source of a fault is in the user. The *postcondition* specifies the condition that holds after the execution of the component. It is a guarantee of service provided the precondition is satisfied. The expression may include the old system state, the arguments, the new system state, and returned values. The failure of the postcondition means that the source of the fault is in the component. The *invariant* specifies a condition that must hold at any time. Loop invariants hold on entry to a loop and on each iteration of the loop (on entry and exit of a recursive function). Data invariants hold both before and after access to a data structure or object. The failure of the invariant means that a fault may exist in either the component or in the user of the component. The *progress condition* specifies the condition under which a loop or a recursive subprogram makes progress toward termination. Progress conditions are rarely used as assertions but should be explicitly noted during program construction and debugging.

Structure Organizational structure refers to the degree and type of horizontal and vertical differentiation, mechanisms of coordination and control, formalization, and centralization of power. The general organizational structures, tree, onion, and network) are described in Figure 1.1.

The organizational design (including those of divisions and departments) should be optimized to supply products and services to its clients and customers for the type of environment it operates within and the technology it utilizes. The functional organization is usually represented as a hierarchy with the links representing communication and control. Project organization may occur in hierarchical organizations. Communication links for the project group show up as a connected component.

Organizations making use of centralization and formalization are mechanistic in form. Organizations making use of output control and cloning are organic in form.

The two extremes in organizational structure are functional organization and project organization. Functional organizations are organized according to discipline. Coordination occurs through rules and procedures, detailed specifications, shared traditions and meetings. Project organizations are organized around a project and coordinated by a cross-functional team.

Organizational Complexity Organizational complexity is the amount of differentiation that exists within different elements constituting the organization. In discussing the general concept of complexity, Luhmann (1995) states "we will call an interconnected collection of elements complex when, because of imminent constraints in the elements" connective capacity, it is no longer possible at any moment to connect every element with every other element... Complexity in this sense means being forced to select; being forced to select means contingency; and contingency means risk" (p. 25).

Complexity measures include:

- specialization and level of qualification
- variety of differentiation in : core processes and technologies, customers, and markets, products and product lines, distribution networks, suppliers, or geographical locations.

The optimal design of an organizational structure is dependent upon:

- Data Inputs: The kind and quality of information that is input to the structure (gathered from its customers, suppliers and partners)
  - How the company gathers the information
  - How it interacts with each of these constituents
- Data Transformation: How this information flows through the organizational structures
  - Who has access to it and who doesn't
  - How is the information utilized in making decisions
  - How the information is stored for ease of use and analyzed
- Metaphor: Whether both the organizational processes and systems reflect and mirror information flow

The organization responds to complexity by building barriers around its technical core; by smoothing input and output transactions; by planning and predicting; by segmenting itself and/or becoming decentralized and by adopting rules.

As the environment moves from simple to complex, an organization moves from a centralized structure to a decentralized structure, i.e., from direct supervision to standardization of work processes, of outputs, of skills, and finally, to mutual adjustment(See Figure 1.7).

Mintzberg (1993) builds from Thompson's ideas and suggests that the more complex the environment, the more decentralized the organizational structure. He suggests that as the environment moves from simple to complex, and organization will move from direct supervision to standardization of work processes, of outputs, of skills, and finally, mutual adjustment. Mintzberg separates the concept of complexity, having to do with comprehensibility, from that of stability or turbulence. His model states that in a complex but stable environment, the organization will choose standardization of its work processes and outputs. A

Comprehensibility

Dynamics

	Simple – few exceptions	*Complex* – many exceptions
Stable	Routine problems which are easily solved using rational analysis. Example: routine manufacturing. Organizational structure: functional organization which is highly centralized and formalized means of coordination and control.	Problems solveable using well-defined principles and technologies which are applicable to a wide variety of new and different situations. Example: product engineering Organizational structure: traditional manufacturing and product development process using a cross-functional team.
Chaotic	Routine problems requiring interactive empiricism for their solution. Example: specialty work. Organizational structure: cross-functional project organization with formal coordination and control	Problems requiring mutual adjustment and networking with no recognized problem solving methodology. Example: non-routine research, creative production. Organizational structure: decentralized with informal means of coordination and control.
Random	No solution	No solution

Figure 1.7: **Complexity and Organizational Structure**

stable environment allows for standardization, for example, of an organization's quality system according to ISO 9000 guidelines. In a complex and turbulent environment, the organization will resort to mutual adjustment. There are losers and winners in such mutual adjustments, but it is the only means by which to move forward at all.

Organizational Change The value of organizational change is dependent on the difference between its current organization and an ideal organization.

These two dimensions are not necessarily correlated.

1.4 Exercises

1. Determine the Halstead measures, cyclomatic complexity, and maintainability index for a piece of software of your choice.

Chapter 2

Creativity and Innovation

2.1 The System Life Cycle

Developing, assessing, deploying, and operating (DADO) are the standard process phases for creating and maintaining systems and services. The phases may be configured in many forms including a waterfall, a cycle, or interlinked (Figure 2.1). "The DADO model is an ongoing process by which the artifacts being created are measured and understood to improve their operational behavior, and this knowledge is being continuously fed back into the development process in the form of better tuning, new features, or bug/performance fixes." [30]

2.2 Response to Complexity

Human agents, faced with overwhelming and/or complex decisions, omit, tolerate errors, queue, filter, abstract, use multiple channels, escape, and chunk in order to deal effectively with the complexity [27]. People deal with complex problems and decisions in several different ways (March, 1994).

Also found in Chapter 1

1. Divide and conquer: First, decisions can be broken down into their constituent parts and treated sequentially.
2. Abstract: Second, secondary and tertiary features of the decision may be ignored, allowing people to focus on only a few select issues.
3. Analogy: Third, people may draw from their experience, seeking solutions from previous problems, and minimizing their effort searching for entirely new solutions.
4. Approximate: Fourth, solutions can be chosen which satisfy, rather than optimize, the given situation, thus reducing the search effort.
5. Insight: Fifth, people make assumptions about data that is not present, rather than going through the effort of questioning those assumptions, and/or seeking confirmation of their assumptions.

Waterfall Development Process:

$$\begin{tikzcd}
\text{Develop} \arrow[d] \\
\text{Assess} \arrow[d] \\
\text{Deploy} \arrow[d] \\
\text{Operate}
\end{tikzcd}$$

Cyclic Development Process:

$$\begin{tikzcd}
 & \text{Develop} \arrow[dr] & \\
\text{Assess} \arrow[ur] & & \text{Deploy} \arrow[dl] \\
 & \text{Operate} \arrow[ul] &
\end{tikzcd}$$

Interlinked Development Process:

$$\begin{tikzcd}
 & \text{Develop} \arrow[dr, leftrightarrow] \arrow[dd, leftrightarrow] & \\
\text{Assess} \arrow[ur, leftrightarrow] \arrow[rr, leftrightarrow] & & \text{Deploy} \arrow[dl, leftrightarrow] \\
 & \text{Operate} \arrow[ul, leftrightarrow] &
\end{tikzcd}$$

Figure 2.1: **DADO-Based Development Models**

6. Metaphor, Use cases, Scenarios: Finally, people use stories and narrative procedures to convey information about complex situations that could not be readily assimilated via other modes of communication

At the organizational level, an organization will respond to complexity by building barriers around its technical core; by smoothing input and output transactions; by planning and predicting; by segmenting itself and/or becoming decentralized; and by adopting rules.

Thompson (1967) discusses theory concerning how an organization will respond to environmental complexity.

1. Encapsulate complexity and provide a user friendly interface: First, following the norms of rationality, an organization is more likely to build barriers around its "technical core" if the environment is complex, surrounding it with input and output components.

2. Provide a user friendly interface: Second, an organization will attempt to reduce the complexity of its environment by smoothing input and output transactions.

3. Anticipate change: Third, an organization will attempt to predict the amount of uncertainty and fluctuation in a complex environment.

4. Partition: Fourth, an organization will segment itself in order to make its sub-environments relatively homogeneous.

5. Fifth, organizations will adopt rules as a means to coordinate the less-complex aspects of its internal and external environments.

The more complex the environment, the more decentralized the organizational structure. Mintzberg separates the concept of complexity, having to do with comprehensibility, from that of stability or turbulence. His model states that in a complex but stable environment, the organization will choose standardization of its work processes and outputs. Further still, an organization may simply standardize work skills.

As the environment moves from simple to complex, an organization moves from a centralized structure to a decentralized structure, i.e., from direct supervision to standardization of work processes, of outputs, of skills, and finally, to mutual adjustment(Figure 2.2) .

2.3 Creativity

The material in this section is from [73, 74].

Creativity: The ability to generate new ideas or synthesize new solutions in the absence of prior examples or paradigms. While creativity is often required in problem solving, it is especially necessary to solve *wicked problems.*

		Comprehensibility	
		Simple	*Complex*
Environment	*Stable*	Single thread of control	Standardization
	Turbulent	Partitioning	Distributed processes

Figure 2.2: **Complexity and Organization**

Levels of Creativity

- Rare revolutionary events (breakthroughs and paradigm-shifting innovations)
- Normal science - useful evolutionary contributions that refine and apply existing paradigms
- Impromptu or personal creativity

Three Perspectives on Creativity

1. Inspirationalists
   - emphasis: the remarkable "Aha!" moments (luck favors the prepared mind)
   - techniques
     - brainstorming
     - free association
     - lateral thinking
     - divergence

This often takes the form of advice offered to students working on "hard problems" which is to become familiar with all of the properties of the problem domain and then go to sleep and let the subconscious work toward a solution.

2. Structuralists
   - emphasis - exhaustive exploration of possible solutions
   - techniques
     - studying of previous work
     - methodical techniques such as Polya's four steps
       (a) Understanding the problem
       (b) Devising a plan
       (c) Carrying out the plan
       (d) Looking back
3. Situationalists

- emphasis - social and intellectual context as the key part of the creative process
- techniques/components of creativity
  (a) domain: a set of symbols, rules, and procedures
  (b) field which determines whether an new idea, performance, or product should be included in the domain
  (c) creative individual who sees a new idea or pattern

2.4 Innovation

Areas for Innovation

1. Platform innovation – the emergence of an entirely new scientific principle to solve the problem. Technology (platform) is a means of solving a problem based on a distinct platform or scientific principle.
2. Component innovation – the use of new parts or materials within the same technological platform.
3. Design innovation – a reconfiguration of the linkages and layout of components within the same technology.

Sustaining and Disruptive Innovation

Christensen [17] defines sustaining innovation (SI) as improved performance of existing products. It is incremental and safe, predictable, conservative, and risk averse. And he defines disruptive innovation (DI) as that which creates new and conflicting value propositions forcing competitors and consumers to make reactive decisions.

There are two distinct types of disruptive innovations. The first type creates a new market by targeting non-consumers. The second competes in the low end of an established market. Disruptive innovations

1. emerge out of insignificant markets
2. initially under-perform against established products
3. few customer's want or think they need the product
4. lead to reduction in profit margins of older products.

and are subject to four sets of variables that affect the probability of success:

1. Taking root in disruption
2. The necessary scope to succeed
3. Leveraging the right capabilities
4. Disrupting competitors, not customers.

Taking root in disruption

Disruptive innovations don't initially perform well enough to be sold or used successfully in mainstream markets. Their key attributes are often

- simplicity
- convenience
- low cost

Criteria for disruption

1. Does the innovation enable less-skilled or less-wealthy customers to do for themselves things that only wealthy or skilled intermediaries could previously do?
2. Does the innovation target customers at the low end of the market who don't need all the functionality of current products? And does the business model enable the disruptive innovator to earn attractive returns at discount prices unattractive to the incumbents?

The scope needed to succeed

- Highly integrated: use interdependent and proprietary architecture to achieve highest efficiency and performance.
- Non-integrated: use modular architecture to achieve simplicity, convenience, speed to market, and customization.

Leverage the right capabilities

Excessive risk-aversion stifles innovation. Answers to the following three questions help to determine the degree of risk-aversion and innovative limits of an organization:

1. Do I have the resources to succeed?
2. Will my organization's processes facilitate success in this new effort?
3. Will my organization's values allow employees to prioritize this innovation, given their other responsibilities.

Innovation requires resources. Resources for innovation include:

1. Managers open to innovation. The management techniques for stable, mature markets are different from those required in disruptive ventures.
2. Money. There are two misconceptions in managing money

   (a) Deep pockets are an advantage.

      No, too much cash allows following a flawed strategy too long while barely enough forces the managers to adapt to the desires of actual customers and forces them to uncover a viable strategy more quickly.

   (b) Large losses will occur for sustained periods prior to reaping the huge upside.

      No, innovators should be patient about the venture's size but impatient for profits. The mandate to be profitable forces the venture to zero in on a valid strategy.

3. Technology.
4. Technical talent.
5. Flexible Process. Standard processes are often inflexible as is they are often designed to get the same job done reliably, again and again.

Disrupt competitors, not customers

Minimize the need for customers to reorder their lives, help them do things more simply and conveniently.

Organizing for innovation

Charles Handy, a British management theorist, identified four kinds of organizational units.

1. Process – its about control through standardization
2. Project – its about the team
3. Networking – its about communication
4. Vision – its about supporting individuals with unique skills

and he suggests that innovation comes from those organizational units that emphasize networking and vision.

Visionary Leadership and Innovative Culture

Tellis[1] criticizes Christensen and suggests that the forces of change lie in visionary leadership and the will of the leader to execute that vison. Long-term market leaders focus on future emerging mass markets, innovate relentlessly to cater to that emerging market, are paranoid about compeditors getting their first, and are willing to cannibalize their current assets to realize that future potential.

[1]Gerard J. Tellis *Disruptive Technology or Visionary Leadership?*

Other internal aspects of an organization that facilitate innovation are: organizational competence, cognitive framing, strategic orientation, and customer orientation.

The following principles[2] should be followed:

- Pursue the opportunity with an emergent strategy plan and an iterative process that incorporates learning and rapid adjustment.
  - *Focus on immediate action.* Determine and pusue information you can gather quickly that will increase your confidence that you are going in the right direction.
  - *Focus on assumptions.* Explicitly testing key assumptions will help reduce risk by finding the right strategy more efficiently and with more targeted investment.
  - *Have defined milestones and check in points* to facilitate the interative process.
  - *Iterate* back and revisit your strategy as you learn more. Some assumptions that seemed to be important will become less important; new assumptions will emerge that need to be tested. If you learn that your strategy appears less attractive than you thought, consider how to shape it in directions that will increase your changces of success. Don't be afraid to kill a fundamentally flawed strategy.
- Remember two principles:
  1. Be impatient for profit, patent for growth.
  2. Scarcity is the entrepreneur's advantage.

  That is, spend as little as possible to test key assumptions and don't prematurely lock into an approach that is probably wrong.

These principles are consistent with the principles and practices of sound risk management and prevent excessive risk-aversion from stifling innovation.

Design Principles for New Services

Kalevi Kilkki (in "Sensible design principles for new networks and services" *First Monday*, volume 10, number 1 (January 2005). `http://firstmonday.ord/issues/issue10_1/kilkki/index.html`) proposes that there are three reasons for an undesirable outcome and three parts to a good design.

The main obstacles preventing practical design solutions are

- a failure to focus on fundamental human needs,
- a tendency to add excessive capabilities, and

[2]from `www.Innosight.com` Disruptive Innovation: an introduction

- a failure to leverage selfish behavior toward the common good.

A commitment to the following three rules is necessary to avoid these obstacles.

- When the value of a new application or service is assessed, we concentrate on practical uses that are likely to become everyday routines for the majority of customers, instead of seeking special cases with the utmost attractiveness. Network technology should be able to efficiently support those routine services.
- The development of a new technology must be based on well-defined, carefully selected core principles, and those principles must always be honored regardless of all attractions to deviate from these principles. Some of the core principles must be able to limit the innate trend toward extreme complexity.
- When a current technology is developed further, we must look for methods and mechanisms that serve both the interest of key stakeholders and the common good. The only way to carry out this is to keenly take into account experiences in real networks during the development of a given technology.

2.5 General Problem Solving Strategies

Polya on Problem Solving

G. Polya[64] proposed a four step plan for solving problems.

1. UNDERSTAND THE PROBLEM
    - What is the unknown? What are the data? What is the condition?
    - Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?
    - Draw a figure. Introduce suitable notation.
    - Separate the various parts of the condition. Can you write them down?
2. DEVISE A PLAN
    - Find the connection between the data and the unknown. You may be obliged to consider auxiliary problems if an immediate connection cannot be found. You should obtain eventually a plan of the solution.
    - Have you seen it before? Or have you seen the same problem in a slightly different form?

- Do you know a related problem? Do you know a theorem that could be useful?
- Look at the unknown! And try to think of a familiar problem having the same or a similar unknown.
- Here is a problem related to yours and solved before. Could you use it? Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible?
- Could you restate the problem? Could you restate it still differently? Go back to definitions.
- If you cannot solve the proposed problem try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Could you derive something useful from the data? Could you think of other data appropriate to determine the unknown? Could you change the unknown or data, or both if necessary, so that the new unknown and the new data are nearer to each other?
- Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

3. CARRY OUT THE PLAN
   - Carry out your plan of the solution and check each step. Can you see clearly that the step is correct? Can you prove that it is correct?

4. LOOK BACK
   - Examine the solution obtained.
   - Can you check the result? Can you check the argument?
   - Can you derive the solution differently? Can you see it at a glance?
   - Can you use the result, or the method, for some other problem?

The Genex Framework

Shneiderman's Genex Framework[73] is a generalization of Polya's four step plan[64] for problem solving. It has four phases each with one or more activities.

1. *Collect:* learn from previous works stored in libraries, the Web, etc
   - *Searching* and browsing digital libraries
   - *Visualizing* data and processes

2. *Relate:* consult with peers and mentors at early, middle, and late stages

- *Consulting* with peers and mentors

3. *Create:* explore, compose, and evaluate possible solutions

    - *Thinking* by free associations
    - *Exploring* solutions - what if tools
    - *Composing* artifacts and performances
    - *Reviewing* and replaying session histories

4. *Donate:* disseminate the results and contribute to the libraries

    - *Disseminating* results

An alternative to the well-defined, ill-defined, wicked classification of problems is whether a problem is solvable using reductionist techniques or whether it requires holistic techniques.

Reductionist approach to problem solving

The reductionist approach to problem solving involves the top-down recursive decomposition of a problem into independent subproblems. It is based on the following five assumptions:

1. *Problems and objectives can be identified.* The model begins with a needs assessment that leads to a problem statement. Objectives are developed that drive processes expressly designed to solve the identified problem. Success or failure is measured against these objectives.

2. *The defined subproblems can be solved by people with appropriate expertise.* Domain experts and software engineers work together to produce a solution.

3. *Problems and objectives are put into operational or quantitative terms and solutions are modeled to achieve optimal performance.* To be rational and measurable, problems and objectives are assigned numerical indicators so progress can be monitored. Individual components are adjusted to conform.

4. *Improvements come from the implementation of solutions.* Once the problem has been identified and solutions modeled, the best model is executed to solve the problem.

5. *The analyst is independent of the problem, a consultant who makes recommendations to a client.* To be fully objective and impartial, analysts remain apolitical and scientific in providing advice and counsel to clientele.

Problem solving based on these assumptions is most successfully employed in addressing problems that may be decomposed into independent parts and where desired outcomes are agreed on – well defined problems. This approach is less successful when applied to solving ill-defined or wicked problems with interdependent subproblems and with contradictory and/or competing goals.

Methods:

- Divide-and-conquer & Stepwise Refinement
- Top-down vs. Bottom-up
- Data Abstraction and Information Hiding
- Patterns and Pattern Languages

By reducing a multifaceted issues into measurable bits, users of the reductionist model ignore critical characteristics of the whole, called emergent properties. If solution to a subproblem creates a new problem or interferes with the solution to another subproblem, a holistic approach may be necessary. Holistic methods are an alternative way of viewing and approaching paradoxical issues. Rather than divide problems into smaller and smaller chunks to be examined by experts, the holistic paradigm requires examining problems in their entirety. It's based on a very different set of assumptions. Reductionists are pigeon holers while holistic solvers are linkers.

Holistic approach to problem solving

The holistic approach to problem solving involves the search for an acceptable solution as an emergent property of interdependent subproblems with competing or contradictory constraints. It is based on five assumptions:

1. *Problems and solutions are constructs of the mind.* Problems don't exist independently from those who define them. Some problems are so complex and so far-reaching that their complete solution is unrealistic and partial solutions must be employed.
2. *People have different views of the same situation.* A problematic situation is viewed and defined differently by the people affected.
3. *People disagree on what's the actual state and desired state.* Since there's no agreement on what the problem is, it follows that there won't be agreement on the solution.
4. *Improvements result from discussion and debate.* A solution is an ongoing work of the group. In effect, the impacted parties learn together.
5. *The analyst becomes part of the problem situation.* The analyst becomes part of the problem situation and uses a variety of tools to learn along with the impacted parties in formulating a solution.

Problem solving based on these assumptions is most successfully employed in addressing ill-defined and wicked problems with interdependent subproblems and where desired outcomes are vague, incomplete, competing or contradictory, and changing. An acceptable solution is often an emergent property. This approach is unwieldy when applied to solving well-defined problems where there is agreement on the desired outcomes.

Methods: Holistic problem solving often requires negotiation, compromise and acceptance of partial solutions.

References

- Buckingham Shum, S. `http://kmi.open.ac.uk/~simonb/` (1996). Negotiating the Construction of Organizational Memory Using Hypermedia Argument Spaces, *Workshop on Knowledge Media for Improving Organizational Expertise, 1st International Conference on Practical Aspects of Knowledge Management*, Basel, Switzerland, 30-31 October 1996. Available: `http://kmi.open.ac.uk/~simonb/org-knowledge/pakm96/negotiating/negotiating.html`,

- A. MacLean, V. Bellotti, and N. Hammond (1996). Graphical Argumentation and Design Cognition. Available in full: `http://kmi.open.ac.uk/~simonb/csca/graph-arg-design/graph-arg-whole.html`

  - The sections that take up the issue of wicked problems are sections 4 and 5, which are also available in html format at `http://kmi.open.ac.uk/~simonb/csca/graph-arg-design/graph-arg-sec4.html` and `http://kmi.open.ac.uk/~simonb/csca/graph-arg-design/graph-arg-sec5.html`
  - An updated version of this paper from February, 1997 is available as Technical Report # KMI-TR-25 (either in postscript or pdf format) at `http://kmi.open.ac.uk/kmi-abstracts/kmi-tr-25-abstract.html`. This is to appear in the publication *Human-Computer Interaction.*

- Buckingham Shum, S. `http://kmi.open.ac.uk/~simonb/` (1997). Representing Hard-to-Formalize, Contextualized, Multidisciplinary, Organizational Knowledge. AAAI Spring Symposium on Artificial Intelligence in Knowledge Management (Mar. 24-26, 1997), Stanford University, Palo Alto, CA, AAAI Press. Available:

  `http://kmi.open.ac.uk/~simonb/org-knowledge/aikm97/sbs-paper2.html` `http://ksi.cpsc.ucalgary.ca/AIKM97/`

- Budgen, David (1994). *Software Design.* Reading, Massachusetts: Addison-Wesley. ISBN 0-201-54403-2.

- Newland, Paul (1990). *Understanding Designer's Knowledge Acquisition Processes: A Potential for Enhancing Information Transfer*, Unpublished

Ph.D Thesis, University of Portsmouth, March 1990, Abstract available: `http://www.envf.port.ac.uk/newmedia/paul/PhD/PhD.html`

  – (includes a nice quote from Rittel and Webber's original article)

- Ofluoglu, S. (1995). *A Model for Organizing Architectural Design Information.* Masters Thesis, The Graduate School, College of Art and Architecture, Pennsylvania State University. Available `http://www.caad.ed.ac.uk/~salih/thesis/thesis.htm`

  – In Chapter 2, on Information and Architecture `http://www.caad.ed.ac.uk/~salih/thesis/chapter2.htm`, Ofluoglu discusses design as a wicked problem.

- Rittel, H. J., and M. M. Webber (1984). "Planning problems are wicked problems

2.6 Exercises

1. Discuss the relationship between wicked problems,innovation, and agile development methodologies such as Extreme Programming.
2. Read First Monday ... open source ... innovation
3. Explain why artificial intelligence is ill-defined but not a wicked problem.
4. Create patterns for
   (a) Wicked problems
   (b) Reductionist problem solving.
   (c) Holistic problem solving
   (d) Inspirationistic creativity
   (e) Structural creativity
   (f) Situational creativity
   (g) Genex
   (h) Disruptive innovation
   (i) Sustaining innovation
5. Can you identify different types of logics for each problem type?
6. TRIZ and Axiomatic Design Theory are formal design methodologies. Report on their use in software design.
7. Explore "Non-linear thinking" and determine its suitability as a methodology for innovation.

Chapter 3

People

Humans, faced with overwhelming and/or complex decisions, omit, tolerate errors, queue, filter, abstract, use multiple channels, escape, and chunk in order to deal with the complexity[27].

Humans possess individuality and the power to think and to do. However, a large part of human activity occurs at the sublinguistic level. Learning often occurs through exposure to examples rather than abstract communication. Intuition plays a large role in human behavior and the motivating values are implicit rather than explicit. Individual differences emerge in part from different experiences. Communication is metaphorical and is often misunderstood. People often reveal their preferences through reaction to objects rather than through explicit and explicit communication.

3.1 The Nature of Stakeholders

include material on rational choice theory and wise crowds found in /web/Articles/CEthics

A stakeholder is anyone that supplies resources for a project/product and expects something in return. The stakeholder expects assurance that the activity will be reliably effective and efficient in the use of the resource i.e., that the stakeholder will receive a reasonable return on the investment.

The primary stakeholder is the client. A client's knowledge may be tacit (clients may not be able to accurately describe what they do). The users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. A client may not know what is technically feasible. Clients may change their minds once they see the possibilities more clearly. Therefore, it may be impossible to say with certainty what the requirements are, let alone whether or not they are met, until the system is actually in place and running. These discoveries made during the later phases may force retrofitting requirements. Often clients often do not appreciate the complexity inherent in software engineering, and the impact of changed requirements. In addition, there may be many social, political, legal, financial, and/or psychological requirements which affect system design.

The Nature of Users

Users engage in three basic activities, knowing, changing, managing. Users differ in perceptual and psychomotor abilities. There are several types of users, first time, intermittent, and expert. Users do not like to read. Users forget things. Users get into trouble (make mistakes, get lost). Users want to maximize their efficiency and effectiveness. The values of the users drive the feature set and the user interface design activities.

The Nature of Programmers

Programmers use software tools to perform their tasks. Therefore, they share many characteristics with users. They engage in three basic activities, knowing, changing, managing. They differ in perceptual and psychomotor abilities. They do not like to read. They forget things. They get into trouble (make mistakes, get lost, forget things). They want to maximize their efficiency and effectiveness. The values of programmers drive the feature set of their work environment and the activities.

3.2 Management

Different people will be motivated by different things, and that a single individual is likely to be motivated by different things at different times. The complexity of human nature demands that managers be flexible, adaptive, experimenting, and learning.

3.3 The Cognitive Dimension of Programming and Software Applications

Thomas Green has developed a framework [34] to facilitate discussions about programming environments and the factors that influence a programmer's effectiveness. He calls this framework the "Cognitive Dimensions Framework". This framework consists of 14 relatively independent features that characterize programming environments. These 14 dimensions may be useful in discussions about user's effectiveness in application environments as well.

Need to apply the CDs to usability, maintainability, and construction

Viscosity is resistance to change. How hard is it to introduce small changes to the program?

Hidden Dependencies are the important links between entities that are not visible. A hidden dependency results when one part of a program changes another part in a way that is not overtly apparent in the program text.

Visibility and Juxtaposibility is the ability to view components and relationships easily. Is required material directly accessible without cognitive work?

Imposed Look-ahead Constraints is an imposed order on doing things. Does the programming environment force the user to make a decision before the information is available?

Secondary Notation is extra information in means other than program syntax. Can statements or primitive elements be grouped or otherwise denoted to provide extra information to the programmer? For example, white space around blocks of code can show related statements.

Closeness of Mapping is the mapping between the representation and the domain. How close is the mapping between the end-user visible problem domain and the programming domain?

Progressive Evaluation is the ability to check while incomplete. Can parts of the program be tested before the entire program is written?

Hard Mental Operations are operations that tax the working memory of the programmer. Does the programming language lead one to tortured combinations of primitive operations that are painful to de-tangle?

Diffuseness/Terseness is a measure of the succinctness of language.

Abstraction Gradient is the amount of abstraction required, versus the amount possible.

Role-expressiveness is whether the purpose of a component is readily inferred.

Error-proneness refers to syntactical structures that provoke slips. Is it easy to make cognitive slips and introduce errors into the code?

Perceptual mapping is conveying important meanings by position, size, color, etc.

Consistency is expressing similar semantics in similar syntax. Can you infer one part of a language from another part?

Part II

Software a Fine Art

Chapter 4

Software Architecture

4.1 The Nature of Fine Art

Fine art is created for aesthetic purposes and is valued for beauty (delighting the mind) or expressiveness (economy of expression). It is about communication that is focused, economical, and insightful. It reveals unexpected connections. It may be motivated by aesthetic, philosophical, technical, or social problems that artists have tried to solve. Fine art is not alway accessible to a broad cross section of society. Its appreciation may require careful study and thought.

Software exhibits all of these characteristics and more. Games are vehicle for personal expression. The Internet and the open source software movement are political and philosophical expression exhibiting both the values of cooperation and self-sufficiency. In these and many other ways, the software engineer is revealed as a philosopher, a critic, a moralist, a dreamer, a prophet, recorder of his time, and entertainer.

The aesthetics of software shares much with the aesthetics of mathematics and scientific theories. See `http://www.cut-the-knot.com/manifesto`.

> "... mathematics is capable of an artistic excellence ... because it gives in absolute perfection that combination, characteristic of great art, of godlike freedom, with the sense of inevitable destiny; because in fact, it constructs an ideal world where everything is perfect but true. "
> – Bertrand Russell (*Autobiography*, George Allen and Unwin Ltd. 1967, v1, p158.)

> "Mathematics ... possesses supreme beauty – a beauty cold and austere, like that of sculpture, ... sublimely pure, and capable of a stern perfection such as only the greatest art can show."
> – Bertrand Russell (*The Study of Mathematics*)

> "Mathematics ... is the one means by which we rise to a complete self-consciousness. ... it is an art; by informing us of the nature of

1. Developer: Code
2. Designer: Architectural design
3. User: Application & user interface
4. Society: The social implications and consequences of software.

Figure 4.1: **Architectural Levels**

our own minds it informs us of much that depends on our minds."
– J. W. N. Sullivan *Aspects of Science* 1925

"... mathematics is a ever-evolving language for the study of structure and pattern. Grounded in and renewed by physical reality, mathematics rises though sheer intellectual curiosity to levels of abstraction and generality where unexpected, beautiful, and often extremely useful connections and patterns emerge. Mathematics is the home of both abstract thought and the laws of nature. It is at once pure logic and creative art."
– Lawrence University catalog, Cited in Essays in Humanistic Mathematics, Alvin White, ed, MAA, 1993.

There are several values that are of particular interest and value to those in the computing disciplines. The key values are regularities, patterns, abstractions and generalizations that simplify the enormous complexity of software.

4.2 Aesthetics and Style

Design is a set of aesthetic values[1] that permit the evaluation or comparison of various designs. It applies to software at several levels and affects several different types of users. The perception of software as a fine art may occur to those who read the code at the level of an algorithm, data structure, or a whole program. Or it may occur to those who use the software (Figure 4.1).

Design as an aesthetic refers to some of the attributes of the product of the design process. The focus of this document is on the those qualities in a design that produce a preference for one design over another in objects that

[1]Aesthetic values are usually implicit and intuitive. The communication of aesthethic values is often difficult as an appropriate language may be unavailable. This is illustrated by the well known comment by Justice Potter Stewart in Jacobellis v. Ohio, 378 U.S. 184, 197 (1964). He tried to explain "hard-core" pornography, or what is obscene, by saying, "I shall not today attempt further to define the kinds of material I understand to be embraced . . . [b]ut I know it when I see it . . . ". Consequently, requirements elicitation is often quite challenging when the values are implicit and the project complex.

are intended to persist over time i.e., as software spends most of its life time in maintenance mode, software design that facilitates its own evolution.

- Architectural Style
- Programming/Language Style
  - Imperative and Procedural Style
    - instructions
    - data types
    - program structure
  - Functional Style
  - Logic Style
  - Object-Oriented Style
- Execution Styles: Batch, Interactive, Reactive
- Window Interface Styles: Mac, Windows, Gnome, KDE ...
- Coding Style

Examples

- Elegance of Recursion over iteration
  - The factorial function.

    f (n) = if (n = 0) then 1 else n*f(n-1)

    vs

    f := 1; while n > 0 do { f := f*n; n:= n - 1 }
  - The Tower of Hanoi Problem.

    Problem: There are three pins. There are n disks on pin A stacked in order of decreasing diameter. Move the n disks from pin A to pin C respecting the following constraints.
    - No disk may be placed on top of a smaller disk.
    - Only one disk at a time may be not on a pin.

    Idea: Move the largest disk to pin C by first moving n-1 disks from pin A to pin B using pin C as temporary storage. Then move the disks on pin B to pin C using pin A as temporary storage.

    Code: hanoi(n, A, B, C)
    ```
    {
      if n=1 then A:=0; C:=C+1
      else hanoi(n-1, A, C, B); A:=0; C:=C+1; hanoi(n-1, B, A,
    C); }
    ```

- Architecture: Client-Server, Unix philosophy
- Application/user interface: Apple MacIntosh
- Social change: Web

4.3 The Nature of Software

Software is flexible. Software is an executable specification of a computation. *Software is expressive.* All computable functions may be expressed in software. Complex event driven systems may be expressed in software. *Software is huge.* An operating system may consist of millions of lines of code. *Software is complex.* Software has little regularity or recognizable components found in other complex systems and there are exponentially many paths through the code and changes in one part of the code may have unintended consequences in other equally remote sections of the code. *Software is cheap.* Manufacturing cost is zero, development cost is everything. Thus, the first copy is the engineering prototype, the production prototype and the finished product. *Software is never finished.* The changing requirements and ease of modification permits the maintenance phase to dominate a software product's life cycle, i.e., the maintenance phase consists of on going design and implementation cycles. *Software is easily modified.* It is natural to use an iterative development process combining requirements elicitation with design and implementation and use the emerging implementation to uncover errors in design and in the requirements. *Software is communication.* Communication with a machine (HCI) but also communication between the client, the software architect, the software engineer, and the coder. Software must be readable in order to be evolvable.

The creation of software is a team effort. Team members may include, team leader, development manager, planning manager, quality manager, process manager, security specialist, support manager, developers and Others specialized team members may be required such as requirements engineer, software architect, security specialist, user-interface specialist, usability engineer, and configuration manager.

Given a problem statement, the *software architect* elicits the set of functional and non-functional requirements and produces an architectural design that satisfies the functional requirements. Software architects attempt to minimize the constraints and maximize the societal benefits and the economic value of the resulting design. The best examples of design become codified for re-use by other architects. Software architects may utilize frameworks, building tools, and architecture design languages for the specification, validation, prototyping and construction of a software system. The complete work product of a software architecture is the documentation and supporting models that describe the aesthetics, functions, performance, structures, components, impacts, and costs of a solution.

Given a set of functional and non-functional requirements and an architectural design, the *software engineer* produces an implementation design that satisfies the non-functional requirements.

Given an implementation design, the *software developer/programmer/coder* produces an implementation.

See Appendix 3 for the nature of users and programmers. See Chapter 2 on the nature of problems.

4.4 Software Quality Characteristics

A program is a specification of a computation. A computational thing is something that displays the execution of a program[39]. The process of design for computational things must take into account the interplay between temporal and spatial structures. Time is a central form element in computational things just as it is in music. Four questions for the existential definitions of computational things.

1. In what way and in what sense does it express the execution of the program?
2. What determines what is to be displayed?
3. What initiates the execution of the program?
4. What defines the given program?

The ISO standard 9126 lists six broad software quality characteristics and twenty-seven additional sub-characteristics (see Figure 4.3).

Computer science focuses on two of the characteristics, functionality and efficiency. All of the quality characteristics are important in the software engineering of real world products.

There are three classes of users who benefit from high quality software. Those who read the code, those who interact with the program in execution, and those who rely on the software to perform a mission critical task. Regardless of whether software is written in the course of doing computer science or as a software product, code is written to be read by two audiences, the machine and those who read the code as part of their professional life (computer scientists or code maintainers).

rewrite this paragraph

The purpose of this work is to encourage students in undergraduate computer science and software engineering courses to recognize software as a medium of professional communication and stimulate them to improve the quality of the code that they write. They should develop an aesthetic sensibility and appreciation for software and the user experience as a fine art. They should see their designs as works of art. The benchmark is *software as a fine art.* A call to software engineers to produce great software. Software that transcends issues of functionality to achieve the highest levels of reliability, usability, efficiency, maintainability, and portability. Software challenges societal views of themselves and reality. Software that inspires

1. Design problem - an identified need which requires further elicitation of the requirements.
2. Design process - elicits the requirements and proposes a selection and assemblage of design elements to solve a design problem.
3. Design elements (and notations) - what is used
4. Design structure - an assemblage of design elements
    - Structural descriptions (static view)
    - Behavior descriptions (dynamic view)
5. Architectural style - An architectural style consists of a vocabulary of design elements, a set of configuration constraints that must be satisfied by any design written in the style, and a semantic interpretation of the connectors.

Figure 4.2: **Software architecture**

4.5 Software Architecture

Software architecture is:

The *profession* of designing software
The *process* of software design
The *conceptual structure* of the software
The *actual structure* of the code

System architecture defines the arrangement of the physical, logical and information elements which come together to realize a required set of functionalities and values. *Software architecture* is that part of system architecture that focuses on the contribution of software to the overall functionality (operational and maintenance characteristics) of the system. *Information system architecture* is that part of system and software architecture that focuses on the contribution of the organization, labeling, and navigation schemes to the system. In the end, there is a user. *User interface architecture* focuses on the tools and conventions that empower the human user of the system.

Software Architecture is the art and science of designing software. The term 'software architecture' designates both the processes of designing software (*design process models*) used by software architects and the design (*structural design models*) produced. A *structural design model* is an abstract representation (conceptual design) of the various components that interact to satisfy the design guidelines and is composed of components, connections among those components, plus (usually) some other aspect or aspects. It provides sufficient detail to commission an implementation and for a software construction company to build a solution that satisfies the requirements. A *design process model* encompasses the significant decisions about (1) the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements, (2) the composition of these elements into progressively larger subsystems, and (3) the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition. An underlying, but integral, part of an architecture is the rationale for the various choices made in defining an architecture - the choice of architectural style, the choice of elements, and the form. It demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the requirements and the stakeholders' needs.

The role of the structural design model is to make a complex software system intellectually tractable in order to facilitate communication of the design, implementation of the design (Chapter 13), and evolution of the software (Chapter 12). The abstract representation may be a subjective mental construct or an objective object expressed using unambiguous notational elements. As an objective object,software architecture provides sufficient detail for a software construction company to build and commission a solution that satisfies the requirements.

There are several architectural viewpoints and approaches to describing an architectural structure.

- Logical view (satisfying the functional requirements)
- Process view (concurrency issues)
- Physical view (distribution issues)
- Development view (implementation units)

Tractability

Software may be large (millions of lines of code) and lack apparent structure. The size and interconnectivity may lead to unexpected behavior that cannot be predicted. The structural model and the rationale provide tools to manage the complexity, limit the emergent properties, and make the software intellectually tractable to the client, the software engineer, and the coder. The following items assist in reducing the complexity of software. Move to maintainability?

- Metaphor (organizational and functional)
- Hierarchical structure - part-whole description and a coordination strategy
- Componentization which depends on
  - Partitioning - a strategy of system decomposition into discrete, non-overlapping parts or components)

  - Abstraction - a naming mechanism which permits a name to represent a component (or partition).
  - Generalization - parameterization mechanism which permits one component to take the place of many similar components.
  - Encapsulation - a mechanism to hide implementation details from other components.

  and a coordination strategy which includes explicitly defined interfaces between the component parts.

- Standards - STL, APIs, Glue

Communication

The structural model is the primary mechanism for the communication of the solution to the various, often disparate groups that must understand the design, modify the design, construct (implement) an end product, or use the product. Ideally, the design is expressed using unambiguous notational elements which facilitate the mapping of the design into the elements of a implementation language.

The structural model facilitates the discovery, communication, and prediction of the

- Operational and Maintenance Qualities (see Figure 4.3)
- Economic and Technological Constraints, Trade-offs and Compromises in
  - Scope (feature set)
  - Cost
  - Development Time
  - Product Quality

Structural Models

A structural model is an abstraction that defines the organizational structure of a software system including functional components, their behaviors and interfaces, and the inter-relationship between the components (connections and communications).

A structural model occurs in at least four distinct incarnations[83]:

- The *conceptual architecture* describes the system in terms of its major design elements and the relationships among them. Two approaches are illustrated in Figures 4.4 and 4.5.

  Also, the architecture may be described using a metaphor or a common architectural pattern.

Operational Qualities	**Maintenance Qualities**
• Functionality: suitability, accuracy, interoperability, security, compliance	• Maintainability: analyzability, changeability, stability, compliance
• Reliability: maturity, fault tolerance, recoverability compliance	• Portability: adaptability, installability, co-existence, replaceability, compliance
• Usability: understandability, learnability, operability, attractiveness, compliance	
• Efficiency: time behavior, resource utilization, compliance	

Figure 4.3: **ISO Standard 9126 Software Quality Characteristics**

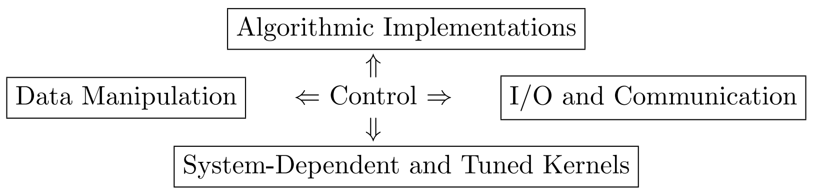

Algorithmic Implementations
Data Manipulation
⇐ Control ⇒
I/O and Communication
System-Dependent and Tuned Kernels

Figure 4.4: **Main components of a complex computer application**

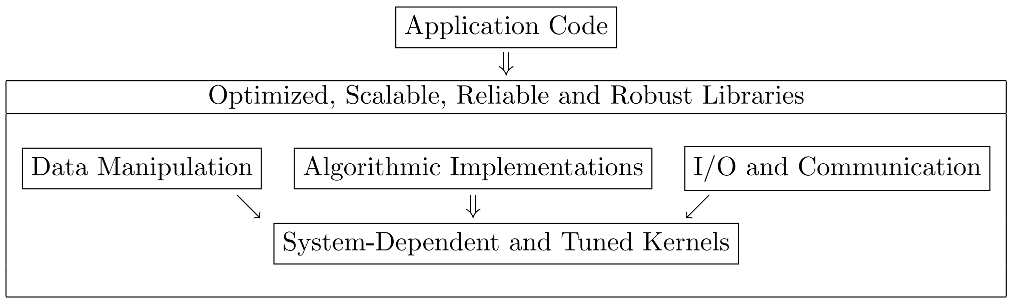

Application Code
Optimized, Scalable, Reliable and Robust Libraries
Data Manipulation
Algorithmic Implementations
I/O and Communication
System-Dependent and Tuned Kernels

Figure 4.5: **Application on top of software libraries**

- The *module interconnection architecture* encompasses two orthogonal structures: functional decomposition and layers.
- The *execution architecture* describes the dynamic structure of a system.
- The *code architecture* describes how the source code, binaries, and libraries are organized in the development environment.

The documentation of the structural model should include a rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the stakeholders' needs. It should provide sufficient detail for a client to commission a solution (an implementation) and for a software construction company to build a solution that satisfies the requirements.

The following are several types of architectures:

- Windows and desktop: These are common metaphors for the user interface to an operating system.
- Domain Specific Architectrue: The architecture for an application may utilize the language and metaphors of the domain. For example, software designed to replace a library's card catalog could utilize the familiar terminology and organization of the card catalog.
- Platform Architecture: A virtual machine (VM) or interpreter and a set of libraries
- Process Driven Architecture: The application is viewed from a process perspective. The process is broken down into activities and each activity is broken down into tasks. The process is executed inside of a process virtual machine.
- Client-server Architecture: The traditional approach to database driven applications. The server is typically a database and the user interface (client) resides on a client machine.
- Service Oriented Architecture: A service-oriented architecture (SOA) is a distributed software model that is a collection of services and a directory. The services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. The services may be dynamically discovered and used.
- Information Architecture: A combination of organization, labeling, and navigation schemes which facilitate task completion and intuitive access to content. Chapter 14 describes information architecture in more detail.

Appendix L provides a list of patterns commonly used to create architectural designs.

Design Elements and Style

There are three different classes of architectural elements: processing elements; data elements; and connection elements. The processing elements are those components that supply the transformation on the data elements. The data elements are those that contain the information that is used and transformed. The connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together.

Functional components

The three classes of architectural design elements may be classified as functional elements. The functional components in a system can be classified as follows (from Sommerville[82])

- **Sensor** - Sensors collect information from a system's environment e.g. input statements.
- **Actuator** - Actuators cause some change in the system's environment e.g., output statements, assignment operation.
- **Computation component** - Computation components are components which, given some input, carry out some computations on that input and produce some output, e.g., a floating point processor.
- **Communication component** - Communication components facilitate communication between other components e.g., global variables, data structures, arguments, sockets, TCP/IP protocols, a network linking distributed computers.
- **Coordinator** - Coordination components coordinate the interactions of other system components, e.g., control structures, procedure call, semaphores, OS scheduler, a state machine.
- **Interface** - Interface components transform the representation used by one system component into the representation used by another component facilitating the interactions of system components e.g., operator interface, a GUI

A software architecture is represented using the following concepts[58]

1. Design Elements (Objects) - Components, interfaces, and connectors are treated as first-class objects- i.e., they have a name and they are refinable.
   (a) Component: processing and data elements – objects with independent existence, e.g., a module, process, procedure, or variable.
   (b) Interface: A typed object that is a logical point of interaction between a component and its environment.

Design	Metaphor	
	Architectural Patterns	
OOP	Design Patterns	
	Objects, STL, LEDA	
Structured Programming	Data Structures	Algorithms
	Data Types	Structured Language
Machine Code	Hardware	

Figure 4.6: **Design Elements in Various Levels of Abstraction**

(c) Connector: A typed object relating interface points, components, or both.

2. Configuration: A collection of constraints that wire objects into a specific architecture.

3. Mapping: A relation between the abstract and a concrete architectures. The objects in an abstract architectural design can be decomposed, aggregated, or eliminated in a concrete architecture.

4. Architectural style: An architectural style consists of a vocabulary of design elements, a set of configuration constraints that must be satisfied by any design written in the style, and a semantic interpretation of the connectors.

   Architectural style at the level of code is a generic architecture. It consists of a topological organization of a set of parameterized modules, together with the inter-modular relationships.

   The design elements appear at different levels of abstraction (See Figure 4.6). Each level imposes configuration constraints on the design elements found at the level below.

Enterprise Applications

Enterprise applications involve persistant data, a lot of data, concurrent access to data, a lot of user interface screens, a need to integrate with other enterprise applications. Appendix M provides a list of patterns commonly used to create enterprise applications.

See [77] for more information about software architecture.

4.6 Exercises

1. Report on Service Oriented Architecture (SOA).

2. Produce a description of and rationale for your personal architectural style for program layout and critique it.

3. For each quality characteristic (functionality, reliability, usability, efficiency, maintainability, portability) describe

   (a) System architectures that provide that quality.

   (b) Design elements or methodologies that produce designs with that quality.

   (c) Programming language elements that facilitate the production of programs with that characteristic.

   (d) Coding techniques that facilitate the production of programs with that characteristic.

   If possible, illustrate with specific examples.

4. Read [84] and create some personal guidelines for producing readable code.

5. Using the functional components, produce a structural model for a program of your choice.

Chapter 5

Design Elements

Once you realize that computing is all about constructing, manipulating, and reasoning about abstractions, it becomes clear that an important prerequisite for writing (good) computer programs is the ability to handle abstractions in a precise manner. [25]

This chapter is a broad introduction to patterns as elements of design. Design patterns were introduced to facilitate design at the level of subsystems[31]. Architectural patterns[13] are overall structuring principles. In this chapter, patterns are extended downward to programming language constructs, programming idioms, data structures, and algorithms. Patterns are a unifying concept and even may be used to describe the design process. Patterns provide a vocabulary for communicating design information.

A pattern is a solution schema and a mental building block rather than a blueprint or a software component.

Example: A compiler constructed as a pipeline of functional modules.
Example: Protocol stacks are constructed as layers.

Metaphors may be used as patterns.

The concept of a pattern is broadly applicable virtually all areas of activity. The inspiration for patterns comes from the architect Christopher Alexander[53].

There are a variety of formats for the description of patterns but a template for a pattern should include at least the context in which the pattern is used, the problem the pattern is intended to solve, and description of the pattern that solves the problem.

- Context - a situation giving rise to a problem.
- Problem - the recurring problem arising in that context
- Solution - a proven resolution of the problem.

The following examples serve to illustrate the use of the template to facilitate the description of common patterns for software architecture, design patterns, data structures, algorithms, coding practices, and programming language elements.

Aside: The various levels of patterns in the following sections are similar to the various levels of patterns that occur in communication; alphabet, words, grammar, various forms of prose and poetry. □

5.1 Programming Language Elements

Programming language elements are the elementary patterns that form the basis for all programs. The elementary patterns available in a programming language are determined by the programming paradigm to which it belongs (imperative, functional, logic, etc) and the specialized requirements of application domains. The example presented here is the common assignment statement found in imperative programming languages.

Name	*Assignment Statement*
Context	Programming using an imperative programming language.
Problems	A need to associate a value with a name.
Solution	The assignment operation has the form: *variable assignOp expression* Depending on the language, the assignment operator may be confused with mathematical equality. The assignment is not communitive i.e. x = e is not the same as e = x. In strongly typed languages the variable may need to be declared. Variables used in the expression must be defined (have values). In strongly typed languages, the type of the expression must be compatible with the type of the variable.
Semantics	$\{Q[e/y]\}\ y := e\ \{y = e \wedge Q\}$
Variants	*variable* := *expression.* (Ada/Pascal) *variable* = *expression.* (C/C++/Fortran/Java) *variable* ← *expression* (APL) input commands
Known Uses	Some typical uses include: • initialize a variable (count := 0) • increment/decrement a counter (count := count + 1) • accumulate values (sum := sum + item) • capture the result of a computation (y := 3*x + 4) • swap two values (t := x; x := y; y := t)

5.2 Idioms - Low Level Patterns

Idioms are low level patterns that are specific to a programming language, a collection of languages, or a language paradigm. Idioms often deal the idiosyncrasies of a language. For example, array element access in C using pointer arithmetic. The example presented here (assertions) is a programming technique that may be used in a variety of programming languages to assist the programmer in the construction of programs that are functionally correct.

Name *Assertion*

Description Assertions are boolean expressions that are evaluated during the execution of a program to insure that both the user and the software are fulfilling their respective obligations.

Example The following definition of factorial will fail to terminate for negative input and for input values that cause overflow.

```
function factorial ( n :  integer ) :  integer
   if n = 0 then 1
   else n*factorial (n-1)
```

Context Assertions assist in checking that parameters are of the proper type and within the range that permits a module to compute correct results. Also for verifying that loop invariants and data invariants are maintained.

Problem A software system is a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations. The problem is to find a way to enforce the obligations.

Solution

```
function assert (expression :  Bool ) :  Bool
   if expression then True
   else False
```

Assertions are a programming technique designed to facilitate the enforcement of the obligations. The specifications are the provisions of the contract and are called *assertions* and are expressed as boolean expressions. The specifications of the mutual obligations constitute a contract. The enforcement of the contract is accomplished using the notion of an assertion. An *assertion* is a boolean expression about the state of a software system. Assertions are inserted into the code and are checked during the system's execution. If any assertion evaluates to false, the system is invalid.

Example resolved

```
function factorial ( n :  integer ) :  integer
   if assert( 0 < n and n < max ) then
      if n = 0 then 1
      else n*factorial (n-1)
```

Variants The assert statement may generate a side effect either printing or logging the error message.

Known Uses	Assertions may be used to provide assurance that • the preconditions and postconditions in modules, • loop invariants, and • data invariants are satisfied at execution time.
Consequences	Assertions help to verify the correctness of a program both during the implementation and maintenance phases.
See also	Error handling facilities in programming languages.
Credits	Programming by Contract is a trademarked approach using assertions to insure program correctness.

5.3 Computational Patterns: Algorithms

Fundamental algorithms are computational patterns that solve frequently occurring computational problems. They may be domain specific as occur in mathematical computation or more general algorithms such as those used to solve searching and sorting problems. The example presented here is the common binary search algorithm as used in arrays.

Name	*Binary Search*
Context	Have an ordered list of items stored in an array with indices running from Low to High inclusive.
Problem	Need to find, in an ordered list, the location of an item that corresponds to a target.
Solution	Assume $-1 < \text{L} \leq \text{U}$ where L and U are the indices of the sequence of cells to be searched, t is the target item, A is the array of cells, and the search returns either the location of the target item or -1 to indicate that the target is not within the search range.

```
function bSearch (
   t :  item,
   A : array [L..U] of items) :  integer
   //Precondition:  −1 < L ≤ U, A[i] ≤ A[i+1] i = L..U-1,
   //Postcondition:  A[L] = t or t = -1 and t ∉ A[L..U]
   if L=U then
      if A[L] = t then L else -1
   else M = (L+U) div 2
      if t ≤ A[M] then bSearch(t, A[L..M])
      else bSearch(t, A[M+1..U])
```

Complexity	Binary search is an O(log n) algorithm.
Variants	Binary search is simplified when the data structure is a binary search tree.
See also	Linear search is used when the list is not ordered. It is an O(n) algorithm.

5.4 Computational Patterns: Data Structures

Fundamental data structures are informational patterns that solve frequently occurring data storage needs and facilitate efficient algorithmic solutions. The example presented here is the common stack data structure. The approach taken in this presentation is to treat the data structure as an abstract data type.

Name	*Stack*
Also known as	LIFO (Last in, First out) Queue
Context	Implementation of a programming language, expression evaluation, and looking for efficiency.
Problem	Have recursive functions or procedures and need to improve the runtime efficiency. Have tree or graph data structure and need a non-recursive algorithm to process the data structure. Need to implement function calls.
Solution	*Algebraic definition of a Stack*

Domains:

Item ::= some collection of items
Stack :: = Nil | Item Stack
Bool ::= False | True

Functions:

newStack: () $\to$ Stack
push : Item $\times$ Stack $\to$ Stack
pop: Stack $\to$ Stack
top: Stack $\to$ Item
empty : Stack $\to$ Bool

Axioms: Let I be an Item and S a Stack.

newStack() = Nil
pop(push(I, S)) = S
push(I, pop(I S)) = I S
top(push(I, S)) = I
empty(push(I S)) = False
empty(newStack()) = True

Defining Equations: Let I be an Item and S a Stack.

newStack() = Nil
push(I, S) = I S
pop(I S) = S
top(I S) = I
empty(I S) = False
empty(Nil) = True

	Errors:
	
	pop(newStack()) or pop(Nil)
	top(newStack()) or top(Nil)

Variants	Stacks may be implemented using arrays or linked lists. They may be explicit or implicit.
Known Uses	Depth-first or breadth-first search of a tree data structure. Finding a path between two nodes in a graph. Convert arithmetic expressions to post fix notation. Hold local data for procedures and functions. Hold return addresses for procedures and functions.

5.5 Design Patterns

A design pattern[31] is a mid-level pattern that is typically applied to a subsystem. Data structures and algorithms are the concrete entities that are used in the in instantiation of a design pattern. Design patterns may be categorized into three categories.

- Creational patterns: builder, factory, prototype, singleton, etc.
- Structural patterns: adapter, bridge, composite, decorator, facade, flyweight, proxy etc.
- Behavioral patterns: chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor etc.

Design patterns emerged out of object-oriented programming and a goal to provide resuable software modules. As a consequence, classes and objects play a significant role in the description of design patterns.

The design pattern presented here is an abbreviated description of the Singleton pattern from the Creational category of patterns.

Name	*State*
Intent	Allow an object to alter its behavior when its internal state changes by appearing to change the class of the object.

Solution

1. Declare an abstract class (State) which defines the interface for each common operation that must vary based on the state of the system.

2. Define a subclass of State for each concrete state of the system.

3. Have the client (Context) class which needs to vary processing based on state maintain a reference to the abstract State class.

4. When ever the state of the system needs to change, have Context change its reference to point to the next state as needed.

5. For each state-based operation in Context, call the appropriate method in State to perform that processing.

Motivation A scenario illustrating the design problem and how the class and object structures in the pattern solve the problem.

Structure Graphical representation of the pattern using the OMT.

Participants The classes and objects and their responsibilities in the design pattern.

Collaborations How the participants collaborate to carry out their responsibilities.

Consequences

- State behavior is localized and partitioned using objects, which simplifies adding or modifying state behavior and transitions.

- State objects may eliminate the need for large conditional statements and may make the code intent easier to understand.

- State transisitions and state-dependant processing is made explicit.

- State objects might be reuseable for other systems, particularly if the state objects contain no data.

Implementation Options for implementing state transitions:

- Let Context implement transitions,

- Let each state decide the following state, or,

- Use a table-based approach.

Sample Code

Known Uses

Related Patterns

5.6 Architectural Patterns

An architectural pattern [13] is an overall structuring principle that may be used to describe a software system. It includes a set of subsystems and their relationships. Architectural patterns may be categorized as

- General structure: layers, pipes and filters, blackboard etc.
- Distributed system: client-server, three-tiers, broker etc.
- Interactive systems: Model-View-Controller, Presentation-Abstraction-Control etc.
- Adaptable systems: micro-kernel, reflection etc.
- Other: batch, interpreters, process control, rule based etc.

The architectural pattern presented here is an abbreviated description of the Model-View-Controller pattern selected from the category of interactive systems.

Name	*Model-View-Controller*
Description	This architectural pattern divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism insures consistency between the user interface and the model.
Context	Interactive applications which are subject to change.
Problem	User interfaces are especially prone to change requests and there are different types of users who require different views of the same data or different modes of interaction.
Solution	Divide an application into three parts: *processing*, *output*, and *input*. The model component encapsulates core data and functionality and is independent of specific input behavior or output representations. The view component displays information obtaining its data from the model. Each view is associated with a controller component which receive input from various input devices.
Structure	A detailed specification of the structural aspects of the pattern.
Credits	[13] p 125

5.7 Exercises

Using a template,

1. Complete a pattern description for a programming language element.

2. Complete a pattern description for an idiom such as program layout and documentation.
3. Complete a pattern description for a data structure.
4. Complete a pattern description for an algorithm
   - Sorting (bubble sort)
   - Quick sort
   - Numerical algorithm
   - data structure manipulation algorithm (graph/tree etc)
5. Complete a pattern description for a functional component.
6. Complete a pattern description for a design pattern.
7. Complete a pattern description for an architectural pattern.
8. Complete a pattern description for a GUI widget.
9. Complete a pattern description for a UI pattern.
10. Complete a pattern description for an website organizational structure pattern.
11. Complete a pattern description for making a purchase.
12. Complete a pattern description for choosing a pattern.
13. Complete a pattern description for a software engineering process such as the waterfall or spiral process.
14. Complete a pattern description for pattern descriptions.
15. Antipatterns are patterns that document those practices or designs that interfere with the successful completion of a project. They represent knowledge of what does not work. For each architectural antipattern (`www.antipatterns.com/arch_cat.htm`) indicate the quality characteristics that are affected.

Chapter 6

Functionality: Requirements and Specifications

Without a specification, the behavior of a system can't be wrong, it can only be surprising! [92]

The requirements are an informal statement of what the system must do. They are written from the perspective of the various stakeholders. The specifications are a formal statement of what the system must do. They are written for the developers. The Software Quality Characteristics listed in the ISO Standard 9126 (Figure 1) provide a list of areas to consider in creating a complete set of requirements and specifications.

6.1 Requirements

The requirements are an informal statement of what the system must do. They are written from the perspective of the various stakeholders.

A stakeholder is anyone that supplies resources for a project/product and expects something in return. The stakeholder expects assurance that the activity will be reliably effective and efficient in the use of the resource i.e., that the stakeholder will receive a reasonable return on the investment.

The primary stakeholder is the client. A client's knowledge may be tacit (clients may not be able to accurately describe what they do). The users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. A client may not know what is technically feasible. Clients may change their minds once they see the possibilities more clearly. Therefore, it may be impossible to say with certainty what the requirements are, let alone whether or not they are met, until the system is actually in place and running. These discoveries made during the later phases

$\boxed{\text{Requirements elicitation(Stakeholders)} \rightarrow \text{Requirements Document}}$

Figure 6.1: **Requirements**

may force retrofitting requirements. Often clients often do not appreciate the complexity inherent in software engineering, and the impact of changed requirements. In addition, there may be many social, political, legal, financial, and/or psychological requirements which affect system design. See Appendix 3 for more information on the human factors in software.

The requirements are elicited (systematically extracted and inventoried) from a combination of human stakeholders, the system's environment, feasibility studies, market analysis, business plans, analysis of competing products and domain knowledge.

Requirements elicitation activities include interviews, facilitated meetings, direct observation of work patterns, and the construction of scenarios. Prototypes are a useful tool to elicit requirements. In a formal sense, epistemological methods are used to extract and construct an ontological object to be implemented in software.

User Stories written by the customers describing things that the system needs to do for them.

Scenario an example of system use in terms of a series of interactions between the user and the system.

Use case an abstraction that describes a class of scenarios.

User interface navigational paths and screen mock-ups.

Prototype a "working model" for demonstration or use.

The work product of the requirements process is a requirements document (Figure 6.1). Appendix A provides an outline of what is expected in a formal requirements document. In addition to the requirements, it may be helpful to document the rationale for each requirement and what the emergent properties of an implementation might be.

6.2 Specifications

The rest of this chapter is quite formal and uses symbolic language.

The specifications are a formal statement of what the system must do. They are written for the developers.

The formal approach to specifications considered in this and the following sections attempt to provide a formal methodology for deriving a correct program from its specifications [26, 35].

Definition 6.1 *A* relational structure *is a set of constants, a set of relations, and a set of functions defined on the set of constants.*

Note: relations may be treated as Boolean functions.

Example 6.1 *The natural numbers,* $\mathbb{N} = \{0, 1, 2, ...\}$*, with the less than relation,* $<$*, and the add,* $+$*, and multiply,* $\times$*, functions is an example of a relational structure.*

Relations are ordered tuples, $<$ is a set of ordered pairs and that both $+$ and $\times$ are sets of ordered triples ($(3, 10) \in <$, $(3, 4, 7) \in +$, $(3, 4, 12) \in \times$).

Definition 6.2 *A* many-sorted relational structure *is a set of sorts (a collection of sets of constants), a set of relations, and a set of functions.*

Example 6.2 *The sorts* $\mathbb{N} = \{0, 1, 2, ...\}$ *and* $\mathsf{Boole} = \{\mathsf{false}, \mathsf{true}\}$ *with the usual Boolean operations and arithmetic functions is an example of a many-sorted relational structure.*

Definition 6.3 *A* state *is a many-sorted relational structure.*

Every relational structure has an associated language. The language is used to reason about the relational structure. Alfred Tarski developed relational structures to provide semantics for logic.

In the case of computational sequences (to be defined later), the language provides a means to reason about sequences of relational structures. The language provides names for each of the constants and for each relation.

Predicate Logic: The octuple, $M = (S, C, \mathbf{f}, \supset, \forall, V, P, T)$, defines *predicate logic* where

- S is a set of sentences or propositions defined as follows:
  - **f** is a distinguished element of S called *falsehood.*
  - $P^i_j(c_{k,0}, c_{k,1}, ..., c_{k,i-1})$ are elements of S where $P^i_j \in P$, $c_{i,j} \in C$, and i, j, k are in $\mathbb{N}$. If i = 0, there are no arguments.
  - $\supset$ is a binary operation on elements of S such that if X, Y $\in$ S, then X $\supset$ Y $\in$ S.
  - $\forall$ is a binary operation on elements of V (a set of variables) and S such that if $A(c)$ is a formula in S with c a constant in C, x a variable in V not appearing in $A(c)$, and $A(x)$ is $A(c)$ with all occurrences of c replaced by x, then $\forall x A(x) \in S$.

  Note: Where necessary, we will use parentheses, brackets, and braces to indicate grouping. Abbreviations:

  - $\neg X$ is $X \supset \mathbf{f}$
  - $X \wedge Y$ is $\neg(X \supset \neg Y)$

  - $X \vee Y$ is $\neg X \supset Y$
  - $X \equiv Y$ is $(X \supset Y) \wedge (Y \supset X)$
  - $\exists x A$ is $\neg\forall x \neg A$

- The set T is defined as follows. Any formula of S of the following forms (called axiom schemas) is in T.

    1. $A \supset (B \supset A)$
    2. $[A \supset (B \supset C)] \supset [(A \supset B) \supset (A \supset C)]$
    3. $\neg\neg A \supset A$
    4. $\forall x A \supset A(a)$ where $A(a)$ is A in which every occurrence of x is replaced by a where a is any constant or a variable not occurring in A.

  The set T is closed under the following *inference* rules:

    1. *Modus ponens:* Whenever formulas of the form A and $A \supset B$ are in T then so is the formula B.
    2. *Generalization:* For each formula A in T, $\forall x A$ where x is a variable is in T.

Definition 6.4 *A* Kripke Structure *is a set of relational structures together with an accessibility relation which describes whether a given relational structure is accessible from another relational structure. Thus, a Kripke structure is a graph whose nodes are relational structures and whose edges are defined by the accessibility relation.*

Model logics provide a language for specifying and reasoning about Kripke structures.

Definition 6.5 *Let $\mathcal{S}$ be a set of states and $\tau : \mathcal{S} \to \mathcal{S}$ be a map. A* computation *is a finite or infinite sequence of states*

$$x_0, x_1, x_2, ...$$

where x_0 is an initial state and every $x_{i+1} = \tau(x_i)$.

A computation may be viewed as a path through a Kripke structure or as the evolution of a state. Temporal logics and algorithmic languages are used to specify and reason about computations. This last observation leads to the following definition.

Definition 6.6 *A* program *is a specification of a computation.*

The construction of a program is an iterative process that elaborates the requirements and formalizes the specification of a relational structure and a computation.It involves identifying events, data items, and ordering events.

expanded.

6.3 Abstract State Machine

Abstract State Machines (ASMs) provide an algorithmic language for specifying and reasoning about computational sequences. They provide a mechanism to describe arbitrary states of arbitrary algorithms at their natural abstraction levels.

6.4 Functional Specifications

recursive functions and relations

A specification is a contract between the user of a system and its implementor. The contract tells the user everything necessary to use the system and tells the implementor everything necessary to implement it.

An implementation should satisfy the specification. A dispute over whether an implementation satisfies an informal specification is settled in a court handling contract law. The question whether an implementation satisfies a formal specification is reducible to the question whether an assertion (the specification) is provable in some mathematical system which describes the properties of the implementation.

Safety properties assert what the system is allowed or not allowed to do. Partial correctness is the safety property that asserts that a program may not generate an incorrect answer. Liveness properties assert what the system must do. The assertion that a program must termination and generate an answer is a liveness property.

Transition axiom method [51]

There is an unspecified set of states $\mathbf{S}$, a set of values of interest $\mathbf{V}$, state function $\mathbf{f} : \mathbf{S} \rightarrow \mathbf{V}$, and a set of actions $\mathbf{A}$.

A safety specification consists of:

- A set of state functions, partitioned into interface and internal state functions.
- A specification of the initial value of every state function.
- A set of actions, partitioned into interface and internal actions.
- For each interface action, a specification of who performs the action. (Internal actions are always performed by the system.)
- A set of rules, called transition axioms, that describe how each action changes the state functions. An interface state function may be changed only by an interface action.

A liveness specification is written in temporal logic, the subject of the next section.

	Safety Properties
$\Box$ (eating(i)$\vee$thinking(i))	Philosophers either eat or think
$\Box\neg$ (eating(i)$\vee$eating(i+1))	Adjacent philosophers cannot eat simultaneously
	Liveness Properties
$\Box$(thinking(i)$\rightarrow\Diamond$eating(i))	Philosophers alternate between eating and thinking
$\Box$(eating(i)$\rightarrow\Diamond$thinking(i))	

Figure 6.2: Safety and Liveness Specifications: Philosopher P(i)

6.5 Temporal Logic for Specification

Temporal logic is ordinary logic extended with temporal operators $\Box$ (read *henceforth*) and $\Diamond$ (read *eventually*). The formula $\Box P$ asserts that P is true now and at all future times, and the formula $\Diamond P$ asserts that P is true now or at some future time. Since P is eventually true if and only if it is not always false, $\Diamond P$ is equivalent to $\neg\Box\neg P$.

Temporal logic, as it has been defined here, cannot formally specify things like average response time and probability of failure. However, it is useful for the specification of safety and liveness properties. *Safety properties* assert what the system is allowed to do, or equivalently, what it may not do. Safety properties are satisfied by a system which does nothing. Restriction to only producing correct answers is an example of a safety property. *Liveness properties* assert what the system must do. Termination is an example of a liveness property.

As an example of temporal specifications and safety and liveness specifications in particular, we provide a specification of the *The Dining Philosophers Problem.* Five philosophers spend their lives seated around a circular table thinking and eating. Each philosopher has a plate of spaghetti and, on each side, shares a fork his/her neighbor. To eat, the philosopher must acquire two forks. The problem is to prevent deadlock or starvation i.e. insure that each philosopher gets to eat.

Fairness is a desirable property of a concurrent system and is definable as a liveness property.

Generalized weak fairness: $\Diamond\Box P \rightarrow \Box\Diamond A$

Generalized strong fairness: $\Box\Diamond P \rightarrow \Box\Diamond A$

Formally, an *action system* consists of an initial state predicate *Init* and a set of predicates A_i on pairs of states. The A_i are called *system actions.* An action system expresses the safety property consisting of every behavior s_0, s_1, ... whose initial state s_0 satisfies *Init* and whose every pair s_i, s_{i+1} of successive states satisfies some system action.

Temporal logic specifications may be compiled into an executable program.

6.6 Exercises

1. What process should be followed in eliciting the requirements for an ill-defined problem?
2. Pick a wicked problem and gain some experience eliciting the requirements.
3. Read *An Abstract State Machine Primer*[42] and do the following:
   (a) Formalize in concise form the specifications for the examples (GCD, StringMatching, Minimum Spanning Trees).
   (b) Complete the exercises at the end of the paper.
4. Fairness: Produce in temporal logic a definition of fairness.
5. For each of the following, produce temporal logic specifications. Clearly indicate the safety and liveness conditions.
   (a) Soda machine: Upon accepting 50 cents in quarters or half-dollars, the soda machine dispenses a soda.
   (b) Producer/Consumer Problem (Bounded Buffer): There is a pool of n buffers that are filled by one or more producer processes and emptied by a consumer process. The problem is to prevent an overlap of buffer operations and to keep the producer from overwriting full buffers and the consumer from reading empty buffers.
   (c) Readers and Writers: There is a shared data structure. There are *reader* processes and *writer* processes. The following conditions must be satisfied.
      - Any number of readers may simultaneously read the data.
      - Only one writer at a time may write to the data structure.
      - If a writer is writing to the file, no reader may read it.

      Any waiting reader or writer must eventually have access to the data structure.
   (d) The Barbershop Problem: The barber shop has n barbers, n barber chairs, and a waiting area with a sofa, There is a limitation of m customers in the shop at a time. The barbers divide their time between cutting hair, accepting payment, and sleeping in their chair waiting for a customer. A customer will not enter the shop if it is filled to capacity. Once inside, the customer takes a seat on the sofa or stands if the sofa is full. When a barber is free, the customer that has been waiting the longest on the sofa is served and the customer that has been standing the longest takes its place on the sofa. When a haircut is finished any barber can accept payment but payment can be accepted for only one customer at a time as there is only one cash register.

(e) User interface: Describe the user interface of some system in terms of an action system.

Chapter 7

Design Process

> Design is a process, by which one or more designs (a plan - the relationship between design elements) are produced which satisfy the functional and non-functional requirements, at least one of which is a *good design* (satisfies an aesthetic value).

The process of software architecture begins with a design problem. The goal is to produce a description of the architecture of the system, its decomposition and organization into components and their interfaces, in such a level of detail to allow for its construction. There is no sharp line between requirements elicitation and design. The process includes requirements elicitation, analysis of the requirements requirements and the selection of components.

Perhaps the most important design decision is a choice of metaphor to drive the design process. Metaphors effectively convey relationships and functionality. They are an excellent choice for the conceptual design.

> Choose a system metaphor to keep the team on the same page by naming classes and methods consistently. What you name your objects is very important for understanding the overall design of the system and code reuse as well. Being able to guess at what something might be named if it already existed and being right is a real time saver. Choose a system of names for your objects that everyone can relate to without specific, hard to earn knowledge about the system.
>
> For example the Chrysler payroll system was built as a production line. At Ford car sales were structured as a bill of materials. There is also a metaphor known as the naive metaphor which is based on your domain itself. But don't choose the naive metaphor unless it is simple enough.
>
> – from `http://www.extremeprogramming.org/rules/metaphor.html`.

There are several techniques that enable design.

- Metaphor: Metaphors are powerful tools to drive a design and communicate a design. They effectively convey relationships and functionality.
- Decomposition and modularization: the structuring of a large system as a collection of independent components.
  - Separate the interface from the implementation: the separation of the specification of a public interface from the implementation details of the component.
  - Minimize Coupling: the degree of independence between modules.
  - Maximize Cohesion: the degree of dependence within a module.
  - Insure Sufficiency, Completeness, and Primitiveness: ensuring that a component implements all the important characteristics of an abstraction.
  - Encapsulate Information (information hiding): making internal details of a module inaccessible.
- Abstraction: which hides details by permitting objects to be referenced by a name.
- Generalization: which increases the applicability of an object through parameterization.

Each of the software quality characteristics must be addressed in the resulting design. In particular, the design must describe how the following issues are handled.

- Concurrency: processes, tasks, threads; efficiency atomicity, synchronization, and scheduling.
- Control and handling of events: flow of data and flow of control, reactive and temporal events.
- Distribution: component communication and middleware for heterogeneous systems.
- Interactive systems: how the system is to interact with users.
- Persistence: how long-lived data is to be handled.

An architectural style guides the composition of the functional elements, their interfaces, and their collaborations into a design. Designing a software system using a generic architecture consists of instantiating the parameters of each parameterized module by a concrete value while maintaining the inter-modular constraints.

7.1 Design Aspects

There are four aspects to a design:

1. Functionality
2. Structure
3. Distribution
4. Configuration

7.2 User Centered Design

User centered design focuses on the users of the systems with the goal to produce user-friendly designs. User types include naive, uninitiated, expert, administrator, and maintainer. User centered design focuses on answering the question, "What can people do with a computer?" This is in contrast with machine centered design which focuses on answering the question, "What can computers do?" Design is facilitated by constructing *scenarios*, and *use cases* for each type of user. *Prototypes* are useful for verifying the requirements and desired user experience.

7.3 Modeling and OO Design

Object-Oriented approach views the world as a collection of objects with data structures and behaviors and events that trigger operations, or object behavior changes, that change the state of objects. The idea that a system can be viewed as a population of interacting objects, each of which is an atomic bundle of data and functionality, is the foundation of object technology and provides an attractive alternative for the development of complex systems.

The goal of Object-Oriented Design (OOD) is to model the problem domain as a system composed of components called objects each of which is a representation of a real-life entity or abstraction. An object specification specifies the structure and the behavior of the object. Different types of models are required to specify the requirements of the objects. The information or object model contains the definition of objects in the system, which includes: the object name, the object attributes, and object relationships to other objects. The behavior or state model describes the behavior of the objects in terms of the states the object exists in, the transitions allowed between objects, and the events that cause objects to change states.

The resulting design is a representation of reality. The execution of the resulting program is a simulation.

Initially there is a focus on objects which attempts to identify objects and their functionality in the application domain. Given a problem statement,

- single out the nouns, verbs, and adjectives and create corresponding objects, methods, and attributes, or
- focus on the collaborations and responsibilities in the system, or
- model the real world and translate the objects found during analysis into design.

The design may be expressed using the Unified Modeling Language (UML). A state machine is often used to coordinate the activities of the objects.

With experience and a family of applications OOD can focus on reuse. This leads to reuse-centered design (previously developed objects and patterns).

7.4 Data Centered Design

Data centered design starts with the data a system must manipulate rather than the functions the system must perform. Design continues with identifying the transformation required.

Relational Database Design

Relational database design begins with the identification of entities and their attributes and relationships between entities (Entity-Relationship diagrams). The E/R diagrams are then mapped to a relational schema.

An alternate approach is to create a super relation containing all the attributes of interest and a list of functional dependences between the attributes. In either case, the relations are normalized to remove redundancies and to eliminate update anomalies.

The design phase contains six sub-phases.

1. Conceptual design consists of E/R modeling or OOD and answers the question “What entities and relationships are needed?”
2. Choice of DBMS answers the question “What is the appropriate DBMS?”
3. Logical design maps the conceptual design to a database schema answering the question “What database model?”
4. Schema refinement is performed to eliminate redundancy through what is called normalization of database schema. It answers the questions “It this the simplest schema?”
5. Physical design looks at performance issues and answers the question “How to get the performance that is needed?”
6. Security design considers the various user groups and access restrictions.It determines what security is necessary and produces an appropriate security design.

Information Architecture (Website)

The information architecture design is preceded by research and strategy phases and followed by implementation and administration. The design emerges out of research into

- Context: the business goals, funding, politics, culture, technology, and human resources.
- Content: the document/data types, content objects, metadata, volume, and existing structure.
- Users: the audiences, tasks, needs, information seeking behavior, experience, and vocabularies.

and the development of an architecture strategy. An architecture strategy is developed through the use of metaphors, scenarios, case studies, conceptual diagrams, blueprints and wireframes. Metaphors are often grouped into

- Organizational metaphors connect the site organization to the user's experience in other domains. Examples include desktop, store etc.
- Functional metaphors make connections between tasks performed in a traditional environment and those in the new environment. A library site would include functions such as browsing, catalog search, and help desk.
- Visual metaphors use graphic elements such as images, icons, and colors to create connections. Visual elements are particularly useful to draw user's attention to changes since their last visit/use.

Scenarios describe a site in terms of the expected interactions of classes of users. Conceptual diagrams, blueprints, and wireframe illustrate the relationships between content, pages, and links.

7.5 Other methods

Functional Design

Functional design (also called structured design and functional decomposition) focuses on identifying the required functionalities. It identifies the major systems and their elaboration and refinement in a top-down manner. The architectural design is presented as a structure chart.

Each functional component should be documented with a specification which includes a precondition and a postcondition. The *precondition* specifies the obligations that a user of the component must meet before the component may be used. The *postcondition* specifies the condition that holds after the execution of the component. It is a guarantee of service provided the precondition is satisfied.

Behavioral Models

Data-Flow Models

Data-flow models model the data processing in the system. The flow of data through a system, i.e., the processing steps necessary to transform input data into output data, determine the structure of the system.

Reactive System, State Machine, & Event-Driven

Teller machines and GUI interfaces are examples of systems that wait for events. Such systems react to events and, depending on the event, perform an action and change state. State machines are used to model how the system reacts to events. The state machine shows system states and events which cause transitions from one state to another.

Formal Methods and Correctness

Formal methods are used to derive correct programs from specifications. The process begins with the construction of formal specifications in a formal specification language. Then, if possible, the resulting specifications are shown to be complete and consistent. Sound transformations are applied to specification to transform it into a provably correct program.

A less ambitious approach utilizes assertions validate inputs to subprograms. See Chapter 13 for more details.

7.6 Pattern Selection

There is no substitute for familiarity with a wide range of patterns. Familiarity with patterns can speed up the design process. Appendix L contains a list of common architectural patterns. Appendix M contains a list of common enterprise application patterns. Appendix O contains a list of common information architecture (website) patterns. Appendix K contains a list of common design patterns. Appendix N contains a list of common widgets and page designs.

The following selection process is a combination of those suggested by Gamma et al [31] and Buschmann [13].

1. Specify the problem
   - (a) Consider how design patterns solve design problems.
   - (b) Scan intent sections
   - (c) Study how patterns interrelate
   - (d) Study patterns of like purpose
   - (e) Examine a cause of redesign
   - (f) Consider what should be variable in your design

2. Select the pattern category that corresponds to the design activity.
3. Select the problem category that corresponds to the general nature of the design problem.
4. Compare the problem descriptions.
5. Compare benefits and liabilities.
6. Select the variant that best implements the solution to your design problem.
7. Select an alternate problem category.

It is possible that no pattern exists that fits your particular problem. It is then necessary to create a unique solution and your solution may give rise to a new pattern. If so, carefully describe it and share it with the software community.

7.7 The Design Document

Appendix B outlines what is expected in a formal software design document.

In architectural design (top level design) the top-level structure and organization of the system is described and various components are identified (how the system is decomposed and organized into components and the interfaces between these components). The design should be expressed using structural and/or behavioral descriptions.

- Structural descriptions (static view)
  - Architecture description languages
  - Class and object diagrams
  - Component diagrams
  - CRC cards
  - Deployment diagrams
  - Entity-Relationship diagrams
  - Interface description languages
  - Jackson structure diagrams
  - Structure charts
- Behavior descriptions (dynamic view)
  - Activity diagrams (flow of control)
  - Collaboration diagrams (interactions among a group of objects)
  - Data flow diagrams

  - Decision tables and diagrams
  - Flowcharts and structured flowcharts
  - Formal specification languages
  - Pseudo-code and program design languages
  - Sequence diagrams
  - State transition and statechart diagrams

The rationale for the proposed software architecture facilitates the evolution of the software. Rationale should be provided for the following:

- Subsystem decomposition
- Hardware/software mapping
- Persistent data management
- Access control and security
- Global software control
- Boundary conditions

In the implementation design, each component is sufficiently described to allow for its coding.

7.8 Design Validation

The following remarks on model validation apply to the work products produced in each phase of the software engineering process: need or problem statement, requirements, specification, architectural design, detailed design, and implementation.

Shanks et. al.[71] propose an ontological approach to conceptual model validation. A model is a faithful representation of a domain as perceived by the stakeholders if it possesses the following attributes.

- Accuracy. The model accurately represents the semantics of the domain.
- Completeness. The model is a complete representation of the semantics of the domain.
- Conflict-free. The semantics are free of contradictions.
- No redundancy. The model does not contain redundant semantics.

Validation is the process of checking that the model faithfully represents the domain. The process of validation must address three issues – scope, people, and methodlogy.

1. Scope.

2. The people involved in the process.

3. The methodology.

   There are several approaches:

   - Review.
   - Questioning.
   - Problem solving.
   - Transaction testing.

7.9 Refactoring

Refactoring is the replacement of a section of code by another. Refactoring occurs in the development process or during maintenance when the observation is made that there is a better algorithm, data structure, or architecture than is currently in use. See p. 24 in [31].

7.10 Some Structural Formalisms

Relational Notation

Functional decomposition/differentiation:

$$f(x,y) \leftarrow f_n(x,y_n),...,f_0(x,y_0),g(y_n,...,y_0).$$

There are $n+1$ functional areas each of which processes the input in parallel. Applications: ...

Domain decomposition:

$$\begin{array}{lcl} f([],[]). & & \\ f(x:xs,y_x:y_{xs}) & \leftarrow & g(x,y_x),f(xs,y_{xs}) \end{array}$$

Applications: a queue, client-server, ...

$$\begin{array}{lcl} cs(x:xs,y:ys) & :- & c(x:xs,y:ys),s(x,y),cs(xs,ys). \\ f(x:xs,y_x:y_{xs}) & \leftarrow & g(x,y_x),f(xs,y_{xs}) \end{array}$$

$$cs(x:xs,y:ys) \leftarrow c(x:xs,y$$

... Left incomplete with respect to the functional formalisms below.

Functional Notation

Functional decomposition/differentiation:

$$f(x) = g(f_n(x), ..., f_0(x))$$

There are $n+1$ functional areas each of which processes the input in parallel. Applications: ...

Domain decomposition:

$$f([]) = []$$
$$f(x : xs) = g(x) : f(xs)$$

Applications: a queue, client-server, ...

Pipeline:

$$p(x : xs) = f_n(...f_1(f_0(x))...) : p(xs)$$

There are $n+1$ processing steps; the output of one is input to the next. Applications: assembly line, CPU-instruction processing, ...

Divide and conquer:

$$f(x) = \begin{array}{ll} g(x) & \text{if terminal}(x) \\ h(f(x_0), ..., f(x_n)) & \text{otherwise} \\ \text{where divide}(x) = < x_0, ..., x_n > & \end{array}$$

A complex input is divided among $n+1$ processes with the parts processed in parallel. Applications: multiple queues, ...

Structured analysis and design:

7.11 Exercises

1. Evaluate the various design approaches for their suitability for conceptual and detailed design.
2. Describe a design methodology using a pattern template.
3. Suggest design methodology anti-patterns.
4. Pick a problem and several design processes. Apply each process to construct an architectural solution. Compare and contrast the results.
5. Compare and contrast Genex with Extreme Programming.
6. For each design process, identify appropriate problem domains.
7. Research the Axiomatic Design process and determine its suitability for the design of software.
8. Research the TRIZ design process or Axiomatic Design Theory and determine its suitability for the design of software.

Chapter 8

Security and the Design of Secure Software

Security provides protection against unauthorized disclosure, alteration, or destruction of data and protection against unauthorized users.

Security is one of the functional qualities of software (Figure 1) and is related to the reliability quality characteristic in that software features that compromise security are often those same features the lead to unreliability. Good security protects critical infrastructures and is necessary to build and preserve trust in computing. A formal security development component to the software development lifecycle is mandatory whenever software is:

- Expected to be used to process personal or sensitive information
- Expected to be used in an enterprise or other organization (including academia, government, or non-profits)
- Expected to be connected to the Internet or otherwise used in a networked environment

Stand-alone applications do not fit the criteria above. However, no software should interfer with the security of the platform (operating system or other software) on which the software operates.

It is appropriate to treat security as risk management. Although the terminology may differ, the essential activities of security risk management are the same (See Figure 8.1).

This chapter is based on material in [41, 55, 92].

8.1 The Security Context

The goal of system security is to provide confidentiality, integrity, and availability. That is: information should not be disclosed to unauthorized users; only

Risk Management	**Security Management**
Risk Identification	Vulnerability and Threat Identification
Risk Analysis	Vulnerability and Threat Assessment
Risk Response Planning	Security Philosophy, Policy, Procedures
Risk Monitoring and Control	Security Audit
	Security Incident Handling

Figure 8.1: **Security Risk Management**

authorized users should be allowed to modify data; authorized users should not be denied access.

Authorization implies two phases. The first is a method for the identification and authentication of users, processes or hosts and the second a mechanism to provide access.

Identification – Who are you? Authentication – Prove it. Authorization – Here is what you are allowed to do. from "Sensible Authentication" by Bruce Schneier in the *ACM Queue* Vol 1 No 10 February 2004.

Secure systems use a variety of mechanisms to protect confidentiality, insure integrity, and authenticate users and systems. Some mechanisms are provided by the operating system others by various subsystems such as a database management system. The mechanisms include administrative policies as well as hardware and software methods.

The reality is that security breaches will occur. Therefore, software must be designed with a reasonable level of preventive measures and must assist in the detection of security attacks and breaches, provide for smooth recovery from a breach, and provide the information necessary for code maintainers to locate and remove the vulnerability. It is these latter aspects that tie security to reliability.

Security is a functionality that may not be visible to the user. It provides enforcement of the rules for accessing resources. In an abstract sense, "Security is a measurement of how robust the system (software) is with respect to a particular security policy." [92] The accuracy or validity of data is absolutely fundamental, security is more of a secondary issue.

The key system factors that make security difficult are system complexity, system extensibility, system accessibility (via networks etc), and lack of programming language support for those features that support security. Security holes and vulnerabilities are often the result of incomplete/bad software design and implementation and thus occur as a sub-characteristic of reliability. Many of the methods used to insure correctness, reliability, fault tolerance, and maintainability are used to achieve security.

Microsoft has identified four high-level principles for building more secure software. The brief definitions of these principles are:

Secure by Design the software should be architected, designed, and implemented so as to protect itself and the information it processes, and to resist attacks.

Secure by Default in the real world, software will not achieve perfect security, so designers should assume that security flaws would be present. To minimize the harm that occurs when attackers target these remaining flaws, software's default state should promote security. For example, software should run with the least necessary privilege, and services and features that are not widely needed should be disabled by default or accessible only to a small population of users.

Secure in Deployment Tools and guidance should accompany software to help end users and/or administrators use it securely. Additionally, updates should be easy to deploy.

Communications software developers should be prepared for the discovery of product vulnerabilities and should communicate openly and responsibly with end users and/or administrators to help them take protective action (such as patching or deploying workarounds).

A complete security solution provides for the following:

- **Prevention.** Use multilevel security methods to authenticate users, hosts, and software, to enforce access rules, and use cryptographic techniques where data may be exposed.

- **Detection.** Detect an attack through monitoring (real-time auditing) such as using an intrusion detection system and a file integrity checker.

- **Recovery.** Recover from an attack through backups, logging, auditing and traceability measures.

8.2 System Components and Points of Attack

A system consists of hosts (servers and clients), networks, applications, system administrators, users, and attackers. Physically, the systems may be simply described as: Host - Communication Link - Host (Man in the middle) - Communication Link - Host. The combination of hosts and communication links offers a number of points of attack.

- The People. Social engineering targets users and the staff bluffing them into providing physical access or passwords. Countermeasures include user and staff education and system monitoring

- The Server. The common approach is to use malicious input to take advantage of buffer overflows. Countermeasures include using programming languages with strong typing, pre- and post-conditions, and good programming practices such as pairs programming and code reviews.

- The Software.

- The Data.

- The Network. For various reasons, networks may drop, corrupt, or duplicate packets. These network problems may be confused with an attack. An attacker may use an intermediate host to spoof another host and eavesdrop on, tamper with, or hijack a stream of packets. Countermeasures include using cryptography and authentication.

There are four general categories of attack. *Interception* is an attack on confidentiality. *Modification* is an attack on integrity. *Interruption* is an attack on availability. *Fabrication* is an attack on authenticity.

Security Team

The responsibilities of the team include:

- Development, maintenance, and enhancement of the security related aspects of the software lifecycle, including definition of mandatory aspects of the process.

- Development, enhancement, and delivery of engineer education.

- Provision of "security advisors" who guide product teams through the process, conduct reviews for product teams, and ensure that product team questions receive timely, accurate, and authoritative responses.

- Serving as subject matter experts on a broad range of security topics, ensuring that questions directed to or through security advisors receive timely and accurate answers.

- Execution of Final Security Reviews before software is released.

- Technical investigation of reported vulnerabilities in software that has been released to customers, to ensure that root causes are understood and the proper level of response is initiated.

Having a central and highly qualified security team is key to bringing product teams across the company up to speed and supporting them as they work to build more secure software. It also ensures that someone outside of the product team conducts the final security review (FSR).

8.3 Security and the Software Life-Cycle

> *To build a secure system, there must be a security model at the core of the system that is simple enough that the designers can actually understand it, and resist all pressure to deviate from it in order to add new features.* - Andrew Tanenbaum

There are security specific activities for each phase of the software life cycle (See Figure 8.2).

Phase	**Actvity**
Requirements	*Functionality*
	Security
	Architecture
Design	*Security Architecture*
	Attack Review
	Threat modeling
Implementation	*Security Development Tools*
	Practices
	Testing
	Documentation
Verification	*Security Response Plan*
	Final Security Review
Release	*Code Signing*
	Sign-off
	Release
Support and Servicing	*Product Support*
	Service Packs
	Security Updates

Figure 8.2: **Security and the Software Life Cycle**

Elicitation of the Security Requirements

The elicitation of the security requirements begins with the answers to the question "The system must be secure from what and from whom?" The relation of security to the business objectives and other central concerns may be determined by answers to questions such as the following.

- What do your users expect in the way of system security?
- Will you lose customers if security is not taken seriously enough, too seriously, or if so severe that functionality is impaired?
- How much down-time or monetary loss has occurred due to security incidents in the past?
- Are you concerned about insider threats? Should you trust your users?
- How much sensitive information is on-line? What is the loss to the organization if this information is compromised or stolen?
- Do you need different levels of security for different parts of your organization?
- What is your network and host configuration? Do you support dangerous network services? Do you require individual hosts to meet a basic security profile?

- Are there security guidelines, regulations, or laws you are required to meet?
- Do business requirements take precedence over security where there is a conflict?
- How important are confidentiality, integrity and/or availability to the overall operation of your company or site?
- Are the decisions you've made consistent with your business needs and economic reality?

The end result of the security requirements elicitation process is a security policy. A *security policy* is a collection of rules for authorizing access to a resource.

Security Design

The design phase identifies the overall requirements and structure for the software. From a security perspective, the key elements of the design phase are:

- Define security architecture and design guidelines: Define the overall structure of the software from a security perspective, and identify those components whose correct functioning is essential to security (the "trusted computing base"). Identify design techniques, such as layering, use of strongly typed language, application of least privilege, and minimization of attack surface, that apply to the software globally. (Layering refers to the organization of software into well-defined components that are structured so as to avoid circular dependencies among components–components are organized into layers and a higher layer may depend on the services of lower layers, while lower layers are forbidden from depending on higher layers.) Specifics of individual elements of the architecture will be detailed in individual design specifications, but the security architecture identifies an overall perspective on security design.
- Document the elements of the software attack surface. Given that software will not achieve perfect security, it is important that only features that will be used by the vast majority of users be exposed to all users by default, and that those features be installed with the minimum feasible level of privilege. Measuring the elements of attack surface provides the product team with an ongoing metric for default security and enables them to detect instances where the software has been made more susceptible to attack. While some instances of increased attack surface may be justified by enhanced product function or usability, it is important to detect and question each such instance during design and implementation so as to ship software in as secure a default configuration as feasible.
- Conduct threat modeling. The product team conducts threat modeling at a component-by-component level. Using a structured methodology,

the component team identifies the assets that the software must manage and the interfaces by which those assets can be accessed. The threat modeling process identifies threats that can do harm to each asset and the likelihood of harm being done (an estimate of risk). The component team then identifies countermeasures that mitigate the risk-either in the form of security features such as encryption, or in the form of proper functioning of the software that protects the assets from harm. Thus, threat modeling helps the product team identify needs for security features as well as areas where especially careful code review and security testing are required. The threat modeling process should be supported by a tool that captures threat models in machine-readable form for storage and updating.

- Define supplemental ship criteria. While basic security ship criteria should be defined at the organization level, individual product teams or software releases may have specific criteria that must be met before software can be released. For example, a product team that is developing an updated version of software that is shipping to customers and subject to extensive attack might elect to require that its new version be free from externally reported vulnerabilities for some period before being considered ready for release. (That is, the development process should have found and removed the vulnerabilities before they were reported rather than the product team having to "fix" them after they are reported.)

Given a preliminary system design, security analysis of the design focuses on identifying the following.

- Data flow between components
- Users, roles, and rights (explicit or implicit) in the design
- Trust relationships of each component
- Potentially applicable solutions to recognized problems

Armed with the results of the analysis and the following guiding principles, the security designer produces an architectural security design.

1. Secure the weakest link.

   Security is a chain which is as strong as its weakest component. Attackers will look for weak points before data is encrypted or after data is decrypted because attacking encrypted data requires too much work. The weakest link often turns out to be the users and the system administrators who are subject to social engineering attacks.

2. Practice defense in depth.

   Defense in depth manages risk with diverse defensive strategies so that if one layer is inadequate, another layer will prevent a full breach. Multiple firewalls combined with data encryption are common techniques to provide defense in depth.

3. Fail securely.

   Failure is often unavoidable. What is avoidable are security problem related to failure.

4. Follow the principle of least privilege.

   The principle of least privilege states that only the minimum privilege necessary to perform an operation should be granted and then only for the minimum amount of time necessary.

5. Compartmentalize.

   The idea behind compartmentalization is to minimize the amount of damage that can be done to a system by configuring it as a collection of modules isolating code that has security privileges. However, compartmentalization must be used in moderation. If each bit of functionality is segregated, then the system may become unmanageable.

6. Keep it simple.

   A corollary might be: limit functionality since limited functionality is easier to protect.

   Simplicity facilitates correctness and maintainability while complexity increases the risk of problems. Unfortunately, the software features for confidentiality, integrity, and authentication tend to increase complexity. Reuse of components such as security libraries and and the use of standard techniques can help to minimize increase in complexity. Multiple layers of defense increase complexity; two layers may be enough. Security can be improved by funneling all security-critical operations through a small number of choke points. A choke point is a small easily controlled interface through which control must pass. There should be no secret ways (administrative back doors) around them.

   Security should be easily available and insecurities should require extra effort. This implies that security should be turned on by default. The following four factors apply to security.

   (a) The user will not read documentation.
   (b) Give users more security than they think that they need.
   (c) Users aren't always right about their security needs.
   (d) Users are lazy.

7. Promote privacy.

   Promote privacy for your users, for your systems, and for your code. Users consider privacy a security concern and the protection of users personal information is necessary. Services running on a machine often give out information that an attacker could use to figure out how to break in. Such information could be removed from the service or misleading information could be supplied instead.

8. Hiding secrets is hard.

   Binary code is not a secure location in which to store secrets such as personal data, keys, and algorithms. This is particularly true for client binaries. Attackers may reverse engineer the binaries to circumvent copy, and access protection.

9. Be reluctant to trust

   Servers should be designed not to trust clients and clients should be designed not to trust servers. Trust is transitive. For this reason, trusted programs should not invoke untrusted programs.

10. Use community resources.

    It is natural to put more trust in publically available algorithms (cryptographic algorithms) and security libraries that have been subjected to repeated use without failure. This is particularly true where there has been public scrutiny of the algorithms and the libraries.

The security design should result in a collection of *security procedures*. Each security procedure is a strategy using various mechanisms to achieve a security policy.

Security Implementation

The elements of the SDL that apply in the implementation phase are:

- Apply coding and testing standards. Coding standards help developers avoid introducing flaws that can lead to security vulnerabilities. For example, the use of safer and more consistent string handling and buffer manipulation constructs can help to avoid the introduction of buffer overrun vulnerabilities. Testing standards and best practices help to ensure that testing focuses on detecting potential security vulnerabilities rather than concentrating only on correct operation of software functions and features.

- Apply security-testing tools including fuzzing tools. "Fuzzing" supplies structured but invalid inputs to software application programming interfaces (APIs) and network interfaces so as to maximize the likelihood of detecting errors that may lead to software vulnerabilities.

- Apply static-analysis code scanning tools. Tools can detect some kinds of coding flaws that result in vulnerabilities, including buffer overruns, integer overruns, and uninitialized variables. Microsoft has made a major investment in the development of such tools (the two that have been in longest use are known as PREfix and PREfast) and continually enhances those tools as new kinds of coding flaws and software vulnerabilities are discovered.

- Conduct code reviews. Code reviews supplement automated tools and tests by applying the efforts of trained developers to examine source code and detect and remove potential security vulnerabilities. They are a crucial step in the process of removing security vulnerabilities from software during the development process.

The technologies (programming language, distributed object platform, operating system, authentication technology) chosen for use in constructing an implementation have security implications. Each technology provides varying degrees of support for good security practices. And may facilitate or impede the construction of various protection mechanisms. The *protection mechanisms* are sets of hardware and software components used to implement any one of different sets of security procedures. The programming languages C and C++ are often chosen for efficiency and performance reason but are notorious as a major factor in buffer overflows which provide opportunities for exploitation by attackers.

Each module should be subjected to a code audit from a security perspective to verify that the security design is properly implemented and that security risks are not introduced through poor coding practices. Short descriptions of the common implementation areas and errors which often compromise security are found in the following subsections.

Confidentiality

An attacker may harvest information of interest from the file system and memory. The attacker may replacing the device driver by one that permits reading arbitrary blocks. The attacker may read preexisting data in memory by using an uninitialized data structure.

Input Validation

Input validation should be a “best practice”. It should be part of standard coding practice. If it were, programs would be less likely to crash and buffers less likely to overflow.

Buffer Overflows

Languages like C and C++ are weakly typed and fail to perform bounds checking. This feature permits a variety of attacks when stack and heap areas are overwritten. In particular, a carefully devised attack permits the execution of arbitrary code. Prevention requires use of a programming language with bounds checking, careful programming practice including input validation, and the use of assertions for checking pre- and post-conditions. A modern language like Java is immune to the problem.

Access Control

Both the Unix access (permissions) model and the Windows access control list (ACL) model provide a mechanism that permits a program to be executed by another user. Careful and knowledgeable use of the access control system is necessary to compartmentalize an attacker.

Race Conditions

Race conditions occur when there are multiple threads or processes that interact. The problems they cause are often intermittent and may difficult to reproduce during testing. Race conditions are often a reliability problem. Prevention requires the use of locking mechanisms to protect shared data structures.

Randomness and Determinism

Since pseudo-random number generators (PRNGs) are in fact deterministic, any security system which depends on random numbers for its security mechanism may be open to attack. Most PRNGs are initialized with a "seed". Use a good pseudo-random number generator (PRNG) and seeds with a high degree of entropy.

Cryptography

Cryptographic methods are useful in meeting security goals. They are used to protect privacy and confidentiality by preventing unauthorized release of information. They are used in authentication to verify the identity of the remote participant. And to insure data integrity, making sure that the message has not been altered.

Trust Management

Good security practice suggests that everything is untrusted by default. And that trust is extended only by necessity. Trust is transitive. Don't trust client code, user input, environment variables, or explicit security modes unless you absolutely must.

Authentication and Data Integrity

Authentication technologies include host-based methods (IP address, DNS name, MAC address, processor id), physical tokens, biometrics, cryptographic methods, and the methodology of defense in depth which uses multiple techniques. System administrators may require photo ID and responses to pre-asked questions to establish user identity. Software may be validated though the use of cryptographic checksums and digital signatures.

Firewalls

A common goal of system administrators is to prevent naïve users from running untrusted code. Firewalls are often the technology of choice. Multiple firewalls may be used. Some to protect from external access and others from internal access. There are two types of firewalls. *Port-based packet filtering* limits users to using administrator selected protocols. An *application proxy* (offers more security) is an intermediary which mimics both the client and the server to each other.

A common goal of application developers is to allow end users to who want to run any application to do so. There are various techniques that may be used to tunnel through a firewall and application developers often create proxy software that may be installed to facilitate tunneling through a firewall.

Database Security

The key issues in database security include, user authentication, encrypted connections (between caller and the database), access control to objects (tables, views, and columns), actions (select, insert, delete, update etc), and privileges (grant). A *view* is a collection of "virtual tables". They are useful for providing access control. Field protection (by encryption) may be necessary to protect data from prying eyes. The side effects include reduced performance and increased difficulty in searching. Statistical attacks enable the attacker to identify an individual associated with a particular row.

Client-side Security

The primary concern of client-side security is limiting software use to legitimately purchased copies. The methods available for providing client-side security are as effective as locks on doors. They help keep honest people honest but do little to detour a determined attacker. The downside of the methods is that, they reduce usability and may decrease performance.

Testing for Security

> *Program testing can be used to show the presence of bugs, but never their absence!* - Edsger W. Dijkstra

Security testing involves probing a system in ways that an attacker might probe it, looking for weaknesses in the software that can be exploited. It is often directed by the system risks identified during architectural-level risk analysis. Code which is unexercised during routine testing may include Trojan horse functionality or serious bugs that can be leveraged into a successful attack.

Software Maintenance and Security

Useful software evolves and in order to stay useful, software must evolve. Distributed systems should be designed with a forced upgrade path. The forced upgrade helps to limit the potential damage should a system be compromised. One approach is for servers to detect when a client should be upgraded and refuse service until the client is upgraded. The upgrade should be obtained using a secure download and automated install or manually.

Security and Software Configuration Management

The software development tree and release packages require protection against access and tampering. In addition to common sense physical security software should be protected with cryptographic checksums and authenticated with digital signatures.

8.4 The Independent Security Audit

A security audit is helpful in validating the security of the architectural design and the implementation. It should be performed by a team that is independent of the development team. The focus of this section is on when to audit, how to audit, and what tools to use.

Security Audit of the Architectural Design

An architectural design audit consists of three phases, information gathering, risk analysis, and reporting. It should be performed after the initial design phase when the key components emerge.

In the first phase, the security audit team examines the requirements and the proposed architecture. The goal is to understand how the system and its environment operates.

The second phase is risk analysis. It must be a methodical way to achieve the goal of identifying the risks, assess the risks, rank the risks in order of severity, and identify countermeasures. The potential that a risk can actually be exploited and the cost of defending against an attack should also be noted.

The method is to build an *attack tree.* First, identify the data and resources of the system that may be targeted. The data and resources are the goals (roots) of the attack tree. The leaves are specific ways of achieving the goal (methods of attack). The intermediate nodes are high-level ways to achieve the goal.

The likely failure points must be identified. These are the modules, communication points between modules, all the classes of users of the system, the computers used, and the networks on which they participate. Using the collected data, the audit team brainstorms possible attacks. The team organizes the attacks into categories and creates the attack tree decorating the tree with estimated cost, estimated risk, and estimated attack effort.

The third phase is to report the findings. The findings are reported in order of perceived risk. The statement of each risk should include a short summary of how serious is the risk, an estimated cost of attack, estimated risk to the attacker, and estimated effort.

Security Audit of the Implementation

There are two phases in an implementation security audit. The first phase attempts to determine whether the implementation corresponds to the design. A reasonable approach is to ask specific questions about the implementation. The second phase attempts to discover implementation specific vulnerabilities.

The strategy for the second phase is first, identify all points in the code that takes input from a user, local or remote, another program, or other untrusted source. These points are network reads, file reads, and GUI inputs. The internal APIs are examined to determine how they handle the inputs. A second item to look for are race conditions; time-of-check/time-of-use (TOCTOU). Some tools that can assist with a source code audit are

- RATS (Rough Auditing Tool for Security). Languages: C, C++, Python, PHP, and Perl. `http://www.securesw.com/rats/`.
- Flawfinder. Languages: C, C++. `http://www.dwheeler.com/flawfinder/`.
- ITS4 (It's The Software, Stupid! [Security Scanner]). Languages: C, C++. `http://www.cigital.com/its4/`

Audited software should contain a prominent notice such as the following:

> *SECURITY NOTICE: This program currently contains no known security problems. If a security problem that could affect you is found, we'd like to tell you about it. If you'd like to receive notification if such a situation does occur, send an e-mail to* my-software-security@mydomain.com. *We will not use your e-mail address for any other purpose than to notify you of important security information.*

8.5 Exercises

1. For each of the following assets (hardware, software, data, communication lines) and each of the security goals (availability, secrecy, integrity, authenticity) determine common threats and suggest protection mechanisms.
2. Classify these methods used in attacking a system.
    - Maliciously modify system software.
    - Introduce vulnerability when adding new features.
    - Install a new program that introduces an unacceptable risk.
    - Accidentally propagate a virus by installing new programs or software updates.
    - Use a Trojan horse to collect passwords.
    - Use a dictionary attack to access a password protected resource.

3. Classify threats as either passive threats or active threats.
4. Classify the following as interruption, interception, modification, or fabrication attacks.
    - Eavesdropping - reading packets in transit.
    - Tampering - modifying packets in transit.
    - Spoofing - generate packets using a stolen identity.
    - Hijacking - take over a packet stream and shut down one of the participating hosts.
    - Capture/replay - copy a packet stream for later retransmission
5. Examine several patterns used in enterprise software for their suitability for use in secure systems.

Chapter 9

Reliability and the Design of Reliable Software

9.1 The Goal: Reliability and Dependability

Table 9.1 places reliability in the context of other quality characteristics identified in ISO 9126 and provides a list of the reliability sub-characteristics.

There are various practices that support the various reliability sub characteristics.

- Maturity: Maintain development across multiple releases.
- Fault tolerance: Use robust methods to protect against permanent failure of a limited number of components. Use stabilizing methods to protect against transitory faults.
- Recoverability: Use recovery oriented methods.
- Compliance: Comply with relevant standards and best practices.

These practices will be covered in more detail later.

Dependability is related to reliability and the SWEBOK lists the following as characteristics of dependability.

Reliability includes:

Maturity
Fault tolerance
Recoverability
Compliance

Figure 9.1: **ISO Standard 9126 Software Reliability Quality Characteristics**

- Availability - the product's readiness for use on demand
- Reliability - the longevity of product performance
- Maintainability - ease of maintenance and upgrade
- Maintenance support - continuing support to achieve availability performance objectives
- Trustability - system's ability to provide users with information about service correctness.
- Additional attributes - fault tolerance, safety, security, usability.

Table 9.2 suggests practices that support the various dependability subcharacteristics.

9.2 Faults

The goal of reliability is necessary because in the real world there is an abundance of system faults, operator errors, bad data and buggy code. Table 9.3 lists the common sources of errors and types of errors. Some errors may be safely ignored. Others errors require prevention through good practices (bombproof, robust, fault-tolerant, fail-safe, defensive programming). To the user, all errors appear to be abnormal termination or an unexpected action of the the application or system. In spite of our best efforts to anticipate and attempts to prevent faults, faults will occur and therefore require methods to detect and recover from errors (recovery oriented computing).

There are two approaches to minimize operator error. Improved user interfaces and the automation of as much of a process as practical help reduce the opportunities for operator error.

9.3 Fault Detection, Diagnosis, and Handling

A software system is a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations. The specifications are the provisions of the contract and are called *assertions* and are expressed as boolean expressions. The specifications of the mutual obligations constitute a contract. The enforcement of the contract is accomplished using the notion of an assertion. An *assertion* is a boolean expression about the state of a software system. Assertions are inserted into the code and are checked during the system's execution. If any assertion evaluates to false, the system is invalid.

There are several kinds of expressions – preconditions, postconditions, invariants, and progress condition. The specification of each component includes a precondition and a postcondition. The *precondition* specifies the obligations that a user of the component must meet before the component may be used.

Value	*Description*	*Design and Implementation Techniques*
Availability	Readiness for use on demand	
Reliability	Longevity of product performance	
Maintainability	Ease of maintenance and upgrade.	
Maintenance support	Continuing support to achieve availability performance objectives	
Trustability	Ability to provide users with information about service correctness and type and location of failure.	RAID, Backups, Log structured file system.
Fault tolerance	Either guarantee continuous correct behavior or suspend operation until repairs can be completed.	Use robust methods to protect against permanent failure of a limited number of components. Use stabilizing methods to protect against transitory faults.
Safety	Insure that nothing bad happens.	Use assertions. Use two phase commit. Use Log structured file system.
Security	Protection against unauthorized disclosure, alteration, or destruction of data and protection against unauthorized users.	Use authentication, protection, and encryption.
Usability	The user interface is expected to be intuitive, to be easy to use, and to allow the users to maximize their efficiency and effectiveness when using it.	

Figure 9.2: **Dependability Values (SWEBOK)**

Bad data ... • Incorrect type • Out of range • Missing item • Non-existent file	**Operator error ...** • Misconfiguration • Failure to take required action • Improper action or incorrect command • System upgrade
System faults ... • Hardware faults • Network faults – Messages ∗ delayed ∗ lost, ∗ corrupted, ∗ reordered – Remote system unreachable • Distributed system faults – Initially dead process – Crash model – Omission failure – Byzantine behavior (malign failure) – Timing error (in synchronous systems only)	**Buggy code ...** • Due to specification error – specification incomplete – specification ambiguous – specification misunderstood • Due to design error (see antipatterns) – Out of memory condition – Loop index off by one. – Buffer index out of bounds. – Infinite Loops – Division by zero and other arithmetic errors – Undefined variables – Missing special case, e.g. head of empty list – Missing synchronizing element, e.g. missing send – Unintended side-effects – Failure to maintain an invariant. – Failure to satisfy pre-condition – Failure to satisfy post-condition – Argument out of range or incorrect type – Failure to include exception handling (and instead letting the system crash or simply displaying an error message).

Figure 9.3: **Sources and types of faults and errors**

Error detection	Assertions Log files try ... raise ... except ... ping ps send queries and test messages: ping, ps, etc
Error diagnosis	Send queries and test messages: ping, ps, etc to determine whether the process is alive, running, or functioning correctly.
Error handling	stop restart

Figure 9.4: **Idioms for fault detection, diagnosis, and handling**

The expression may include the system state and the arguments passed to the component. The failure of the precondition means that the source of a fault is in the user. The *postcondition* specifies the condition that holds after the execution of the component. It is a guarantee of service provided the precondition is satisfied. The expression may include the old system state, the arguments, the new system state, and returned values. The failure of the postcondition means that the source of the fault is in the component. The *invariant* specifies a condition that must hold at any time. Loop invariants hold on entry to a loop and on each iteration of the loop (on entry and exit of a recursive function). Data invariants hold both before and after access to a data structure or object. The failure of the invariant means that a fault may exist in either the component or in the user of the component. The *progress condition* specifies the condition under which a loop or a recursive subprogram makes progress toward termination. Progress conditions are rarely used as assertions but should be explicitly noted during program construction and debugging.

Some programming languages provide support for assertions.

9.4 Fault-tolerance

Fault-tolerant hardware systems handle faults through through redundancy and algorithm support. The outputs of the redundant components are compared, evaluated, and the correct output selected. Transaction systems which may consist of multiple components (not necessarily redundant components) rely on a software algorithm to maintain the integrity of transactions in the presence of hardware failures. Software faults may be handled through redundancy using different but functionally equivalent modules. As with hardware, the outputs of the modules must be compared, evaluated, and the correct output selected. These tasks are performed by an adjudicator. There are two approaches to fault tolerance, robust systems and stabilizing systems.

Robust Systems

A *robust system* is one which guarantees the continuous correct behavior of the system in spite of failures occurring during its operation. Robust systems protect against *permanent failure* of a limited number of components. *Robust algorithms* are designed to guarantee the continuous correct behavior of correctly operating processes in spite of failures occurring in other processes during their execution.

Stabilizing System

A *stabilizing system* is one in which any number of failures of arbitrary types is allowable, but correct behavior of the system is suspended until some time after the repair of the failures. Stabilizing systems protect against *transitory failures*, i.e., temporary misbehavior of system components. *Stabilizing algorithms* are designed to handle any number of failures of arbitrary types, but correct behavior of the algorithm is suspended until some time after the repair of the failures.

9.5 Robust Systems

The robust approach is based on the mantra: *the best defense is a strong offense.* The approach requires fault detection and action to handle the fault. Handling the fault may range from displaying a message and terminating to extensive attempts to remove the fault and continue execution. The system is required to handle bad (invalid or absurd) inputs in a reasonable way and if an internal error occurs, the program terminates gracefully and provides enough information for the programmer to debug the system.

Error Handling Facilities in Programming Languages

Process

- Detect an error.
- Stop the computation and report the source of the problem
  `error "error message"`
- Use dummy values. For example, define the tail of a list to be the empty list.

- Pass a dummy value to be used as an error value. For example, pass a value to the function which extracts the head of a list. The value is used only when the list is empty.

Error Handling

Once an error has been raised

- *transmit* the error through to the next higher routine
- *trap* the error and return

The Patterns

Name:	Call-Error-Function-and-Quit
Example:	error "argument to factorial is negative"
Name:	RaiseError
Example:	throw ExceptionName (argument)
Name:	TryBlock
Example:	try Code with raise exception except exception handler

Fault-tolerant algorithms

When there are N functionally equivalent components, there must be a method to identify the correct response thus identifying the faulty component(s).

- N-version method - select using
  - majority voting
  - consensus voting
  - maximum likelihood voting
- Recovery Block - select output of first component whose output passes an acceptance test.
- Hybrid techniques
  - Consensus recovery block
  - Acceptance voting
  - N-self checking programming

9.6 Stabilizing Systems

A study of Internet problems reveals that operator error is the cause of 51% of the problems followed by software and then hardware. The motivation for stabilizing systems is based on the recognition that, in spite of best efforts, failures will occur and therefore focuses on recovery.

Self-maintaining software

Software systems are installed, maintained and removed. In the first phase of service the software is

- loaded onto the system
- configured and
- made available.

In the second phase of service the software may be

- reconfigured
- patched and
- upgraded.

While in service the software may

- generate error messages and fault reports and
- generate log files which trace its activities.

At the end of its service life a software system is retired. Tools are often required to assist with these administrative tasks.

Recovery Oriented Computing

The goal of Recovery-Oriented Computing is to improve system dependability. It is a generalization of the stabilizing approach. The motivation for ROC comes from a study of Internet problems which reveals that operator error is the cause of 51% of the problems followed by software and then hardware. Rather than building failure proof systems, ROC-solid computing recognizes that failures will occur and focuses of rapid recovery.

The ROC approach takes the following three assumptions as its basic tenets:

- failure rates of both software and hardware are non-negligible and increasing
- systems cannot be completely modeled for reliability analysis, and thus not all of their failure modes can be predicted in advance
- human error by system operators and during system maintenance is a major source of system failures

The four principles behind ROC-solid computing are:

1. **Speedy Recovery:** the system must be designed for speedy recovery. Isolation is crucial for fault containment and safe online recovery, and it is an enabler for many diagnostic and verification techniques. Isolation naturally demands redundancy, as redundancy allows continued service delivery while portions of the system are isolated. The recovery mechanisms must be reliable, effective, and efficient.

   Example: use log structured file systems to shorten reboot times.

2. **Pinpoint the source of faults:** operators need the tools to enable them to pinpoint the sources of faults. Isolation is crucial for fault containment and it is an enabler for many diagnostic and verification techniques. All modules in the system should be self-testing, and should verify the behavior of all other modules that they depend upon. The components should cooperate to track dependencies between modules, resources, and user requests.

   Example: share configuration information and log files

3. **Undo function:** An undo function is necessary to permit operators to correct their mistakes. It provides a three-step process of *rewinding* time, *untangling* problems, then *replaying* the system back to the current time. An undo facility should cover all aspects of system operation, from system configuration to application management to software and hardware upgrades.

   Example: make a copy of the current configuration prior to making a change.

4. **Inject test errors:** operators need the ability to inject test errors to permit the evaluation of system behavior and to train operators in the detection and correction of errors.

   Example: send invalid information to an application to determine whether it is robust or whether it will correctly react.

The ROC principles may be used in the software development process. The source of faults is more easily pinpointed buy using small modules and assertions. The use of a version control system to manage the code base provides an undo function. The use of a test harness and regression testing permits developers to inject test errors into the system. Speedy recovery from faults (quick recognition and repair of faulty code) is facilitated by the use of small modules, assertions, and a test harness.

See `http://roc.cs.berkeley.edu` for more details.

9.7 Leverage Points in Dynamic Systems

A system is in a certain state, and contains a stock, with inflows (amounts coming into the system) and outflows (amounts going out of the system). At a

$$\begin{tikzcd}
 & \arrow[r, "\textit{Control}"] & {} & \textit{Inflow} \arrow[d, Rightarrow] \\
\boxed{\text{Discrepancy} = \text{Goal} - \text{Preceived State}} \arrow[ur, dash] \arrow[dr, dash] & & & \boxed{\text{System State}} \arrow[lll, Rightarrow] \arrow[d, Rightarrow] \\
 & \arrow[r, "\textit{Control}"'] & {} & \textit{Outflow}
\end{tikzcd}$$

Figure 9.5: **Dynamic Element**

given time, the system is in a certain perceived state. There may also be a goal for the system to be in a certain state. The difference between the current state and the goal is the discrepancy (Figure 9.5).

Levers are places within a complex system where a “small shift in one thing can produce big changes in everything”. Donella Meadows[1] proposed the following scale of general leverage points (in decreasing order of effectiveness) to intervene in a system:

1. World view: The power to transcend paradigms. No paradigm is right, you can choose whatever one will help you to achieve your purpose.

2. Paradigm: Paradigms are the sources of systems. Systems (their goals, information flows, feedbacks, stocks, flows) are developed from shared social agreements about the nature of reality.

3. System level goals: The goals of the system are parameters that can make a big difference. Survival, resilience, differentiation, evolution are system level goals. The software quality characteristics are system level goals.

4. Self-organization: Self-organization is the power to add, change, evolve, or self-organize system structure – changing any aspect of a system lower on this list. Self-organization is necessary for the emergence of intelligence.

5. Rules: The rules of the system (such as incentives, punishment, constraints) define the scope of the system, its boundaries, and its degrees of freedom. The malfunctions of a system may be understood by identifying the rules and who has power over them.

6. Information flow: The structure of information flow (who does and does not have access to what kinds of information). Adding or restoring information is much easier than rebuilding the system.

7. Feedback loops

   (a) *Self-reinforcing feedback loop* (positive feedback): They are sources of growth, explossion, erosion, and collapse in systems. Chaos occurs

[1]from Donella Meadows “Leverage Points: Places to intervene in a system” `http://www.sustainabilityinstitute.org/pubs/Leverage_Points.pdf`.

when a system starts changing faster than its negative loops (see next) can react to it.

(b) *Self-correcting feedback loop* (negative feedback): They are controls to keep system state within safe bounds. The strength of negative feedback loops must be relative to the effect they are trying to correct against. They are often used in safety systems.

(c) Delays: Delays in feedback loops are a common causes of oscillations.

8. Structure: The structure of material stocks, flows, and their relative arrangement can have an enormous effect on how a system operates.
9. Buffers: Stocks which are large relative to their flows are more stable than small ones while small ones are more flexible.
10. Parameters: Constants and parameters rarely change behavior.

Levers 1-3 are structural levers affecting multiple subsystems. Levers 4-9 are dynamic levers which produce changes over time. Levers 10-12 are direct levers (cause and effect) and have immediate effect.

9.8 Self-aware Systems

IBM's autonomic computing initiative has identified these eight elements of (or goals for) autonomic systems.

1. *Self-awareness*. An autonomic computing system needs to "know itself" - its components must also possess a system identity. Since a "system" can exist at many levels, an autonomic system will need detailed knowledge of its components, current status, ultimate capacity, and all connections to other systems to govern itself. It will need to know the extent of its "owned" resources, those it can borrow or lend, and those that can be shared or should be isolated.
2. *Self-configuring*. An autonomic computing system must configure and reconfigure itself under varying (and in the future, even unpredictable) conditions. System configuration or "setup" must occur automatically, as well as dynamic adjustments to that configuration to best handle changing environments.
3. *Self-optimizing*. An autonomic computing system never settles for the status quo - it always looks for ways to optimize its workings. It will monitor its constituent parts and fine-tune workflow to achieve predetermined system goals.
4. *Self-healing*. An autonomic computing system must perform something akin to healing - it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction. It must be able to discover problems or potential problems, then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly.

5. *Self-protecting.* A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection. It must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.

6. *Self-adapting.* An autonomic computing system must know its environment and the context surrounding its activity, and act accordingly. It will find and generate rules for how best to interact with neighboring systems. It will tap available resources, even negotiate the use by other systems of its underutilized elements, changing both itself and its environment in the process – in a word, adapting.

7. *Function in a heterogeneous environment.* An autonomic computing system cannot exist in a hermetic environment. While independent in its ability to manage itself, it must function in a heterogeneous world and implement open standards – in other words, an autonomic computing system cannot, by definition, be a proprietary solution.

8. *Anticipate future requirements.* An autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden. It must marshal I/T resources to shrink the gap between the business or personal goals of the user, and the I/T implementation necessary to achieve those goals – without involving the user in that implementation.

Sun Microsystems has a less ambitious initiative, N1, whose I-Fabric Architecture has three layers, control, communication fabric, and resource. A self-aware entity can

- sense events,
- direct its attention to specific events (selectively sense events),
- generate internal and external events,
- distinguish between internal and external events,
- distinguish between external events generated by self and non-self,
- recall past events and their contexts, and
- anticipate future events.

A more sophisticated self-aware entity may be able to plan and evolve (learn). Self-awareness may be built into software systems using a monitoring component and a functional component. Here is a design pattern for self-aware systems that could have several of the elements of IBM's autonomic systems. [2]

Name	Self-aware system
Context	complex, unreliable

[2] Consider inserting in here a discussion of the Goedelian machine at `http://www.idsia.ch/~juergen/goedelmachine.html`.

Problem	Need a system which is dependable in spite of faults
Solution	Build a self-aware system which consists of monitor processes and functional processes. The monitor processes monitors feedback from both the functional components and the environment and can direct the functional components to take action. The functional components respond to signals from the monitor processes, provide feedback to the monitor processes and interact the environment. • The monitor process is able to start, stop, and restart individual functional components. • The functional components generate log files of their internal states. • The monitor processes have access to the log files generated by the functional components and is able to construct usage scenarios. • The monitor processes are able to perform diagnostic tests on the components.
Related patterns	Architectural patterns: Layers, Blackboard. Design patterns: Memento Pattern, Observer Pattern

9.9 High Availability Systems

To be completed

9.10 Exercises

1. Construct a classification of operator errors. Consider all types of users from systems administrators to casual users of applications.
2. For each type of operator error, suggest a robust solution to the error and a stabilizing solution.
3. Construct a classification of faults or problems that may occur during the installation and service phase of a software system.
4. For each type of installation and service fault or problem, suggest a robust solution and a stabilizing solution.
5. Design the general forms of the error messages associated with preconditions, postconditions, and invariants.
6. Using either a library function, your own version of an assert function, or built in error handling routines, instrument a program of your own choice. The errors should be written to a log file. The program must include loops, subprograms, and objects with complex data structures which require invariants.

7. What are the methods used by transaction systems to provide database integrity?
8. Analyze a three-tier database driven website from the point of view of ROC. Indicate what must be done to provide the highest levels of reliability.
9. Apply ROC techniques to the design of a computer network which includes DNS, DCHP, and mail servers. Use the self-aware pattern and construct a self-aware web server. item Construct an architectural pattern (and if necessary, some design patterns) to support autonomic computing.
10. Compare ROC and Service oriented architecture.

Chapter 10

Usability and the Design of the User Interface

Good tools background themselves allowing users to focus on the task at hand.

Users engage in three basic activities, knowing, changing, managing. Users differ in perceptual and psychomotor abilities. There are several types of users, first time, intermittent, and expert. Users do not like to read. Users forget things. Users get into trouble (make mistakes, get lost). Users want to maximize their efficiency and effectiveness. The values of the users drive the feature set and the user interface design activities.

Henderson identified six dimensions along which to describe the activity in which people take part while making use of technology. His dimensions provide a helpful description of the context for usability requirements. The six dimension fall into two groups. The first answers the question as to what must be done. The second answers the question of why it should be done. The "What" group is further divided into enabling activities and operating activities. The "Why" group consists of empowering activities. The enabling activities are the dimensions of support and practice, the operating activities are dimensions of users and trouble, and the empowering activities are the dimensions of values and designers. Each dimension gives rise to one or more requirements. Figure 10.1 lists the six dimensions, provides some additional detail, and suggests how each impacts the requirements and design.

Usability is a measure of the quality of a user's experience interacting with a product or web site. It involves users' ability to do what they want and need with the product or site. There are four main questions about the usability of a product or web site:

1. How effectively and efficiently can users accomplish their goals? Usable products and sites support users in accomplishing their goals, improve productivity by allowing users to do so quickly, efficiently, and accurately,

What must be done ...

1. Enabling activities

   - *Support:* Users engage in three basic activities, knowing, changing, and managing.

     Requirements: Provide support for the activities associated with knowledge, changes, and resource management.

   - *Practice:* Genré, Routine, Morés

     Requirements: Provide support for genrés, routines, and morés.

2. Operating activities

   - *Users:* There are many different types of users.

     Requirements: The interface should be intuitive for the novice and uninitiated, and efficient in perception and motion for the expert.

   - *Trouble:* Users get into trouble.

     Requirements: Prevent users from getting into trouble. Help users get out of trouble.

Why it should be done ...

3. Empowering activities

   - *Values:* The values of the user drive the activities.

     Requirements: Make it clear how the product provides value to the user.

   - *Designers:* Users drive the ongoing design.

     Requirements: Provide a mechanism for user feedback.

Figure 10.1: **Henderson's Six Dimensions of Activity**

and are designed to reduce user errors and to give users easy ways to recover from any errors they do make.

2. How easy is it for new users to learn how to use the product or site? Usable products and sites allow new users to get up to speed and accomplish meaningful work in a relatively short period of time, match the way in which users approach their tasks, are easy to remember, require less training, and allow users to return to peak levels of performance more quickly after absences.

3. How easy is it for experienced users to use the product or site? Usable products and sites are designed to require a minimum number of keystrokes or mouse-clicks, allow experienced users to customize actions and streamline processes to maximize their efficiency, present information in ways that support rather than hinder decision making, and allow users to move easily from one task to another

4. How much do users prefer the system to other systems? Usable products and sites leave users feeling productive and satisfied when they're done and can reduce staff turnover and absenteeism by making users feel that they are adequately supported in their jobs.

Usability is one of the software quality characteristics identified in the ISO Standard 9126 (Figure 10.2). The Standard places the focus on the characteristics, values, and needs of the user rather than requiring the user to conform to the needs of the computer.

The *User-Centered* approach to design (UCD) and assessment focuses on the mindset of target users. This can include:

- Users' prerequisite knowledge and skills
- Users' goals and objectives (which are often different from, and in some cases in opposition to, that of authors, designers, and/or programmers of a product or site).
- Users' reactions to getting lost or frustrated or being unable to accomplish their goals.

Accessibility is a pro-active design quality that attempts to provide a usable interface for all users regardless of disability. Rather than a design annoyance, attention to accessibility issues provides improved interfaces for all users. The W3C Web Accessibility Initiative (`http://www.w3.org/WAI`) provides guidelines, checklists and techniques for the design of user interfaces to satisfy accessibility requirements.

Human Factors is the science of making systems safe, effective, and easy to use. The psychology of human factors is the study of:

- Human capabilities and limitations
- How humans interact with technology

Usability includes:

- *Understandability:* A good conceptual model with a clear locus of control.
- *Learnability:* Consider the needs of novice and uninitiated users.
- *Operability:* Develop scenarios and use cases for novice, uninitiated, and expert users. Provide navigational efficiency - users can locate the information they want.
- *Attractiveness:* Layout, graphics, color, dynamic elements.
- *Compliance:* Follow standards and accepted practices.

Figure 10.2: **ISO Standard 9126 Software Usability Quality Characteristics and Design Hints**

- How the relationship between humans and technology can be optimized.

Among the goals of human factors psychology are to:

- Make it easier and safer for people to use products such as computers, cars, and cell phones.
- Improve home, work, and leisure environments.
- Help engineers design systems and products that work while accommodating the limits of human performance and abilities.

Human factors engineering (HFE) is a crossover discipline that combines psychology and engineering with computer science, software engineering, and others to explain how people behave in the real world. Based on the research from their multi-disciplinary field, human factors engineers try to design systems and products to fit people, rather than the other way around, because it is easier to design system characteristics than it is to change human characteristics.

Usability evolved from the human factors focus on users and products that are "user-friendly". *Usability Engineering* is the hard work that makes a product or web site "user-friendly". It is a methodical and practical approach to product or site design and development. The process can involve:

- gathering information about intended users and their needs
- developing and testing prototypes
- evaluating design alternatives

- analyzing usability problems
- examining solutions
- redesigning
- user testing.

Usability Evaluation is a part of the usability engineering process. Trained usability experts study the behavior and responses of users by applying a variety of evaluation methods and report the findings so that designers can apply the results to improve the product or site.

10.1 User Interfaces

The user interface is composed of methods and devices and facilitates two-way communication between the user and the machine. The user interface is expected to be intuitive, to be easy to use, and to allow the users to maximize their efficiency and effectiveness when using it. The focus here is on user interfaces that include a graphical display and various input devices – the devices in a graphical user interface (GUI). Rather than requiring the user to memorize commands, a GUI presents all possible options to the user in a visual format. At present, the design of intuitive interfaces is more an art than a science though it draws on psychology and cognitive reasoning. Good tools so integrate the product into the user's values and activities that the activity rather than the tool is the focus of the user.

Human User Interface

The design of the human interface must draw on psychology and cognitive reasoning and cater to the needs of different types of users including the first time, intermittent, and expert users. Catering to the needs of different types of users may require a multi-layer interface [75].

The SWEBOK[86] identifies adaptivity, intelligent help, and display salience as the key components of an interface for human-computer interaction. Figure 10.3 is an elaboration of these ideas including architectural hints. Bruce Tognazzini [88] has an excellent list of principles for interaction design.

The usual GUI interface consists of a display, a keyboard, and mouse. There is also the possibility of video input and audio output.

Examples of Usability Heuristics (Guidelines)

Some examples of usability guidelines (adapted from Jakob Nielsen, Usability Engineering, 1993) are:

- Visibility of System Status - Users should always know where they are and what's going.

Quality Factors	**Adaptivity**	**Intelligent Help**	**Display Salience**
Quality attributes	Intuitive for the novice. Efficient in perception and motion for the expert. Rich in incrementally learnable functions, like the alphabetical shortcuts on the Mac interface	Robust under misuse. Facilitates recovery from cognitive or manipulative mistakes. Helpful in diagnosing errors and suggesting corrective action.	Simplicity, Elegance, Clarity, Order, and Consistency
Architectural considerations	Provide a bug report and new feature request facility. Keep the interface simple and consistent. Provide both GUI and command-line interfaces. Provide shortcuts.	Prevent errors: Keep the interface simple and consistent. Reduce short-term memory load. Offer informative feedback. Provide help Correct errors: Provide an undo function.	Simplicity, consistency, contrast, and layout. Feedback: Dialogs should yield closure. Appropriate technology Good mappings A common look and feel between applications and pages on a website.

Figure 10.3: **Human Interface Quality Characteristics**

- Flexibility and Efficiency of Use - Users should be able to customize the execution of frequent actions and to control the speed of interface interactions whenever possible.
- Aesthetic and Minimalist Design - Rarely needed information should be easy to access but hidden from view. Screens should contain only that information necessary to use them successfully. This is a variation on the KISS principle - Keep It Short and Simple.
- Error Prevention - Products should be designed to prevent errors. It is better to design to avoid errors than to provide good error messages.
- Consistent "Look and Feel" of the Interface - Every screen should be recognizable as a particular product. For example, there should be a common layout, background and/or color scheme for all screens.

Web Site Interface

The characteristics of target users is more difficult to define for web site evaluations. Because of the wide range of web site user options for hardware, software, browsers, connectivity, bandwidth, and screen configurations, users can have vastly different experiences with the same site. The usability of web sites differs from the usability of other software products or tools only in the order of importance of the usability issues (Jakob Nielsen). For example:

- Ease of learning is often the most important usability issue, since users rarely spend enough time on any individual web site to make ease of use the primary concern.
- The subjective measure of user satisfaction or preference is a more critical issue since users rarely have any real investment in a particular site and other options are readily available and easily accessible.
- Reducing user errors is more critical for Extranet, especially e-commerce, sites than is efficiency of use.
- Efficiency of use is more critical to Intranet sites because, by their nature, such sites are used repeatedly by the same users for the same purposes, usually business.

For web sites, the usability characteristics may be better expressed in terms of effectiveness, affectiveness, and navigational efficiency.

Effectiveness presentation should be complete, sensibly organized, and accurate in conveying the intended message. This includes the technical specifications.

Affectiveness interesting, stimulating, and enjoyable (appealing to the user). This covers the look and feel of the site/page.

Navigational efficiency the ease with which users can locate the information they want. This includes the structural principles followed.

Graphical User Interface (GUI)

Aesthetics - what separates a great chair from an ordinary chair is not its primary functionality but its nonfunctional qualities and its aesthetics.

Basic Visual Design Elements

While all visual design elements are derived from the geometric elements (points, lines, planes, shapes, focal point), scale, texture, value, color, and space, the GUI design community has developed a collection of widgets to facilitate the creation of interactive interfaces. Widget lists usually include:

- Pull-down/Drop-down Menus - dynamic expansion of an item
- Push Buttons - A labeled or iconified object that responds to clicks or enter
- Icons - represent objects or used as metaphors for functions or actions. Often “clickable” as with buttons.
- Checkboxes -
- Radio buttons -
- Scrolling lists -
- Text Field -
- Popup List -
- Spin boxes -
- Sliders -

The placement of the widgets to facilitate attractiveness and display salience require attention to design values such unity, harmony, contrast, proportion, rhythm, balance, and visual illusion. These may be condensed into the four graphic design values of

1. Contrast
2. Repetition
3. Alignment
4. Proximity

Command Line Interface (CLI)

nt needed

Alternate names: conversational interface, dialog interface

The Unix command line interface (CLI) provides five features that map well to the user's world view. The first is that it supports dialog. Second, it gives the user natural control over tasks as the user chooses which tasks run in the background and which is the task of current focus. Third, it allows users do discover features of the space in a natural manner. The `man` command, the command option `--help` or `-h` and Tab-completition of commands facilitate discovery. ... see Richard Wareham's "The Command Line - The Best Newbie Interface?"

10.2 Architectural Support for Usability

The usability and interface architecture includes both a graphical component and a task structure. With the increased prominence of the Web, the evolution of the browser toward the user interface of choice, it is not necessary to distinguish between application oriented GUI design and web design.

- Application interface. While primarily designed for interaction it may also include browsable browsing features such as directories and online help systems.

- Web site interface. While primarily designed to provide access to information it may also contain interactive elements such as an on line store.

Every web site and page consists of two fundamental elements, information objects and an information structure (the relationship between information objects).

- Content - the information objects consisting of text and multimedia which are obtained by a process of information decomposition which identifies the component parts to be presented on the page or the site.

- Linkage - the information structure which maps the relationships between the information objects

The architectural design elements include

- Organizing metaphors and idioms e.g.,

- Window/Page - which may represent a view into the computer

- Dynamic and reactive elements associated with events - functions initiated internally or by the user

- Linked information structures

It is easier to direct a user's attention when the page layout that is clean and uncluttered. Items should be positioned according to importance and commonality (related items should be clustered together). In western society, attention begins in the upper left corner and moves to the lower right. The alignment of elements is important. Verticals and horizontals are to be preferred over jagged lines. Topography (font family, character style, font size), graphics, and color should be chosen to achieve the design objectives.

Task structures are organizational schemes which support a type of task. Some common task structures and information structures are

- Linear - A linear sequence of tasks.
- Tutorial - Instruction and evaluation followed by either remedial instruction or advancement to the next module.
- Web structure - (all interlinked)
- Hierarchical - The common *whole-part* or *top-down* structure.
- Cluster - Collection of interrelated activities.
- Catalog - An index.

The links provided should facilitate site navigation. A site of a thousand pages arranged in a hierarchical structure with ten links per page requires only three clicks to reach any page from the home page. When information requires more space than is provided in a single window there are three options.

1. Avoid Navigation: Expand the window/page
2. Limit Navigation: Navigate within the same window/page
3. Facilitate Navigation: Navigate from one window/page to another

10.3 Usability Engineering Process

From a user-centered perspective, usability evaluations should start early in the development process and occur repeatedly throughout the design cycle, not just when the product is completed.

Usability testing methods can be adapted for any of the general phases detailed below in the software or technology product development process. Usability evaluation at each phase is a critical part of ensuring that the product will actually be used and be effective for its intended purpose(s).

The Usability Engineering Life Cycle goes along with the software being developed:

1. Requirements Analysis * The end user as well as secondary users are considered, and user profiles are conceived. * Task analysis reveals task requirements and the current workflow as well as specific conditions of use. * Platform capabilities and constraints are recognized at this time. 2. Conceptual Model

Operational Qualities	**Maintenance Qualities**
• Functionality: *Simplify* - Do one thing well, Don't generalize, Get it right. *Corollaries* - Don't hide power, Use procedure arguments, Leave it to the client. *Continuity* - Keep basic interfaces stable, Keep a place to stand. • Reliability: End-to-end: Log updates, Make actions atomic • Usability: • Efficiency: Make it fast: Split resources, Static analysis, Dynamic translation	• Maintainability: • Portability:

Figure 10.4: **Software Quality Values and Lampson's Interface Design Method**

Design includes reengineering the workflow and writing down the application structure. 3. Testing and Debugging in HCI means designing rapid prototypes and user test them. 4. Implementation of the "final" system, delivery and installation. 5. Evaluation and refinement of the requirements lead to an update of the product, repeating the usability engineering life cycle.

The usability engineering process consists of the same phases found in any software engineering process but with greater emphasis placed on user input during the requirements elicitation, design, and testing phases. Use cases, scenarios, and prototypes developed with the participation of users are particularly helpful for requirements elicitation and in the design phase.

Lampson offers the advice in the table: Figure 10.4.

`http://www-128.ibm.com/developerworks/library/it-nielsen3/`

The one thing that works for creating usable systems is a full usability engineering lifecycle that corrects the quality of the design at every single step of the way.

Here is the lifecycle I recommend, divided across the three main stages of a development project:

Requirements Phase

- Conduct field studies of the ways your users currently approach the problem. Not that you have to do exactly that in the software, but you want to know what people actually do in real life and not what the procedures manual claims that they do.

- Run a usability test of the old system. If one is in place, don't just discard it. Much can be learned from seeing its strengths and weaknesses exposed in a study.

- Conduct competitive studies of any other solutions on the market. Instead of relying on the hearsay you get from reading industry analysis, collect real data about what actually happens when real people use these other systems. You will often find that some of their highly touted features don't work very well, but that you have opportunities for doing things in a much easier way.

Design Phase

- Start with a parallel design exercise: Explore the design space by making up diverging designs that solve the problem in many different ways. Collect a small amount of user test data on each approach, even if it is only "implemented" as a few screen mock-ups or paper drawings. Decide on the best approach for going forward. (Usually, this will involve elements from several of the parallel designs.)

- Iterative design: Evolve the chosen design direction through several rounds of usability feedback, adjusting it each step of the way. The more usability data you can incorporate into the design, the better.

- Prototyping: Gradually evolving from low-fidelity prototypes in the early stage of the project to high-fidelity prototypes in the later stages. Even very low-fi prototypes can be tested with users, and even if you don't learn quite as much as when testing hi-fi, you can do so much earlier when:

  - It costs less to make changes.
  - You are not so emotionally invested in the preliminary solution because you have not worked on it for very long.
  - You still have time to act on the findings and change the architecture of the solution.

Maintainability and portability requirements are best supported when there is a separation of the user interface from the rest of the application. A useful organization is that of a three-tier architecture which consists of a

- Presentation layer, an

- Application or business layer, and a

- Database layer.

Verification

Usability testing methods yield both qualitative and quantitative data that can be used to continuously improve the quality of the product. Usability evaluation can yield information about:

- How users do things. For example, the paths users take to accomplish their goals.
- The errors users make and the criticality of those errors.
- When and where users get frustrated or confused.
- Why users get frustrated or confused.
- How quickly users accomplish their goals.
- How successfully users accomplish their goals.
- How satisfied users are with their experience.
- Possible solutions to the problems users encounter.

When the primary focus is on the user-interface, usability evaluations address:

- the physical features of the user-interface, for example, buttons and button placement, graphics, text, layout, etc.
- the functions of the software (program or web site), for example, loading, saving, printing, editing, navigating, setting parameters, etc.

When the primary focus is on user performance, usability evaluations address:

- the mental states of the users. Are users, for example, confused? Frustrated? Ignorant? Hostile? Misinformed? Rapt? Motivated?
- the integration of the software into institutional processes. Have users, for example, been properly trained on the software/technology product? Do they know what they are expected to accomplish using it? What, if any, are the barriers to the successful use of the product in the environment for which it is intended?

In usability testing, if learnability is the primary goal, then uninitiated users should be used. If the goal is ease of use, then novice users should be used.

Support and Servicing

You may think that once your project ships, you are done with this pesky usability work. No way. More steps:

- Collect statistics and feedback from real use: check log files in case of a Web site; interview real customers about how they are actually using the software. (No matter how careful you were in the pre-design phase, there may be some new uses that you did not predict.)

- Refresh: In case of a Web site, you can take advantage of having direct control over the running code on the server and fix any immediate problems your post-design studies reveal. Such changes should be in the nature of tweaks to the original design and not a full-fledged redesign.

- And at some point of time you need to start planning for an actual redesign: remember that your current release is your best prototype of the next release, and start planning for the next release while you still have enough time to do a good job.

The model may seem extensive, but many of the steps are only a matter of one or two days' work.

Doing things right will only add a few percent to the cost of a development project. You will save many times this cost by not having to make expensive adjustments and dot releases. Plus, the resulting user interface will probably be around 50% to 100% easier to use, reducing training budgets dramatically and increasing user productivity. If you happen to be running an e-commerce site, you will have the sweetest gains of all: customers will finally start making purchases now that they can find what they want on the site. Rule #1 of e-commerce: if you can't find it, you can't buy it.

10.4 Exercises

1. Read *First Principles of Interaction Design* (`http://www.asktog.com/basics/firstPrinciples.html`). What did you learn? What surprised you?

2. Reread *First Principles of Interaction Design* (`http://www.asktog.com/basics/firstPrinciples.html`) from the perspective of a software developer involved in the development or maintenance of a large system. Based on your reading what are some *best practices* that should be followed to simplify your development and maintenance tasks?

3. Read *Maximizing Human Performance* (`http://www.asktog.com/basics/03Performance.html`). What did you learn? What surprised you?

4. For each of the ISO Standard 9126 usability quality characteristics, explain what human characteristics motivate the quality.

5. Using Henderson's dimensions of activities, evaluate a user interface for an application program.

6. Select two web sites (e.g., two universities, two Fortune 500 companies) and analyze the structure and content categories presented on their web site. What are the similarities and the differences. Limit your response to two pages.

7. Select two software applications (e.g., office application, GUI OS interface) and analyze them using the quality attributes. How well do they do? Limit your response to two pages.

8. Select a web site or an application an analyze it with respect to the four graphic design values (contrast, repetition, alignment, proximity).

9. Using the description method for patterns, describe widgets.

10. Using the description method for patterns, describe the page layouts

11. Using the description method for patterns, describe the various navigational patterns (or task structures).

12. Evaluate a website from the perspective of accessibility.

13. Evaluate an application from the perspective of accessibility.

Phase 1 - Concept Exploration

* Ideas for the software/technology product to help users solve problems or accomplish goals are developed. * What the product needs to do (requirements analysis) and how the product will be used (task analysis) are explored. * Platform capabilities and constraints are considered. * The Usability focus is on getting the product to work.

Usability evaluations in Phase 1 can yield information about

* the need for the product * the needs for particular functions/features of the product.

Key Usability Tests in Phase 1: User Needs Assessments, Laboratory User Testing of paper mock-ups or prototypes

Phase 2 - Demonstration and Validation

* The concept is implemented. * Product requirements and constraints are refined and prioritized in order to identify or select those elements critical to the prototype. * The key market is defined. * The Usability focus is on having the right functions.

Usability evaluations in Phase 2 can yield evidence of the success or failure of the design concept.

Key Usability Tests in Phase 2: Heuristics and Walkthrough evaluations and Laboratory User Testing

Phase 3 - Detailed Design and Construction

* The required software elements are defined and integrated into the prototype. * Further context analysis is done to determine if any significant changes have occurred in the users' workplace since Phase 1.

Usability evaluations in Phase 3 can be used to determine whether or not the software is ready to go into production, or if further development and testing is needed.

Key Usability Tests in Phase 3: Heuristics and Walkthrough evaluations and Laboratory User Testing

Phase 4 - Production and Operation

* Plans for integrating and testing the product in the "real world" environments for which it is intended are developed. * Plans for "final" product delivery and installation are developed. * Product maintenance and support plans are developed. * Processes for disposing of the product when it is no longer viable are defined.

Usability evaluations in Phase 4:

* Can guide product revisions, marketing and delivery plans, and the support services required. * Lead to product updates that require another round of usability evaluation thus repeating the usability engineering life cycle.

Key Usability Tests in Phase 4: Field User Testing

Chapter 11

Efficiency and the Design of Efficient Software

The quality of efficiency includes time behavior, resource utilization, and compliance with standards.

Focus on the design of the least flexible portions of the task first. Place elsewhere

11.1 Time Behavior

The performance of a system is dependent on three main factors: the goals of system, the resource demands of the system traffic, and the system performance and configuration of underlying technologies[1]. Each system is unique and must be tailored to address traffic impact on four resources: bandwidth, CPU, memory, and disk input/output.

Motivation

The design and implementation of large-scale systems that make optimum use of computational resources is a challenging problem. Very often, in a straight forward parallel, distributed, or clustered solution, resources are under-utilized or used inefficiently.

The factors which determine a program's performance are complex, interrelated, and oftentimes, hidden from the programmer. Some of them are listed by category below.

- Application Related Factors:
  - Algorithms
  - Dataset Sizes
  - Memory Usage Patterns
  - Use of I/O

[1]Note: material adapted from `http://www.llnl.gov/computing/tutorials/performance_tools/`, `http://www-unix.mcs.anl.gov/dbpp/text/node107.html`, and `http://en.wikipedia.org/wiki/Performance_analysis`.

  - Communication Patterns
  - Task Granularity
  - Load Balancing
  - Amdahl's Law

- Hardware Related Factors:
  - Processor Architecture
  - Memory Hierarchy
  - I/O Configuration
  - Network

- Software Related Factors:
  - Operating system
  - Compiler
  - Preprocessor
  - Communication protocols
  - Libraries

Because of these challenges and complexities, performance analysis tools are essential to optimizing an application's performance. They can assist you in understanding what your program is "really doing" and suggest how program performance should be improved.

11.2 Performance Analysis

We distinguish three basic steps in the performance analysis process: data collection, data transformation, and data visualization. Data collection is the process by which data about program performance are obtained from an executing program. Data are normally collected in a file, either during or after execution, although in some situations it may be presented to the user in real time. Three basic data collection techniques can be distinguished:

- Profiles record the amount of time spent in different parts of a program. This information, though minimal, is often invaluable for highlighting performance problems. Profiles typically are gathered automatically.
- Counters record either frequencies of events or cumulative times. The insertion of counters may require some programmer intervention.
- Event traces record each occurrence of various specified events, thus typically producing a large amount of data. Traces can be produced either automatically or with programmer intervention.

The raw data produced by profiles, counters, or traces are rarely in the form required to answer performance questions. Hence, data transformations are applied, often with the goal of reducing total data volume. Transformations can be used to determine mean values or other higher-order statistics or to extract profile and counter data from traces. For example, a profile recording the time spent in each subroutine on each processor might be transformed to determine the mean time spent in each subroutine on each processor, and the standard deviation from this mean. Similarly, a trace can be processed to produce a histogram giving the distribution of message sizes. Each of the various performance tools described in subsequent sections incorporates some set of built-in transformations; more specialized transformation can also be coded by the programmer.

Parallel performance data are inherently multidimensional, consisting of execution times, communication costs, and so on, for multiple program components, on different processors, and for different problem sizes. Although data reduction techniques can be used in some situations to compress performance data to scalar values, it is often necessary to be able to explore the raw multidimensional data. As is well known in computational science and engineering, this process can benefit enormously from the use of data visualization techniques. Both conventional and more specialized display techniques can be applied to performance data.

A profiler is a performance analysis tool that measures the behavior of a program as it runs, particularly the frequency and duration of function calls. The output is a stream of recorded events (a trace) or a statistical summary of the events observed (a profile). Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, operating system hooks, and performance counters.

As the summation in a profile often is done related to the source code positions where the events happen, the size of measurement data is linear to the code size of the program. In contrast, the size of a trace is linear to the program's runtime, making it somewhat impractical. For sequential programs, a profile is usually enough, but performance problems in parallel programs (waiting for messages or synchronisation issues) often depend on the time relationship of events, thus requiring the full trace to get an understanding of the problem.

A variety of profiling and execution analysis tools exist for both serial and parallel programs. They range widely in usefulness and complexity and include: tools

- Simple command line timing utilities
- Fortran and C timing routines
- Profilers
- Execution trace generators
- Graphical execution analyzers - with/without trace generation
- Both real-time and post-execution tools

Most of the more sophisticated and useful tools have a learning curve associated with them and deserve a full day tutorial.

11.3 Performance Considerations and Strategies

- The most important goal of performance tuning is to reduce a program's wall clock execution time. Reducing resource usage in other areas, such as memory or disk requirements, may also be a tuning goal.
- Performance tuning is an iterative process used to optimize the efficiency of a program. It usually involves finding your program's hot spots and eliminating the bottlenecks in them.
  - *Hot Spot:* An area of code within the program that uses a disproportionately high amount of processor time.
  - *Bottleneck:* An area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays.
- Performance tuning usually involves profiling - using software tools to measure a program's run-time characteristics and resource utilization.
- Use profiling tools and techniques to learn which areas of your code offer the greatest potential performance increase BEFORE you start the tuning process. Then, target the most time consuming and frequently executed portions of a program for optimization.
- Consider optimizing your underlying algorithm: an extremely fine-tuned O(N * N) sorting algorithm may perform significantly worse than a untuned O(N log N) algorithm.
- For data dependent computations, benchmark based on a variety of realistic (both size and values) input data sets. Maintain consistent input data during the fine-tuning process.
- Take advantage of compiler and preprocessor optimizations when possible.
- Finally, know when to stop - there are diminishing returns in successive optimizations. Consider a program with the following breakdown of execution time percentages for the associated parts of the program:

11.4 Profiling Using Instrumentation

- Manual. Done by the programmer, e.g. by adding instructions to explicitly calculate runtimes.
- Compiler assisted. Example: "gcc -pg ..." for gprof.

- Binary translation; the tool adds instrumentation to a compiled binary. Example: ATOM
- Runtime instrumentation: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool. Examples: PIN, Valgrind
- Runtime injection: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst

11.5 Suggestions

The following may be used to improve performance/efficiency:

- Replace with more efficient algorithm
- Prioritize service (unfair/starvation)
- Clone and load balance
- Decompose (if consists of a complex set of heterogeneous services) into functionally different services
- Pipeline (if milti-step service)

Chapter 12

Maintenance and the Design of Maintainable Software

This chapter is largely incomplete

Maintainability and portability are the maintainance qualities in the ISO Standard 9126 (see Figure 12.1). Refactoring (covered in Chapter 7) is the process of replacing one section of code by another. Refactoring may be used during the design and implementation phases as well as the maintenance phase.

Readability: related to lexical and semantic complexity.

Software must evolve or die. The majority of maintenance costs (80%) is for non-corrective action i.e., evolution of the software. As software is modified and evolves, its complexity increases. Personnel changes occur in the maintenance or development team. As a consequence, a great deal of effort (40-60% of the maintenance effort) is devoted to understanding the software to be modified. Changes in the software provides an opportunity to restructure (refactor) the software to improve its efficiency, its value, its design.

Maintainability includes:	**Portability includes:**
Analyzability	Adaptability
Changeability	Installability
Stability	Co-existence
Compliance	Replaceability
	Compliance

Figure 12.1: **ISO Standard 9126 Software Maintenance and Portability Quality Characteristics**

Operational Qualities	Maintenance Qualities
• Functionality: Plan to throw one away, Keep secrets, Use a good idea again, Divide and conquer • Reliability: Cache answers, Use hints, Use brute force, Compute in background, Batch processing • Usability: • Efficiency: Make actions atomic, Use hints	• Maintainability: • Portability:

Figure 12.2: **Software Quality Values and Lampson's Design for Implementation**

12.1 Maintainability

> *Software Maintenance* is the totality of activities required to provide cost-effective support to a software system. It involves the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

There are several categories of maintenance. Each is in response to some need.

- *Corrective maintenance* is the reactive modification to correct discovered problems. The problems may be caused by either design flaws or incorrect requirements.
- *Adaptive maintenance* is modification in response to changed requirements. It keeps the software usable in a changed or changing environment. It includes modifications to interface with other systems and converting it to use other hardware. It includes routine maintenance such as is required in a store front website.
- *Perfective maintenance* is to improve the design. It covers the full range of quality characteristics but often deals with performance or maintainability issues.
- *Preventive maintenance* is modification to detect and correct latent faults.

The maintainability of software is directly related to the degree to which it is comprehendable by the maintainers. The following enhance the maintainability of software.

- Documentation
- Componentization
- Abstraction
- Assignment of functionality to design elements
- Functional requirements
- Partitioning: decomposition and composition
- communication, synchronization, data access
- Selection of design elements
- Hierarchical structures: assembly of elements into larger structures
- Performance requirements: scalability and availability
- Style and aesthetics
- Centralized, replicated, distributed

The following increase the difficulty of software maintainance and portability. (from [31] p. 24):

- Creating an object by specifying a class explicitly.
- Dependence on specific operations.
- Dependence on hardware and software platforms.
- Dependence on object representations or implementations. Use information hiding.
- Algorithmic dependencies. Isolate the algorithm.
- Tight coupling. Use loose coupling and layering.
- Extending functionality by subclassing.
- Inability to alter classes conveniently.

12.2 Portability

- Standards
- Programming language
- Libraries
- Localize platform dependencies

12.3 Debugging

David J. Agans[7] identifies nine rules for debugging software and hardware. His rules are the following:

1. Understand the system: Read the manual, read everything in depth, know the fundamentals, know the road map, understand your tools, and look up the details.

2. Make it fail: Do it again, start at the beginning, stimulate the failure, don't simulate the failure, find the uncontrolled condition that makes it intermittent, record everything and find the signature of intermittent bugs, don't trust statistics too much, know that "that" can happen, and never throw away a debugging tool.

3. Quit thinking and look (get data first, don't just do complicated repairs based on guessing): See the failure, see the details, build instrumentation in, add instrumentation on, don't be afraid to dive in, watch out for Heisenberg, and guess only to focus the search.

4. Divide and conquer: Narrow the search with successive approximation, get the range, determine which side of the bug you're on, use easy-to-spot test patterns, start with the bad, fix the bugs you know about, and fix the noise first.

5. Change one thing at a time: Isolate the key factor, grab the brass bar with both hands (understand what's wrong before fixing), change one test at a time, compare it with a good one, and determine what you changed since the last time it worked.

6. Keep an audit trail: Write down what you did in what order and what happened as a result, understand that any detail could be the important one, correlate events, understand that audit trails for design are also good for testing, and write it down!

7. Check the plug: Question your assumptions, start at the beginning, and test the tool.

8. Get a fresh view: Ask for fresh insights, tap expertise, listen to the voice of experience, know that help is all around you, don't be proud, report symptoms (not theories), and realize that you don't have to be sure.

9. If you didn't fix it, it ain't fixed: Check that it's really fixed, check that it's really your fix that fixed it, know that it never just goes away by itself, fix the cause, and fix the process.

12.4 Software Evolution

Software evolves though gradual change consisting of numerous, successive, slight modifications, through compositional change consisting of new combinations of fully functional components[93], and through refactoring which alters structure without changing external behavior.

12.5 Exercises

1. What relationship should exist between the documents which describe the requirements, the specifications, the design, and the implementation?
2. With reference to the previous question and the assumption that the maintenance of the relationship is important for maintenance, design a system to facilitate the managment of the relationship.
3. Suggest how information architecture (Chapter 14) may be used to facilitate maintainability.

Chapter 13

Coding and the Structure of Code

I love writing software, and I enjoy reading other people's source – how they've expressed instructions, the subtle differences when two good programmers use the same language for the same task. Then there's the pleasure of working through a new computer language: how its structure, its form, changes the way a problem is approached, a solution is expressed.
– *Dylan Harris*

Describe programmers: They have a hard time critqueing their own work. ... get from Maintenance and Construction.

13.1 Introduction

Requirements, specification, design, and implementation are coevolving phases of software construction. The code is validated by demonstrating that the program satisfies its requirements.

The goal of software construction is to produce code that

- fully and correctly processes data for its entire problem space,
- anticipates and handles all plausible classes of errors,
- runs efficiently, and
- is structured to be resilient and easy-to-change over time.

There are four principles which affect the way software is constructed.

1. Complexity: Keep it simple.
2. Diversity: Make it easy to change.
3. Validation: Make it easy to test.

4. Standards: Make it easy to read.

Reduction of complexity is facilitated by the general principle of parsimony, use the simplest solution. When complexity is unavoidable, it is best isolated in small units or modules that are small enough for a person to understand in their entirety. Anticipation of diversity (change) is facilitated by generalizing and by localizing areas of anticipated change. Structuring for validation is facilitated by modularizing the software so that each module performs one conceptual task. Use of external standards facilitates understanding by a larger group of individuals.

Small programs

Small programs are written using a simple iterative development model:

```
Repeat
    edit / compile / test(execute)
Until there is a complete working program
```

It may take several iterations of the model to produce a working program. As programs get more complicated, testing and debugging alone may not be enough to produce reliable code. Instead, we have to write programs in a manner that will help insure that errors are caught or avoided.

Incremental Program Construction

As programs become more complex, changes have a tendency to introduce unexpected effects. Incremental program construction tries to isolate the effect of changes and facilitates testing. Add new features in preference to adding new functions, and add new functions rather than writing new programs. The program implementation model becomes:

```
define classes / compile / fix;
add load and dump functions/ compile / test;
add first processing function / compile / test / fix;
add features / compile / test / fix;
add second processing function / compile / test / fix;
keep adding features / and compiling / and testing / and fixing.
```

Compositional Program Construction

Over time it is possible to develop a collection of reusable functions, modules, and objects. It is then possible to speed up the development of code by selecting a previous developed item from a library and inserting it into the development base.

Refactoring

Refactoring is a technique to restructure code in a disciplined way. It may be used to correct errors in the code, to simplify the code, to add additional functionality, introduce or reduce the level of abstraction, or to improve the aesthetics of the code.

> Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. – Martin Fowler

See `http://www.refactoring.com` for more information about refactoring techniques.

Development Environment

Tools

The development tools include specialized editors, compilers, version control systems, debuggers, code generators, editors, tools for path and coverage analysis, test scaffolding and documentation tools.

Versioning

In version control systems there are often a base line version, a development version, and one or more release versions.

Testing

Typically, sample runs of the program demonstrating the behavior for expected data values and boundary values are required.

> **Working Program Rule:** Always leave a program in a working state.

Modules

Provides structure which facilitates readability and permits reuse. Modules should be self-contained, independent, ..., global variables, ...

> **Write Once, Reuse Everywhere Rule:** Construct programs from self-contained reusable modules.
>
> **Write less code:** Modules improve productivity and correctness by permitting reuse of code.

13.2 Methods

Top-Down Software Construction

If the design is well understood, construction may begin with the top level module (the module that that is not called by any other module and only invokes other modules) and works toward the modules that do not invoke other modules. The implementation may proceed depth-first or breadth-first. Rule 1 is satisfied by using stubs (see below) for those modules that are called but not yet implemented. Testing may be performed by the customer.

Stubs

Stub programming is the implementation analogue of top-down design and stepwise refinement. It supports incremental program development by allowing for error and improvement.

> **Stub programming** is the use of skeleton versions of modules to allow construction of a program that can be compiled and run. It doesn't implement details of the algorithm or fulfill all the job requirements. However, it does contain rough versions of modules and their parameter lists.

Extensive use of modules and parameters are the difference between stub programs and prototypes. Quick and dirty prototypes should not be improved–they should be rewritten. A stub program helps demonstrate that a program's structure is plausible. Its procedures and functions are unsophisticated versions of their final forms, but they allow limited use of the entire program. In particular, it may work for a limited data set. Often the high-level procedures are ready to call lower-level code, even if the more detailed subprograms haven't even been written. Such sections of code are commented out. The comment brackets can be moved, call by call, as the underlying procedures are actually written.

Bottom-Up Software Construction

Whether a final design is available or not, construction begins with the bottom level modules (modules that do not call any other modules) and works toward the top-level module. Programmer designed test harnesses (see below) are used to test individual modules. The top-level module constitutes the final test harness.

Test Harnesses

> A **test harness** is a program shell that is used to test procedures in isolation, before they are integrated into a more complex final program.

A program is a delivery system for procedure calls. A test harness is precisely that–a delivery system for procedure tests that contains:

- the type definitions;
- procedures that initialize and/or inspect data structures; and
- the new procedure that is undergoing testing or modification.

The new procedure can be tested without having to deal with a main program that is more complex and finicky than the harness is. Once it works, the new procedure can be transferred to the main program.

Object-Oriented Programming

Given an object-oriented design,

- inheritance, reference, and class attributes are implemented in programming language classes,
- object interactions are encoded as methods belonging to a selected class, and
- the permitted sequences of operations are recognized by state machines.

From [18]

13.3 Testing

Verification is the process that determines whether the product is built right (implementation satisfies the specification). Validation determines whether the right product was built (implementation meets the needs of its users).

Verification

> A **trace** of a program consists of a listing of the values of each variable at each point in the execution of the program.

It is often difficult to determine what causes a program to fail. While program tracing is useful, it is difficult to perform on any but the smallest programs. Stub programs let larger systems be debugged and tested as they are being built, a small portion at a time. Major program connections are tested first, which means that major bugs and shortcomings are detected early in the game. Furthermore, testing and debugging are distributed throughout the entire implementation. Even if a program isn't completely finished by the due date it's a preliminary working version and not just a useless mess of code. Often procedures are built into programs to assist in testing a program and left behind in case they're needed again. It is also common practice to build certain testing tools that are thrown away. Testing is not a substitute for careful, thoughtful design and implementation.

Program testing can be used to show the presence of bugs, but never their absence!
– Edsger W. Dijkstra

Walkthrough

Working on a program tends to create a mind set in the programmer that renders obvious mistakes invisible. Merely explaining a program aloud can give a totally new view of it.

A **walkthrough** is an explanation and defense of the program's algorithm and implementation to an audience.

Program Animation/Instrumentation

Program animation/instrumentation is a way to inspect a program while it is running. It differs from tracing in that the values of selected variables are printed at specific points in the program.

A **program probe** is an output statement added to a program to indicate the progress of execution.

When the program is executed, the output statements are probes into the program which serve to animate the program. The program can then be compared to the output to isolate the program errors. A program probe often takes the following form:

```
Debug = true;
...
if Debug //label and print the value of the variable
...
```

Often the particular value of a variable is not as important as whether or not it meets some particular constraint (e.g. whether an expression is positive). Statements which express such constraints are called **assertions**.

An **assertion** is a boolean expression which is to be true at a particular point in a program.

A probe which prints an error message only when variable fails to meet the constraint can be written as follows:

```
void assert(bool BooleanExpression,  string ErrorMessage)
{ if not BooleanExpression print ErrorMessage ; }
```

Then a statement such as: `assert(X≥ 0,"X is negative")`, can be inserted into an appropriate point in the code.

Programs that are run as processes such as a server often are written to generate a record of their activity. Such a record is called a log file.

A **log file** is a file created during the operation of a program which ...

Log files are useful for ... error diagnosis ...

Integration Testing

Testing to check that modules combine together correctly. In addition, there should be a final product test and acceptance testing by the client.

Regression Testing

Testing that is performed to insure that modifications to a program have not modified previously correct behavior. This requires a collection of test data be maintained for the purpose of regression testing.

Validation: Acceptance Testing

Testing that is performed by the customer to determine whether the software solves the problem (whether the right product was built).

13.4 Coding Style and Documentation

Goal: readability

- Readability
  - Layout
  - Visual cues
- Maintainability
  - Documentation

The visual appearance of your programs is as important as the solution they represent. Use plenty of white space (blank lines, indentation) to draw attention to the structure of your solution and to improve readability. In addition follow a consistent style in program layout. The suggestions that follow are just that, suggestions. You will develop your own style and employers usually set their own standards. Illustrations employ C++.

Comments and program structure

A space follows the '{' beginning a comment and before the '}' ending a comment. If a comment extends over more than one line, subsequent lines should be indented the same level as the '{'.

A comment that applies to a group of declarations or statements should appear before the group and be preceded by a blank line. Major sections of code should be introduced by comments in boxes.

```
/*****************************************
*  Section name and other information
*****************************************/
```

Indentation

Indentation should be used to show the structure of the program, declarations and statements. The amount of indentation should be the same – two to four spaces seem to be good values. Statements or declarations needing more than one line should have subsequent lines indented more than one level.

```
#include <iostream.h>;
int some variable;
void function()
{
    body
}

void main()
{   if true
       initialize
    else
       process
}
```

When a declaration or statement can be placed on a single line without appearing to be cramped, consider doing so;

```
if (x < Limit) y = K*p;
```

is better than

```
if (x < Limit)
   y = K*p;
```

Blank Lines

Procedure declarations should be separated by at least one blank line. Long lists of procedure parameter declarations should be written putting each parameter on its own line.

Reserved Words and Identifiers

Variables should be commented to identify their purpose. Reserved words should be distinguishable from user defined identifiers either by writing them in all caps or in all lower case or by judicious use of indentation and white space.

User defined identifiers should be chosen to aid the reader in understanding the program. Long names should be avoided as they tend to reduce the readability of a program. Single letter identifiers should be avoided except for loop indices in which case *i, j,* and *k* are typical indices.

1. Identifiers that name modules, procedures, exceptions, types and constants should begin with an uppercase letter. (e.g. `GetData, Factorial(n)`)
2. Identifiers that name variables, parameters, record and object components should begin with a lowercase letter. (e.g. `radius, score[i], student.name`)
3. The initial letter of each embedded word except the first should be capitalized or separated from the previous word by an underscore. Acronyms should be in a single size. (e.g. `thisLongName, this_long_name, CPUType, cpuType`)

Constants

Define and use symbolic names for constants. Only rarely should numeric literals other than 0 and 1 appear in the program text (e.g. `Pi = 3.14159`).

Types and Data Structures

User defined types should end in a capital `T` to permit the use of the name as a variable (e. g. `type_identifierT = type_definition, identifier = identifierT;`).

Data structures sould be commented to identify their purpose, organization and access functions.

Punctuation

A single space usually appears before and after all binary operators including `:=` and `=` used in assignments and declarations (e.g. `x + y, x*y, first = last + 1, assignable := expression`).

A single space or a newline appears after colon, comma, and semicolon, but none before. Unless required by adjacent tokens, no spaces appear before or after left and right parentheses, square brackets, or curly braces, or the uparrow (ˆ), period (.), or double period (..). (e.g. `Factorial(n), score[i], a comment, list^.tail, name.last, array[0..Max]`)

Statements in sequence are separated by semicolons. For consistency, also place a semicolon after the last statement in a sequence. (e.g. `First; Second; ... Last;`)

Programs

Programs must have a header identifying the program, programmer, revision history and other pertinent information.

```
/**************************************************
Program file name:
Language:
Operating System:
Programmer:
Date:
Revision History:
Title:  program title or name
Purpose:  short description or purpose of the program
Input:  what the user must supply
Output:  what the program prints/produces
Copyright information:
Special requirements:
*************************************************/

#include <iostream.h>;

void main()
{ the main program }
```

The following must be included when the code is submitted as part of a class assignment.

```
Class:
Section:
Assignment:
```

Procedures and functions

Procedures and functions should be commented to identify their pre- and post-conditions.

```
/*
Title:  function title or name
Purpose:  short description or purpose of the program
Input/Precondition:  what the user must supply
Output/Postcondition:  what the function computes
Data Structures:
Algorithms:
*/

type function identifier (parameters)
{
    body of the function
    return expression
}
```

The pre- and post-conditions may be implemented using assertions.

Repetitive structures

Repetitive structures (including recursive functions and procedures) should be commented to identify their purpose, termination conditions and progress functions.

```
/*
Goal/Invariant:  overall purpose of the loop
Bound/Termination condition:  reason loop will end
Plan/Metric:  the action the loop will take including approaching
the bound
*/
```

The invariant may be implemented using assertions and should be checked prior to entering the loop and at the end of each iteration.

Complex Algorithms

Complex algorithms should be commented to assist the reader in understanding the code and where appropriate a citation of the source of the algorithm.

13.5 Quality and Software Construction Techniques

Quality in software is affected by the organization of the software and the methods and styles used in its construction. The SWEBOK[86] identifies two dimensions of quality software – software organization and styles of construction. Patterns, libraries and APIs, programming language features, standards and practices are then identified with respect to the style and principle that each facilitates. Table 13.1 displays the result.

13.6 References

1. Some portions are adapted from *Oh! Pascal* third edition.

2. Coding style is adapted from: **Modula-3** by Samual P. Harbison.

13.7 Exercises

1. Select a programming language and construct a programming style guide for it.

2. Critique a style guide.

Quality:	Functionality	Maintainability	Functionality, Maintainability	Portability
Three Styles of Construction	Four Principles of Software Organization			
	Reduction in complexity Removal, Automation, Localization	**Anticipation of diversity** Generalization, Experimentation, Localization	**Structuring for validation** Modularity	**Use of external standards** Communication
Linquistic Methods	• Design patterns • Software templates • Functions, procedures, code blocks • Objects and data structures • Encapsulation and abstract data types • Component libraries and frameworks • Higher-level and domain specific languages • Files and libraries • Formal inspections	• Information hiding • Comments • Complete and sufficient method sets • OO methods • Glue languages • Table driven software • Configuration files, internationalization • Naming and coding styles • Reuse and repositories • Self-describing software (plug and play)	• Modular design • Structured programming • Style guides • Stepwise refinement	• Standardized languages • Standardized data description languages • Standardized alphabet representations • Standardized documentation • Inter-process communication standards • Component-based software • Foundation classes
Formal Methods	• Functions and procedures • Functional programming • Logic programming • Concurrent and real-time programming • Spreadsheets • Program generators • Mathematical libraries	• Parameterization • Generics • Objects • Error handling • Extensible frameworks	• Assertion-based • State machine logic • Redundant systems, self-diagnosis, and fail-safe methods • Hot spot analysis and performance tuning • Numerical analysis	• POSIX standards • Data communication standards • Hardware interface standards • Standardized mathematical representation languages • Mathematical libraries
Visual Methods	• OOP • Visual creation of user interfaces • Visual programming • Style aspects • IDEs supporting source browsing	• Object classes • Visual configuration specification • Separation of GUI design and functionality implementation	• Complete and sufficient design of OO class methods • Dynamic validation of visual requests	• OO language standards • Standardized widgets • Visual markup languages

Figure 13.1: Software Organization and Styles of Construction (SWEBOK)

3. Submit a program you have written which illustrates all of the suggestions in this chapter. In particular, it illustrates the coding style documentation and the use of assertions for pre- and post-conditions and invariants.

4. Write a program which merges two ordered files (or arrays of ordered elements) and use assertions to verify the data structure invariants.

5. Write a program which applies transactions against a master file creating an updated master file. Assume that the master file and the transaction file are sorted into assending order. Use assertions to verify the data structure invariants.

6. Propose a categorization scheme for refactorings (see `http://www.refactoring.com/catalog/index.html`.

7. Critique Table 13.1.

8. Critique this chapter using Green's Cognitive Dimensions in Appendix 3.1.

Chapter 14

Information Architecture

This chapter is largely based on Chapter 1 and Chapters 4-8 in [66]

14.1 Defining Information Architecture

Information Architecture is

1. The combination of organization, labeling, and navigation schemes within an information system.

2. The structural design of an information space to facilitate task completion and intuitive access to content.

3. The art and science of structuring and classifying web sites and intranets to help people find and manage information.

4. An emerging discipline and community of practice focused on bringing the principles of design and architecture to the digital landscape.

Information architecture is done by a team that includes graphic designers, interaction designers (human-computer interaction), usability engineers, experience designers, software developers, content managers, and knowledge managers. The number and titles of the various team members give an indication of the type and number of evaluative criteria that must be used to evaluate a website. In particular, the needs and capabilities of the user predominate.

14.2 The Anatomy of an Information Architecture

The components used in an information architecture are subject to different categorizations. The categorization used in this document is organizational systems, labeling systems, navigation systems, and searching systems. An alternative categorization is browsing aids, search aids, content and tasks, and "invisible" components.

14.3 Organization Systems

The motivation for an organizational system is information overload. Organizational systems facilitate the recognition, identification, and access to information. *Organizational systems* are composed of *organizational schemes* and *organizational structures*.

An *organizational scheme* defines the shared characteristics of content items and determines the logical grouping of those items. The challenges to any organizational scheme include

- Ambiguity
- Heterogeneity
- Differences in perspective
- Internal politics

There are two types of organizational schemes, *exact* and *ambiguous*. Exact organizational schemes include alphabetical, chronological, and geographical. Ambiguous organizational schemes include topic, task, audience, metaphor, and hybrid schemes. Exact schemes are best for known-item searching when users know precisely what they are looking for. Ambiguous schemes are best for browsing and associative learning when users have a vaguely defined information need.

An *organizational structure* defines the types of relationships between content items and groups. Organization structures include the hierarchy, the database-oriented model, and hypertext (or network model). Large sites will use all three types of structures. The hierarchical model is often used for the top-level of the site. Collections of structured, homogeneous information are candidates for the database model. Less structured, creative relationships are handled through hypertext.

14.4 Labeling Systems

Labels are used to clarify the organization of a site and for navigation systems. Labels appear in two formats: textual and iconic. Labels are commonly used for contextual links, headings, navigation system choices, and index terms.

14.5 Navigation and Search Systems

There are two types of navigation systems. The first type are *embedded navigation systems*. They are embedded in the content pages. They include global, local, and contextual navigation systems. The second are *supplemental navigation systems* such as site-maps, indexes, and guides that exist outside of the content-bearing pages. The common architectural features used for navigation include links, navigation bars, pull-down, pop-ups, and cascading menus. *Search systems* are an alternate mechanism for finding information.

It is easy for users to get lost in a multi-page website. Contextual clues help to keep users from getting lost. Users should always know which site they are in and their relative location within the site.

A simple measure of navigational complexity is the # links followed. A web site may be designed with the three click rule in mind.

> **Three Click Rule:** users should be able to access the information they need with just three clicks.

Embedded Navigation Systems

Global navigation systems often provide links to the home page, search functions, and provide contextual clues the the user's current location within the site. A global navigation system should be present on every page throughout a site. It is often implemented as a navigation bar at the top (and bottom) of each page.

Local navigation systems enable the user to access a *subsite* of the web site. The local navigation system may be tightly or loosely integrated with the global navigation system. It is often implemented as a navigation bar at the left of the page. *Contextual navigation* provides access to items specific to a particular page, document or object. These links are often determined by the page content and are not architectural in nature. It is often implemented as links embedded in the text or (for more visibility) they may be collected into a specific area of the page.

Embedded navigation systems are often implemented in the form illustrated in Figure 14.1.

Supplemental Navigation Systems

A *sitemap* is based on the idea of a table of contents. It presents an overview of the site in terms of the top few levels of an information hierarchy. It is best suited to sites with a strong hierarchical organization. The site map should follow the following rules of thumb:

1. Reinforce the information hierarchy so the user may become familiar with how the content is organized.
2. Facilitate fast, direct access to content for experienced users.
3. Avoid overwhelming the user with too much information.

A *site index* is based on the idea of a back-of-book index. It presents an alphabetical list of keywords and phrases. It bypasses the site architecture to provide know-item finding. *Guides* occur in several forms: guided tours, tutorials, and micro-portals. A guide should follow the following rules of thumb:

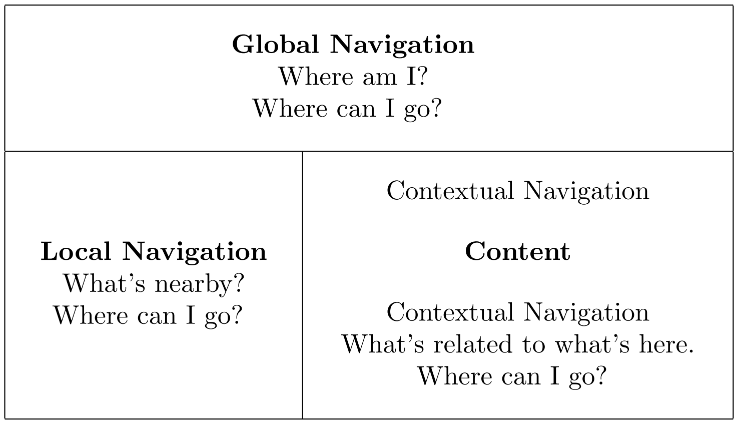


Figure 14.1: **Embedded Navigation**

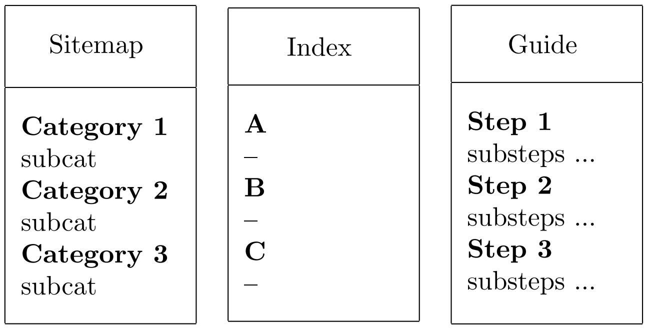


Figure 14.2: **Supplemental Navigation Forms**

1. It should be short.
2. At any point, the user should be able to exit the guide.
3. The navigation tool placement should be consistent.
4. It should answer the user's questions.

Supplemental navigation systems are often implemented using the form illustrated in Figure 14.2.

Search Systems

A search system is helpful in the following context:

- There is too much information to browse.
- The site is fragmented perhaps with many different authorities.

- Users expect it.
- The site contains highly dynamic content.

The following questions are helpful in determining whether a site needs a search system.

- Does your site have enough content to justify a search system?
- Will investing in a search system divert resources from improving the navigation system?
- Are there sufficient resources (time and know-how) to optimize the search system.
- Are there better alternatives such as a site index?
- Will your site's users bother with search?

14.6 Exercises

1. Define the circumstances that call for each type of organizational structure.
2. Pick a website for analysis and answer the following questions:
   (a) What kind of supplemental navigation systems are provided and how effective are they?
   (b) What kind of embedded navigation systems are provided and how effective are they?
   (c) How well does the site navigation features integrate with the common browser navigation features (link colors, back button, bookmark feature)?
3. Pick a website for analysis and perform the following Navigation Stress Test[46]:
   (a) Ignore the home page and jump directly into the middle of the site.
   (b) For each random page, can you figure our where you are in relation to the rest of the site? What major section are you in? What is the parent page?
   (c) Can you tell where the page will lead you next? Are the links descriptive enough to give you a clue what each is about? Are the links different enough to help you choose one over another, depending on what you want to do?
4. Pick a website and determine whether or not it needs a search system.
5. Pick a website and justify its search system.

6. Identify and formally describe one or more information architecture patterns.
7. Using an information architecture, design an approach to software architecture that minimizes the reading overhead for software maintainance.
8. Research and report on the available open source search engines.
   (a) For use as a general web search engine.
   (b) For use as a website search engine.
   (c) For use as a user directory search engine.

Part III

The Appendices

Appendix A

The Requirements Document

The requirements [1] are the features that the system must have and constraints that it must satisfy for the product to be accepted by the client (and stakeholders). The requirements of the system are obtained from stakeholders (including different people at different levels of an organization), feasibility studies, market analyses, business plans, analyses of competing products, domain knowledge, and from the environment in which the system must operate. The document is written for the system stakeholders written in terms of the customer's domain.

Introduction

The overall objectives for the system.

1. *Purpose of the system*
2. *Scope of the system*
3. *Objectives and success criteria of the project*
4. *Definitions, acronyms, and abbreviations*
5. *References*
6. *Overview*

Overall Description

A description of the target environment.

[1]This outline is adapted from IEEE Standard. 830-1998 IEEE Recommended Practice for Software Requirements Specifications

1. *Product perspective*
2. *Product functions*
3. *Stakeholders*
    (a) *Users and user characteristics*
    (b) *Customers*
    (c) *Market analysts*
    (d) *Regulators*
    (e) *System developers*
4. *Constraints*
    (a) *Cost*
    (b) *Time*
    (c) *Quality*
    (d) *Scope*
5. *Assumptions and dependencies*

Complete List of System Requirements

The results of the analysis of the data collected during requirements elicitation.

1. *External interface requirements*
    (a) *User interfaces*
    (b) *Hardware interfaces*
    (c) *Software interfaces*
    (d) *Communications interfaces*
2. *Operational Requirements*
    (a) *Functionality: suitability, accuracy, interoperability, security, compliance*
    (b) *Reliability: maturity, fault tolerance, recoverability compliance*
    (c) *Usability: understandability, learnability, operability, attractiveness, compliance*
    (d) *Efficiency: time behavior, resource utilization, compliance*
3. *Maintenance Requirements*
    (a) *Maintainability: analyzability, changeability, stability, compliance*

(b) *Portability: adaptability, installability, co-existence, replaceability, compliance*

4. *Physical Environment*
5. *Software system attributes*
6. *Other requirements including prioritization of the requirements.*

Appendix

The data collected during requirements elicitation.

- *Conceptual System Models*
  - *User Stories – written by the customers describing things that the system needs to do for them.*
  - *Scenario – an example of system use in terms of a series of interactions between the user and the system.*
  - *Use case – an abstraction that describes a class of scenarios.*
  - *User interface – navigational paths and screen mock-ups*
  - *Principle domain entities: an object model (Data dictionary, Class diagrams)*
  - *Dynamic models: data, information, and workflows*
- *The system's environment*
- *Feasibility studies*
- *Market analyses*
- *Business plans*
- *Analyses of competing products*
- *Domain information*

Appendix B

The Software Design Document

The design document outlined in the IEEE Std. 1016-1998 IEEE Recommended Practice for Software Design Descriptions is structured as follows:

- Introduction
  - Purpose
  - Scope
  - Definitions, acronyms, and abbreviations
- References
- Decomposition description
- Dependency description
- Interface description
- Detailed design

Alternatively, a design may be presented in two levels – a top level design called the architectural design which focuses on components and interfaces and a low level design which focuses on code.

The Architectural Design

In architectural design (top level design) the top-level structure and organization of the system is described and various components are identified (how the system is decomposed and organized into components and the interfaces between these components).

The design should cover the following issues:

- *Concurrency: processes, tasks, threads; efficiency atomicity, synchronization, and scheduling*
- *Control and handling of events: flow of data and flow of control, reactive and temporal events.*
- *Distribution: component communication and middle ware for heterogeneous systems*
- *Error and exception handling and fault tolerance: how to prevent and tolerate faults and exceptional conditions.*
- *Interactive systems: how the system is to interact with users*
- *Persistence: how long-lived data is to be handled.*

The architectural structures and viewpoints include:

- *logical view (satisfying the functional requirements)*
- *process view (concurrency issues)*
- *physical view (distribution issues)*
- *development view (implementation units)*

The design should be expressed using structural and/or behavioral descriptions.

- *Structural descriptions (static view)*
  - *Architecture description languages*
  - *Class and object diagrams*
  - *Component diagrams*
  - *CRC cards*
  - *Deployment diagrams*
  - *Entity-Relationship diagrams*
  - *Interface description languages*
  - *Jackson structure diagrams*
  - *Structure charts*
- *Behavior descriptions (dynamic view)*
  - *Activity diagrams (flow of control)*
  - *Collaboration diagrams (interactions among a group of objects)*
  - *Data flow diagrams*
  - *Decision tables and diagrams*

- *Flowcharts and structured flowcharts*
- *Formal specification languages*
- *Pseudo-code and program design languages*
- *Sequence diagrams*
- *State transition and statechart diagrams*

The rationale for the proposed software architecture facilitates the evolution of the software. Rationale should be provided for the following:

- *Subsystem decomposition*
- *Hardware/software mapping*
- *Persistent data management*
- *Access control and security*
- *Global software control*
- *Boundary conditions*

The Implementation Design

In the implementation design, each component is sufficiently described to allow for its coding.

Appendix C

Information Systems Analysis and Design

http://www.netnam.vn/unescocourse/sys_ana_des/system.htm

C.1 System Requirements

C.2 System Design

Appendix D

Software Quality Characteristics

This chapter is

Connell's design values

- Cooperation - integration with the environment which consists of hardware, OS, middleware, other applications, style of the intended users

- Appropriate form - internal form follows external functions

- Simplicity - work and solve the problem in the simplest manner possible.

Putting it together - a global quality of a beautiful overall design.

Architecture and critical non-functional requirements

Figure D.1 displays the quality characteristics identified in the ISO Standard 9126. The quality characteristics are not orthogonal to each other.

D.1 Functionality

Separate normal and worst case

Simplify: Do one thing well. Don't generalize. Get it right.

Corollaries: Don't hide power. Use procedure arguments. Leave it to the client.

Continuity: Keep basic interfaces stable. Keep a place to stand.

conceptual integrity, correctness, completeness, buildability

Operational Qualities	**Maintenance Qualities**
• Functionality: suitability, accuracy, interoperability, security, compliance	• Maintainability: analyzability, changeability, stability, compliance
• Reliability: maturity, fault tolerance, recoverability compliance	• Portability: adaptability, installability, co-existence, replaceability, compliance
• Usability: understandability, learnability, operability, attractiveness, compliance	
• Efficiency: time behavior, resource utilization, compliance	

Figure D.1: ISO Standard 9126 Software Quality Characteristics

Suitability

Accuracy

Interoperability

Security

System architecture should be layered with most critical assets protected in the innermost layers with a high level of security validation applied to these layers. System architecture should be designed so that safety-related operations are all located either in a single sub-system or in a small number of sub-systems to reduce costs and problems of safety validation and to make it possible to provide related protection systems.

Compliance

Comply with relavant standards and practices.

D.2 Reliability and Dependability

End-to-End: log updates, make actions atomic System architecture should be designed so that safety-related operations are all located either in a single sub-system or in a small number of sub-systems to reduce costs and problems of safety validation and to make it possible to provide related protection systems.

Reliability refers to the longevity of product performance. robustness

Maturity

Maintain development across multiple releases.

Fault Tolerance

Use robust methods to protect against permanent failure of a limited number of components. Use stabilizing methods to protect against transitory faults.

Recoverability

Use recovery oriented methods. System architecture should be designed with components that can be restarted independently of the other components. System architecture should be designed with an *undo function* to rewind time, untangle problems, and replay the system back to the current time.

Compliance

Comply with relavant standards and practices.

Dependability

Availability

Availability refers to the product's readiness for use on demand. System architecture should be designed with redundant components that can be replaced and updated without stopping the system.

Reliability

The longevity of product performance.

Maintainability

The ease of maintenance and upgrade

Maintenance Support

Maintenance support is a continuing support to achieve availability performance objectives.

Trustability

Trustability refers to the system's ability to provide users with information about service correctness.

Additional attributes

fault-tolerance, safety, security, and usability

D.3 Usability

Understandability

Understandability requires a good conceptual model with a clear locus of control.

Learnability

Learnability requires attention to the needs of the novice and uninitiated users. The uninitiated user is one that has no previous experience with the software or similar software. The novice user has either had some experience with similar software or has limited experence with the software.

Operability

Part of the design process for operability is to develop scenarios and use cases for novice, uninitiated, and expert users. Operability is enhanced through navigational efficiency i.e., users can locate the information they want.

Attractiveness

Attractiveness is achieved through layout, graphics, color, and dynamic elements.

Compliance

Usability is enhanced through following standards and accepted practices.

D.4 Efficiency

Shed load, End-to-end - error detection: Cheap test, Performance defects appear under heavy load, Safety first, scaling, throughput, latency, availability

Time Behavior

System architecture should be designed to localize critical operations within a small number of sub-systems with as little communication as possible between these sub-systems. Possibly using relatively large-grained rather than fine-grained components to reduce communication.

To make it fast: split resources, use static analysis, and dynamic translation.

Resource Utilization

Compliance

Comply with relavant standards and practices.

D.5 Maintainability

Maintainability refers to the ease of maintenance and upgrade. It also refers to continuing support to achieve availability performance objectives. System architecture should be designed using fine-grain, self-contained components. Data producers should be separated from consumers. Shared data structures should be avoided.

loose coupling, manageability, deployment, comprehensibility, extensibility, reusability, and platform independence, product evolution

Readability - clarity of design and documentation of the intention of each module.

1. Clarity built into the code
    - meaningful names
    - good use of white space
    - transparent control structures
    - straight-line normal execution paths
2. Comments to annotate the code - comments which educate about topics that cannot be gleaned from the code itself.
3. System minimality - software should be as simple as possible.
4. Component singularity - one component for each function
5. Functional locality - related items are placed in close proximity

Analyzability

Diagnostics: System architecture should be designed to pinpoint the source of failure. testability, traceability

Changeability

modifiability

Stability

Compliance

Comply with relavant standards and practices.

D.6 Portability

Architecture should isolate system dependencies. Loose coupling. integrability, reusability

Adaptability

Installability

Co-existance

Replaceability

Compliance

Comply with relavant standards and practices.

D.7 Quality Characteristics of Special Types of Systems

Human User Interface

The quality characteristics of the human-user interface are adaptivity, intelligent help, and display salience.

Get more from Usability chapter

Adaptivity

Adaptivity suggests that the system should be designed to the needs of different types of users. Grapical user interfaces (GUIs) which contain iconic and text are useful for a variety of human users while command line interfaces are suitable for batch processing. It should be intuitive for the novice and efficient in perception and motion for the expert. It should be rich in incrementally learnable functions, like the alphabetical shortcuts on the Mac interface

Intelligent Help

Intelligent help systems are those that are designed to help diagnose errors and suggest corrective action. In addition they may provide explanations and assistance to users. It should be robust under misuse. It should facilitate recovery from cognitive or manipulative mistakes. It should be helpful in diagnosing errors and suggesting corrective action.

Display Salience

Display salience implies that systems should be designed so that the important features are readily recognizable. Good layout and design principles as well as simplicity, elegance, clarity, order, and consistency help to insure display salience.

Websites and Information Systems

The quality characteristics of an information system and websites are effectiveness, affectiveness, and navigational efficiency. The quality characteristics of information systems incorporating a database system include: ease of query, high recall, high precision, and appropriate tradeoffs.

Effictiveness

Effectiveness implies that the presentation should be complete, sensibly organized, and accurate in conveying the intended message. This includes the technical specifications.

Affectiveness

Affectiveness implies that the interface is interesting, stimulating, and enjoyable (appealing to the user). This covers the look and feel of the site/page.

Navigational Efficiency

Navigational efficiency implies the ease with which users can locate the information they want. This includes the structural principles followed. A measure of navigational efficiency is the number of clicks required to locate an item of information.

Intelligent and Knowledge Based Systems

Anytime property (best answer within a given amount of time), explanation capability

D.8 Other Considerations

- Stylishness of code
- Code and object reusability
- Traceability: from requirements to code/test documentation and vice versa.
- Modularity of code and independence of modules

Appendix E

Templates

There are a variety of formats for the description of patterns. The essential components are:

Name	A well chosen name facilitates communication of the design.
Context	A situation giving rise to a problem.
Problem	A motivating problem that is recurs in that context.
Solution	A proven resolution of the problem.

The following sections present a variety of templates. The collection illustrates how the idea may be adapted to specific needs.

E.1 Architectural Pattern Template

Most important items are in bold.

Name	The name, a familiar descriptive name or phrase, usually indicative of the solution rather than the problem or context.
Description	Short summary of the pattern.
Also known as	Other names for the pattern, if any are known.
Example	A real-world example demonstrating the existence of a problem and the need for the pattern.
Context	Situations in which the pattern may apply. Often includes background, discussions of why this pattern exists, and evidence for generality.
Problem	A description of relevant forces and constraints, and how they interact. Sometimes design and construction constraints.
Solution	Static relationships and dynamic rules describing how to construct artifacts in accord with the pattern, often listing several variants and/or ways to adjust to circumstances, references and relation to other patterns.
Structure	A detailed specification of the structural aspects of the pattern.

Dynamics	Typical scenarios describing the run-time behavior of the pattern.
Implementation	Guidelines for implementing the pattern.
Example resolved	
Variants	Description of variants or specializations of a pattern.
Known Uses	Examples of the use of the pattern, taken from existingsystems.
Consequences	Benefits and any potential liabilities.
Depends on	
Is part of	some larger pattern.
See also	References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.
Credits	

Contributed by from [13] p. 19-21.

E.2 Design Pattern Template

The following short form for a pattern template is from [31] p. 3.

Name	The pattern name.
Problem	The problem describes when to apply the pattern.
Solution	The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations.
Consequences	The consequences are the results and trade-offs of applying the pattern.

The following long form for a pattern template is from [31] p. 6,7.

Name	
Classification	
Intent	What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it solve?
Also Known As	
Motivation	A scenario illustrating the design problem and how the class and object structures in the pattern solve the problem.
Applicability	The situations in which the design can be applied.
Structure	Graphical representation of the pattern using the OMT.
Participants	The classes and objects and their responsibilities in the design pattern.
Collaborations	How the participants collaborate to carry out their responsibilities.
Consequences	The trade-offs and results of using the pattern.
Implementation	Pitfalls, hints, techniques, or language specific issues useful in implementing the pattern.
Sample Code	

Known Uses
Related
Patterns

E.3 Enterprise Application Template

This template is suggested by Fowler[29].

- Name
- Intent
- Sketch
- Motivating problem
- How it works
- When to use it
- Further reading
- Examples

E.4 Algorithms and Data Structure Template

- Name
- Problem
- Solution
- Example
- Variants
- Complexity

E.5 Idiom Template

Idioms are patterns that are specific to a programming language (or family of programming languages).

- Name
- Problem
- Solution
- Example

E.6 Refactoring template

- Name
- Problem
- Solution
- Motivation
- Mechanics
- Example
- Comments
- Contributors

E.7 Antipatterns template

- Name
- Synopsis
- Refactored solution

Appendix F

Elements of Programming Languages

F.1 Data Types and Expressions

- Data Types
  - Simple
  - Complex
- Expressions
  - literals
  - variables
  - arithmetic expressions
  - relational expressions
  - logical expressions
  - functions

F.2 Commands/Control Structures

The basic operations and control structures include:

- Assignment and input
- Selection
- Iteration (definite and indefinite)
- Input & output
- Linear sequence
- Parallel Composition

To do: add axiomatic semantic descriptions.

Assignment and Input

Name *Assignment Statement*

Context Programming using an imperative programming language.

Problems A need to associate a value with a name.

Solution The assignment operation has the form: *variable assignOp expression*
Depending on the language, the assignment operator may be confused with mathematical equality. The assignment is not communitive i.e. x = e is not the same as e = x. In strongly typed languages the variable may need to be declared. Variables used in the expression must be defined (have values). In strongly typed languages, the type of the expression must be compatible with the type of the variable.

Semantics $\{Q[e/y]\}\ y := e\ \{y = e \wedge Q\}$

Variants *variable* := *expression*. (Ada/Pascal)
variable = *expression*. (C/C++/Fortran/Java)
variable ← *expression* (APL)
input commands

Known Uses Some typical uses include:

- initialize a variable (count := 0)
- increment/decrement a counter (count := count + 1)
- accumulate values (sum := sum + item)
- capture the result of a computation (y := 3*x + 4)
- swap two values (t := x; x := y; y := t)

The order in which assignments are performed is important for example, if the first and second assignments in the swap sequence were interchanged, x and y would end up assigned to the same value. The input operation and the output operation share some of the same constraints.

If-Then-Else

Name If - Then - Else

Description The pattern permits the selection ...

Also known as Choice, Selection

Example In the state of Washingtion, the If-then-else may be used to choose among different sales tax rates based on place of residence.

Context	If-then-else is used when there are values which require separate processing. For example, the computation of overtime pay, the roots of the quadratic formula and responding to user input of menu choices.
Problem	The condition must be a boolean expression. Programming languages may have additional syntactic requirements especially when the then or else clauses contain more than a single command.
Solution	if *condition* then *action1* else *action2*
Semantics	Selection Axiom:

$$\frac{\{P \wedge C\}\ S_0\ \{Q\},\ \{P \wedge \neg C\}\ S_1\ \{Q\}}{\{P\} \text{ if } C \text{ then } S_0 \text{ else } S_1\ \{Q\}}$$

Dynamics	
Implementation	
Example resolved	if *OregonResident* then *salesTaxRate := 0* else *salesTaxRate := 0.085*
Variants	No else clause: if condition then action; Elseif clauses: if condition then action elseif condition then action ... else action; Guarded Choice: if { guard_1 $\rightarrow$ action_1, guard_2 $\rightarrow$ action_2, ... guard_n $\rightarrow$ action_n}
Known Uses	
Consequences	
Depends on	
Is part of	One of several programming language statements
See also	Case Statement
Credits	

When the alternatives are mutually exclusive use the choice composition as given. When a value fits into more than one case, sequential composition is used determine when cases are applicable.

Definite Iteration

Name	Definite Iteration
Also known as	for loop, do loop
Context	
Problem	The for loop is used whenever a sequence of actions is to be repeated a fixed/known number of times.

Solution The for loop comes in two varieties, one that counts down and one that counts up.
for *index* := *start* to *finish* do *body*
for *index* := *start* downto *finish* do *body*
Do Something A Specified Number Of Times: used when it is known in advance how many data items must be processed or how many times an action must occur.
for *Index := 1* to *Number of Items* do *perform some action* end

Indefinite Iteration

Name Indefinite Iteration
Also known as while loop, repeat-until, do-while, do-loop, guarded loop
Context The while loop is used whenever an action is to be repeated zero or more times for an indefinite number of iterations. The condition is evaluated and if it is true, the body is executed otherwise control passes to the following statement. After the body is executed the statement is repeated.
Problem
Solution while *condition* do *body*
Input and process until done: used when there are zero or more items to be processed but the number of items to be processed is not known.
while *not finished* do *get an item; process it* end
Sentinel controlled loop: used when the number of items is not known in advance and the last item is distinct from the preceding items (e.g., process a sequence of positive numbers and the last item *the sentinel* is negative).
get first value while *value != sentinel* do *process it; get next value* end
Accumulate Values Until Done:

```
initialize the accumulator
while there are values do
Accumulate the value
end
```

for example – compute the average of a sequence of numbers.
Semantics Indefinite Iteration Axiom:

$$\frac{\{P \wedge \neg C \rightarrow Q\},\ \{P \wedge C\}\ S\ \{P\}}{\{P\}\ \text{while}\ C\ \text{do}\ S\ \{Q\}}$$

Variants Repeat-Until: Repeat *body* until condition
Guarded Loop:

```
do  guard_1 → action_1,
guard_2 → action_2,
...
guard _n → action_n
```

Output

Name Output Operation

Example The output operation occurs in one of two forms: write(*expressions*), writeln(*expressions*) where the output expressions are separated by commas.

Context

Problem The order in which outputs are performed is important.

Solution The values of the expressions must be defined. The *write* form is often used to display a prompt and is then followed by an input operation.

Known Uses Typical uses include:

- output the value of a variable or expression: write(2*pi*radius)
- label and display a value:
  writeln('The circumference of a circle with radius ', radius:1:3, ' is ', 2*pi*radius:1:3)
- prompt the user for input: write('Enter a number: ')
- display a menu:

```
writeln(' Option Menu');
writeln;
writeln('D - Display Result');
writeln('I - Input item');
writeln('M - Display this menu')
writeln('P - Process data');
writeln('Q - Quit');
```

The *writeln* form terminates a line of output so that future output begins on the next line. The *write* form does not terminate a line of output so that future output continues on the same line.

Sequential Composition

Name	Sequential Composition
Also known as	Compound statement
Context	Linear sequence is used to compose a linear sequence of actions or when parallel composition is not available and concurrent execution is not required.
Problem	When a program is under construction, it is easy to forget to include the *END*. It is good coding practice to always code *BEGIN* and *END* as pairs and then enter the sequence of statements. Each statement must be separated from the next statement by a semicolon (the linear sequence operator).

Solution	BEGIN
	statement;
	statement;
	...
	statement
	END
	Some common applications of linear sequence include:
	Input, Process, Output: used when all of the data must be available before processing can begin.
	readln(Item);
	Result := ...Item...;
	writeln(Result)
	Prompt, Then Read: used in interactive programs to prompt the user for input.
	write('Enter a number: ');
	readln(number);

Semantics	Sequential Composition Axiom:

$$\frac{\{P\}\ C_0\ \{Q\},\ \{Q\}\ C_1\ \{R\}}{\{P\}\ C_0; C_1\ \{R\}}$$

Parallel Composition

Parallel Composition

(P_0||...|| P_n)

is used to compose actions which may occur in no particular order. It is not available in traditional programming languages.

F.3 User Defined Constructs

- Data structures

- Functions
- Relations
- Procedures
- Classes and Objects

F.4 Threads

F.5 Sockets

F.6 Name Space and Scope

F.7 Exercises

1. Use a template to complete the following:
   (a) i++, ++i, y += a
   (b) Data types and structures
   (c) User defined constructs
   (d) Name space and scope
   (e) etc
2. Determine the correctness of a program or significant code fragement using algebraic semantics.

Appendix G

Programming Styles

G.1 Imperative and Procedural

- Recursive

G.2 Declarative

- Functional
- Relational
  - Logic
  - Database
  - Constraint

G.3 Object-Oriented

...

Gardner[32] suggests that the general idea of inheritance consists of five relationships.

1. *Specialization* is used to create subtypes.
2. *Variation* is used to create different implementations. Variants contain the same methods.
3. A *view* provides different methods for accessing and manipulating an object.
4. *Construction* is used to combine two or more components into a system.
5. *Evolution* is used to facilitate change. The new class replaces the previous class.

G.4 Engine-Driven Computing

Engine-driven computing is type of declaritive programming. It consists of a compute engine which uses user provided rules to perform specific tasks. Examples of engine-driven computing include interpreters, scanners, parsers, and inference engines.

Finite State Machine

Example Application: Scanner
Implementation: gotos, loop with case statement, loop and array, recursive functions,

Push-down Machine

Example Application: Parser
Implementation: stack machine, recursive functions

Inference Engine

Example Application: Expert System

G.5 Other

- Implicit and explicit concurrency
- Reactive (event-driven)
- Distributed
- ROC
- Distributed
- Concurrent
  - Shared memory
  - Message passing
    - Client-server
    - Peer to peer
  - Distributed
  - Threads

G.6 Exercises

1. Pick a problem and construct a solution in each programming style.
2. Illustrate each type of inheritance.

Appendix H

Idioms - Low Level Patterns

H.1 Loop termination

Loop termination

- Counter
- Sentinal

H.2 Error handling

Error handling

- Assertions and goto
- Try, Raise, and Catch

H.3 Memory Management

Memory management for languages without garbage collection.

- new - dispose

H.4 Assertions

Assertions for program correctness

Name	*Assertion*
Description	Assertions are boolean expressions that are evaluated during the execution of a program to insure that both the user and the software are fulfilling their respective obligations.

Example The following definition of factorial will fail to terminate for negative input and for input values that cause overflow.

```
function factorial ( n : integer ) : integer
   if n = 0 then 1
   else n*factorial (n-1)
```

Context Assertions assist in checking that parameters are of the proper type and within the range that permits a module to compute correct results. Also for verifying that loop invariants and data invariants are maintained.

Problem A software system is a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations. The problem is to find a way to enforce the obligations.

Solution

```
function assert (expression : Bool ) : Bool
   if expression then True
   else False
```

Assertions are a programming technique designed to facilitate the enforcement of the obligations. The specifications are the provisions of the contract and are called *assertions* and are expressed as boolean expressions. The specifications of the mutual obligations constitute a contract. The enforcement of the contract is accomplished using the notion of an assertion. An *assertion* is a boolean expression about the state of a software system. Assertions are inserted into the code and are checked during the system's execution. If any assertion evaluates to false, the system is invalid.

Example resolved

```
function factorial ( n : integer ) : integer
   if assert( 0 < n and n < max ) then
      if n = 0 then 1
      else n*factorial (n-1)
```

Variants The assert statement may generate a side effect either printing or logging the error message.

Known Uses Assertions may be used to provide assurance that

- the preconditions and postconditions in modules,
- loop invariants, and
- data invariants

are satisfied at execution time.

Consequences Assertions help to verify the correctness of a program both during the implementation and maintenance phases.

See also Error handling facilities in programming languages.

Credits Programming by Contract is a trademarked approach using assertions to insure program correctness.

Appendix I

Algorithms and Data Structures

- Data structures and associated algorithms
- Databases and relational knowledge
- STL

Note: in the following notation is abused. The symbol for a set of elements is also used for an element of the set.

I.1 Introduction

Computer memory is composed of a collection of words each of which may be accessed using an natural number 0..max. The memory may hold either data or instuctions. The data and instructions may occupy one or more words of memory. Recurring patterns of instructions are identified as algorithms. Recurring patterns of data are identified as data structures. Every data structure must be created and if dynamic, expand and contract, allow for

- Create
- If dynamic, expand and contract.
- Insert item into the data structure
- Modify the contents of the data structure
- Insert an item into the data structure.
- Look up an item in the data structure for an item which satisfies some constraint.

- Restructure the data structure

Values
Data Design

I.2 Data Structures and Associated Algorithms

I.3 C++ Standard Template Library

I.4 Maps, Arrays and Tables

Map

Definitions

Bool ::= false | true
D ::= set of domain values
C ::= set of codomain values
M = D → C the set of functions from D to C

- Empty : () → M
- isEmpty : M → Bool
- Codomain : D × M → C
- isDefined : D × M → B
- Assign : D × C × M → M
- Deassign : D × M → M

Array

Definitions

Bool ::= false | true
D ::= set of domain values
C ::= set of codomain values
A = D → C the set of functions from D to C

- Empty : () → A
- Codomain : D × M → C
- Assign : D × C × M → M

I.5 Lists, Stacks, Queues, and Dequeues

If finite, then there must be an isFull test.

Lists

Definitions

Bool ::= false | true
E ::= set of elements
L ::= empty | E L

- Empty : () → L
- isEmpty : L → Bool
- Head : L → E
- Tail : L → L
- Attach : E × L → L
- and various other operations as desired.
  - append : L × L → L
  - reverse : L → L

Stacks

Definitions

Name	*Stack*
Also known as	LIFO (Last in, First out) Queue
Context	Implementation of a programming language, expression evaluation, and looking for efficiency.
Problem	Have recursive functions or procedures and need to improve the runtime efficiency. Have tree or graph data structure and need a non-recursive algorithm to process the data structure. Need to implement function calls.
Solution	*Algebraic definition of a Stack* Domains: Item ::= some collection of items Stack :: = Nil | Item Stack Bool ::= False | True Functions: newStack: () → Stack push : Item × Stack → Stack pop: Stack → Stack top: Stack → Item empty : Stack → Bool

	Axioms: Let I be an Item and S a Stack. newStack() = Nil pop(push(I, S)) = S push(I, pop(I S)) = I S top(push(I, S)) = I empty(push(I S)) = False empty(newStack()) = True Defining Equations: Let I be an Item and S a Stack. newStack() = Nil push(I, S) = I S pop(I S) = S top(I S) = I empty(I S) = False empty(Nil) = True Errors: pop(newStack()) or pop(Nil) top(newStack()) or top(Nil)
Variants	Stacks may be implemented using arrays or linked lists. They may be explicit or implicit.
Known Uses	Depth-first or breadth-first search of a tree data structure. Finding a path between two nodes in a graph. Convert arithmetic expressions to post fix notation. Hold local data for procedures and functions. Hold return addresses for procedures and functions.

Implementations

- Array
- Linked List

Algorithms

Applications

Queue

Definitions

Q ::= Empty | Element Q
Element ::= some element type
Bool ::= False | True

- Empty: () → Q
- IsEmpty: Q → Bool

- Head: Q $\rightarrow$ E
- Dequeue: Q $\rightarrow$ Q
- Enqueue: E $\times$ Q $\rightarrow$ Q

Implementations

- Array (circular)
- Linked list

Algorithms

Applications

Dequeue

Definitions

- Empty
- Head
- EnqueueHead
- DequeueTail
- Tail
- EnqueueTail
- DequeueTail

Implementations

Algorithms

Applications

I.6 Ordered Lists and Sorted Lists

Definitions

Implementations

Array

Linked List

Algorithms

Applications

I.7 Hashing, Hash Tables, and Scatter Tables

I.8 Trees

Binary, 2-3, AVL

Insert from `../Math/Graph.html`

Definitions

Implementations

Algorithms

Applications

I.9 Search Trees

I.10 Heaps and Priority Queues

I.11 Sets, Multi-sets, and Partitions

I.12 Automatic Dynamic Memory Management - Heap

- Allocation
- Free
- Fragmentation: Compaction

- Reclamation: Garbage collection
- Algorithms
  - Reference counting
  - Mark and sweep
  - Copying
- Parallel

I.13 Graphs

Definitions

Insert definitions from ../Math/Graph.html

Implementations

- Adjacency matrix
- Adjacency list

I.14 Dictionary

I.15 Algorithms

- Algorithmic analysis
- Asymptotic notation

Algorithm Design Process
General algorithmic patterns

Data-Driven Programming

Brute-Force and Greedy Algorithms

Brute-force algorithms are distinguished approaching the solution of a problem in the most natural, obvious, simple, or direct method in contrast to a more clever or sophisticated way. Exhaustive enumeration is an example of a brute-force algorithm.

Greedy algorithms make decisions in some order and once the decision is made, the decision is not reconsidered. Greedy algorithms can run significantly faster than brute-force algorithms but may fail to generate a correct solution.

Examples: Count out the change using the smallest possible number pieces of notes and change.

- Brute-force: enumerate all possible combinations less than or equal to the desired amount and select the combination with the fewest pieces.

- Greedy algorithm: order the notes and coins from largest to smallest and select change in that order.

Backtracking Algorithms

Backtracking algorithms systematically consider all possible solutions (thus can be an example of a brute-force algorithm) but sometimes can detect when exhaustive search is unnecessary.

Examples:

- depth-first tree search
- breadth-first tree search

Top-Down Algorithms (Divide-and-Conquer)

Divide and conquer is an instance of a top-down problem solving method. It is distinguished by dividing a problem into one or more subproblems each of which is solved independently and the solutions are combined in order to obtain the solution.

Examples:

1. Binary search
2. Computing Fibonacci numbers
3. Merge sorting
4. Top-down parsing (Recursive descent parsing)
5. Matrix multiplication

Dynamic Programming

Dynamic programming is an instance of a bottom-up problem solving method. It is distinguished by solving a series of subproblems and combining the solutions to solve the given problem.

Examples:

1. Bottom-up parsing (LR, LALR etc Unix tools YACC and Bison)
2. Bottom-up Fibonacci number computation
3. Computing Binomial Coefficients

Randomized Algorithms

Randomized algorithms utilize an element of randomness to solve a given problem.

Examples:

1. Monte Carlo Methods

Shared Memory Algorithms

Message Passing Algorithms

Message passing algorithms are used to solved problems by sending and receiving messages in an environment of multiple processes.

Examples:

- Client-server computing
- Networking algorithms

Numerical Algorithms

I.16 Basic Algorithms

Syntax directed algorithms: Parsing

- Top-down parsing
  - Recursive descent
  - Table-driven top-down parsing
- Table-driven bottom-up parsing

Searching and Sorting

search: select; if not found then search

- Searching
  - Sequential search: selection is sequential in randomly ordered data
    - array w/wo sentinel
    - list
  - Binary search: selection is mid point in ordered data

Name	*Binary Search*
Context	Have an ordered list of items stored in an array with indices running from Low to High inclusive.
Problem	Need to find, in an ordered list, the location of an item that corresponds to a target.
Solution	Assume $-1 < \mathrm{L} \leq \mathrm{U}$ where L and U are the indices of the sequence of cells to be searched, t is the target item, A is the array of cells, and the search returns either the location of the target item or -1 to indicate that the target is not within the search range.

```
function bSearch (
   t : item,
   A : array [L..U] of items) : integer
   //Precondition: $-1 < L \leq U$, A[i] $\leq$ A[i+1] i = L..U-1,
   //Postcondition: A[L] = t or t = -1 and t $\notin$ A[L..U]
   if L=U then
      if A[L] = t then L else -1
   else M = (L+U) div 2
      if t $\leq$ A[M] then bSearch(t, A[L..M])
      else bSearch(t, A[M+1..U])
```

Complexity	Binary search is an O(log n) algorithm.
Variants	Binary search is simplified when the data structure is a binary search tree.
See also	Linear search is used when the list is not ordered. It is an O(n) algorithm.

- Sorting – sort: if not ordered then permute; sort
  - Insertion sorting
    - Straight insertion sort
    - Binary (tree) insertion sort
  - Exchange sorting
    - Bubble sort
    - Quicksort
  - Selection Sorting
    - Straight Selection Sorting
    - Heap Sorting
  - Merge Sorting
  - Distribution Sorting
    - Bucket sort
    - Radix sort
  - External Sorts
- Graph
  - Depth-first traversal
  - Breadth-first traversal
  - Topological sort
  - Testing for cycles
  - Testing for connectedness
  - Shortest-path algorithms
    - Single-source shortest path - Dijkstra's algorithm

    - All pairs shortest path - Floyd's algorithm
  - Minimum-Cost spanning trees
    - Prim's algorithm
    - Kruskal's algorithm

Memory Managment

Parallel and Distributed

I.17 References

Preiss, Bruno R. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++* Wiley 1999.

I.18 Exercises

1. Use a template to define a data structure and its associated algorithms.
2. Pick an algorithm and determine its type.
3. Pick an algorithm and show that it is correct.

Appendix J

Functional Components

There are three different classes of architectural elements: processing elements; data elements; and connection elements. The processing elements are those components that supply the transformation on the data elements. The data elements are those that contain the information that is used and transformed. The connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together.

The three classes of architectural design elements may be classified as functional elements. The functional components in a system can be classified as follows (from Sommerville[82])

- **Sensor** - Sensors collect information from a system's environment e.g. input statements.
- **Actuator** - Actuators cause some change in the system's environment e.g., output statements, assignment operation.
- **Computation component** - Computation components are components which, given some input, carry out some computations on that input and produce some output.
- **Communication component** - Communication components are components that facilitate communication between other components e.g., global variables, data structures, arguments, sockets, TCP/IP protocols.
- **Coordinator** - Coordination components are system components that coordinate the operation of other components, e.g., control structures, procedure call, semaphores, OS scheduler, a state machine.
- **Interface** - Interface components are components that transform the representation used by one system component into the representation used by another component e.g., GUI

Appendix K

A summary of the GOF design patterns

From Gamma et. al *Design patterns elements of reusable object-oriented software* Addison-Wesley 1995

Values Reuse, Flexibility, Modularity, UnderstandabilityDesign patterns are communicating objects and classes that are customized to solve a general design problem in a particular context. Design patterns deal with *micro-architectures* (also known as *object structures*) – static and dynamic relations among objects (and/or their classes) encountered in object-oriented development.

K.1 Creational Patterns

AbstractFactoryPattern Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

BuilderPattern Separate the construction of a complex object from its representation so that the same construction process can create different representations

FactoryMethodPattern Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

PrototypePattern Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

SingletonPattern Ensure a class only has one instance, and provide a global point of access to it. Idea - Have an encapsulated static variable holding the single instance of a class. Provide a static get-operation that creates the single instance once and returns it from then on.

K.2 Structural Patterns

AdapterPattern Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

BridgePattern Decouple an abstraction from its implementation so that the two can vary independently.

CompositePattern Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

DecoratorPattern Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

FacadePattern Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

FlyweightPattern Use sharing to support large numbers of fine-grained objects efficiently.

ProxyPattern Provide a surrogate or placeholder for another object to control access to it.

K.3 Behavioral Patterns

ChainOfResponsibilityPattern Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

CommandPattern Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

InterpreterPattern Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

IteratorPattern Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

MediatorPattern Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

MementoPattern Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

ObserverPattern Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

StatePattern Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

StrategyPattern Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

TemplateMethodPattern Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

VisitorPattern Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Appendix L

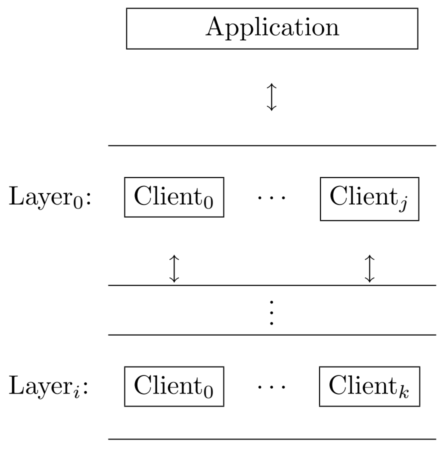
Architectural Patterns

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them [13].

Note: need diagrams ...

L.1 From Mud to Structure

Layers - This architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.



Pipes & Filters - This architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allow you to build families of related systems.

Input: $\rightarrow \boxed{\text{Process}_0} \rightarrow$ ... $\rightarrow \boxed{\text{Process}_n} \rightarrow$ Output

Blackboard - This architectural pattern is useful for problems for which no deterministic solution strategies are known. Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

$$\begin{array}{ccc}
\boxed{\text{Process}_0} & \cdots & \boxed{\text{Process}_n} \\
\updownarrow & \updownarrow & \updownarrow \\
\multicolumn{3}{c}{\boxed{\text{Blackboard}}}
\end{array}$$

Repository

L.2 Distributed Systems

Broker - This architectural pattern is a structure for distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

$$\begin{array}{ccccc}
\boxed{\text{Clent}_0} & \searrow & & \nearrow & \boxed{\text{Service}_0} \\
\vdots & \rightarrow & \boxed{\text{Broker}} & \rightarrow & \vdots \\
\boxed{\text{Clent}_i} & \nearrow & & \searrow & \boxed{\text{Service}_j}
\end{array}$$

Client-Server -

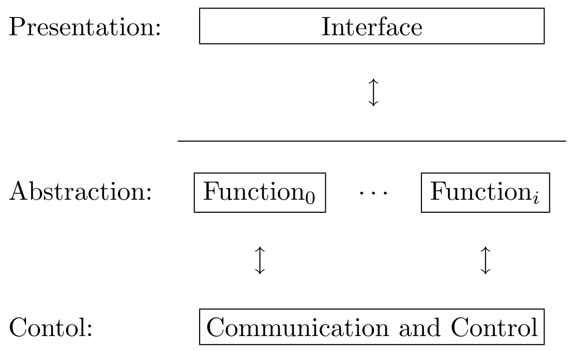
$$\begin{array}{ccc}
\multicolumn{3}{c}{\boxed{\text{Server}}} \\
\updownarrow & \updownarrow & \updownarrow \\
\boxed{\text{Client}_0} & \cdots & \boxed{\text{Client}_n}
\end{array}$$

L.3 Interactive Systems

Model-View-Controller (MVC) - This architectural pattern divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism insures consistency between the user interface and the model.

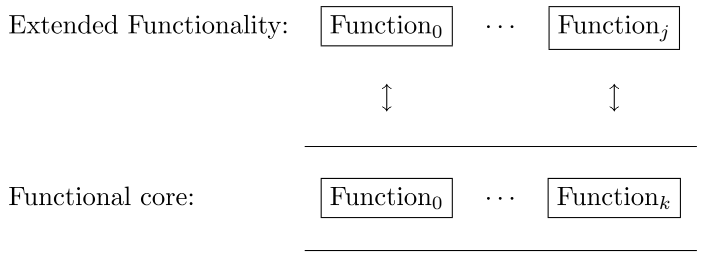
$$\begin{array}{lcc}
\text{User Interface:} & \boxed{\text{Controller}} & \boxed{\text{View}} \\
 & \downarrow & \uparrow \\
 & \multicolumn{2}{c}{\boxed{\text{Model}}}
\end{array}$$

Presentation-Abstraction-Control (PAC) - This architectural pattern defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.



L.4 Adaptable Systems

Microkernel - This architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.



Reflection - This architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

Appendix M

Enterprise Application Patterns

Enterprise applications are often constructed in three layers, presentation, domain, and data source. The layers may be logical or imply physical separation. The implementation may vary in complexity from a single script, separate procedures, separate classes, to separate packages. A number of patterns are used in the construction of enterprise applications. These enterprise application patterns are identified, organized, and described in Fowler[29].

M.1 Domain Logic Patterns

The domain layer (logic) is sometime called the business logic layer. It is between the presentation and domain layers. It performs data validation, intermediate calculations and dispatches the appropriate data source.

- Transaction Script
- Domain Model
- Table Module
- Service Layer

M.2 Data Source Logic Patterns

The data source layer (logic) provides the API for the actual tasks. Often it is a database.

Data Source Patterns

- Table Data Gateway
- Row Data Gateway
- Active Record
- Data Mapper

Object-Relational Patterns

Behavioral Patterns

- Item of Work
- Identity Map
- Lazy Load

Structural Patterns

- Identity Field
- Foreign Key Mapping
- Association Table Mapping
- Dependent Mapping
- Embedded Value
- Serialized LOB
- Single Table Inheritance
- Class Table Inheritance
- Concrete Table Inheritance
- Inheritance Mappers

Metadata Mapping Patterns

- Metadata Mapping
- Query Object
- Repository

M.3 Presentation Logic Patterns

The presentation layer (logic) handles that interaction between the user and the software. The user interface (UI) options include rich-client, web-browser, and commandline.

Web Presentation Patterns

- Model View Controller
- Page Controller
- Front Controller
- Template View
- Transform View
- Two Step View
- Application Controller

M.4 Supporting Patterns

Distribution Patterns

Distribution patterns are used to describe coarse grained applications on multiple machines.

- Remote Facade
- Data Transfer Object

Off-line Concurrency Patterns

Off-line concurrency refers to concurrency control for data that is manipulated-during multiple data during multiple database transactions.

- Optimistic Off-line Lock
- Pessimistic Off-line Lock
- Coarse-Grained Lock
- Implicit Lock

Session State Patterns

The session state patterns provide for the retention of state during transactions.

- Client Session State
- Server Session State
- Database Session State

Base Patterns

- Gateway
- Mapper
- Layer Supertype
- Separated Interface
- Registry
- Value Object
- Money
- Special Case
- Plugin
- Service Stub
- Record Set

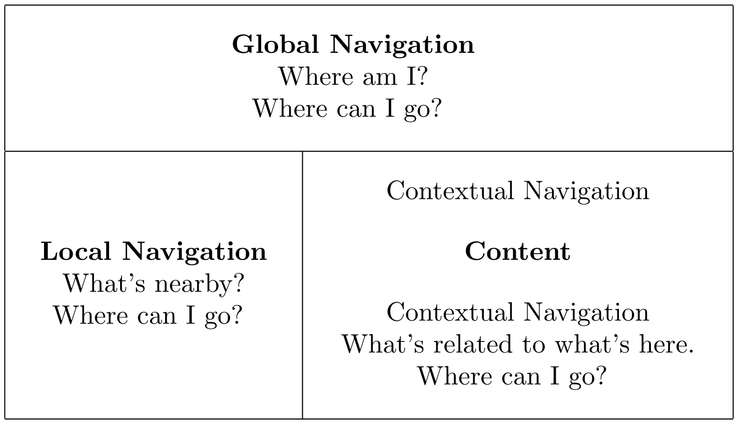
Appendix N

GUI Patterns

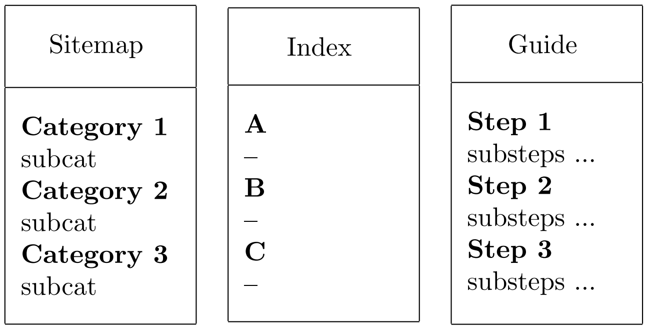
Visual design elements are derived from the geometric elements (points, lines, planes, shapes, focal point), scale, texture, value, color, and space. The GUI design community has developed a collection of widgets to facilitate the creation of interactive interfaces. Widget lists usually include:

- Pull-down/Drop-down Menus - dynamic expansion of an item
- Push Buttons - A labeled or iconified object that responds to clicks or enter
- Icons - represent objects or used as metaphors for functions or actions. Often “clickable” as with buttons.
- Checkboxes -
- Radio buttons -
- Scrolling lists -
- Text Field -
- Popup List -
- Spin boxes -
- Sliders -

Embedded navigation systems are often implemented in the following form:



Supplemental navigation systems are often implemented using the following forms:



Appendix O

Website Patterns

O.1 Organizational Structures

- Hierarchy – A hierarchical organization provides for efficient search when the subhierarchies are unrelated and the hierarchy is a balanced tree.
- Database – A database organization is best for a domain with a large number of similar items.
- Network – A network structure is best for a domain with numerous interconnections and complex or ambiguous relationships.

O.2 Navigation Systems

Embedded navigation systems include:

- Global – The global navigation system provides the user with contextual information for the entire site. This is most likely, the top level context for a hierarchical organization.
- Local – The local navigation system provides the user with contextual information for a localized collection of pages. The current level of a hierarchical organization.
- Contextual – The contextual navigation system provides the user with navigation opportunities within the content page.

Supplemental navigation systems include:

- Index – An index is useful when the material has a natural ordering and users are likely to know what they want.
- Site maps –
- Guides –

O.3 Search Systems

Bibliography

[1] ACM Interactions. ACM.

[2] ACM Transactions on Iinformation and System Security. ACM.

[3] ACM Queue Vol. 1 No. 6. ACM, September 2003.

[4] 4PM.COM. `http://www.4pm.com`.

[5] Anthony A. Aaby. Self-aware computing. `http://www.cs.wwc.edu/~aabyan/Logic/System.html`.

[6] Anthony A. Aaby. Lecture Notes on the SWEBOK. `http://www.cs.wwc.edu/~aabyan/435/intro.html`, Walla Walla College, 2002.

[7] David J. Agans. The 9 indispensable rules for finding even the most elusive software and hardware problems. In *AMACOM*, 2002.

[8] Atluri. *Security of Data and Transaction Processing.* Kluwer Academic Publications, 2000.

[9] Kent Beck. *Extreme Programming Explained.*

[10] Matt Bishop. Robust computing. CS Department, UC Davis, 199.

[11] Frederik Brooks. Three great challenges for half-century-old computer science. *Journal of the ACM*, 50(1):25–26, January 2003.

[12] Brown et. al. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.

[13] Buschmann et. al. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[14] Silvano Castano, editor. *Database Security.* Addison-Wesley, 1994.

[15] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM TOC*, 21(3):236–26, August 2003.

[16] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, 1994.

[17] Clayton M Christensen. The rules of innovation. *MIT Technology Review*, June 2002. A review of: *The Innovator's Dilemma* Harvard Business School Press. 1997.

[18] Coleman et. al. *Object-Oriented Development: the Fusion Method*. Prentice-Hall.

[19] Sue A. Conger and Richard O. Mason. *Planning and Designing Effective Web Sites*. ITP, 1998.

[20] Charles Connell. Most software stinks. `http://www.chc-3/pub/beautifulsoftware.htm`.

[21] Construx.com. `http://www.construx.com`.

[22] James Cooper. *Java design patterns: a tutorial*. Addison-Wesley, 2000.

[23] E. DeBono. *Six Hats Thinking*. Little, Brown and Co, 1985.

[24] Delphi Research. The information value chain. `http://www.delphigroup.com`, 2004.

[25] Keith Devlin. Why universities require computer science students to take math. *Communications of the ACM*, 46(9):37–39, September 2003.

[26] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hal, 1976.

[27] Kevin Dooley. *International Encyclopedia of Business and Management*, chapter Organizational Complexity, pages 5013–5022. Thompson Learning. `http://www.public.asu.edu/~kdooley/papers/iebm.PDF`.

[28] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[29] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[30] Fox, Jordan, Katz, Paterson, Shenker, and Stoica. Radlab technical vision. `http://radlab.cs.berkeley.edu/w/uploads/2/23/RADLabWhite.pdf`, December 2005.

[31] Gamma et. al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[32] Tracy Gardner. *Inheritance relationships for disciplined software construction*. Springer-Verlag, 2002.

[33] D Garvin. What does 'product quality' really mean? *Sloan Management Review*, Fall 1984.

[34] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7:131–174, 1996.

[35] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[36] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, July 2000.

[37] Haimes, Schooff, and Chittister. A holistic management framework for software acquisition. *Acquisition Review Quarterly*, Winter 1997.

[38] Seth Hallem, David Park, and Dawson Engler. Uprooting software defects at the source. *ACM Queue*, November 2003.

[39] Lars Hallnäs and Johan Redström. From use to presence: On the expression and aesthetics of everyday computational things. *ACM Transactions on Computer-Human Interaction*, 9(2):106–124, June 2002.

[40] Austin Henderson. Design for what? six dimensions of activity. *ACM Interactions*, VII.5:17–22, Sept & Oct 2000.

[41] Michael Howard. A look inside the security development lifecycle at microsoft. *MSDN Magazine*, November 2005.

[42] James K. Huggins and Charle Wallace. An abstract state machine primer. Computer Science Technical Report CS-TR-02-04, Michigan Technological University, 2002.

[43] Watts Humphrey. *Introduction to the Team Software Process.* Addison-Wesley, 2000.

[44] L. Hyatt and L. Rosenberg. A software quality model and metrics for identifying project risks and assessing software quality. In *8th Annual Software Technology Converence*, Utah, 1996.

[45] IEEE. *IEEE Standards Software Engineering*, volume 1, chapter IEEE Std 1062, 1998 Edition I EEE Recommended Practice for Software Acquisition. 1999.

[46] Keith Instone. Stress test your site. *Web Review*, 1997. Also see `http://keith.instone.org/navstress`.

[47] IS-MCA. International Society for Mathematical and Computational Aesthetics. `http://www.rci.rutgers.edu/~mleyton/ISMA.htm`.

[48] ISO. ISO standard 9126 software quality characteristics.

[49] D. Jensen, J. Feland, M. Bowe, and B. Self. A 6-hats based team formation strategy: Development and comparison with an MBTI based approach. In *Proceedings of the ASEE Annual Conference*. ASEE, 2000.

[50] Thomas Kühne. *A Functional Pattern System for Object-Oriented Design.* Verlag Dr. Kovac, 1999.

[51] Leslie Lamport. A simple approach to specifying concurrent system. *Communications of the ACM*, 32:32–45, January 1981.

[52] Butler W Lampson. Hints for computer system design. *ACM Operating Systems Rev*, 15(5):33–48, Oct 1983. Reprinted in *IEEE Software* 1, 1 (Jan. 1984), pp 11-28.

[53] Doug Lea. Christopher Alexander: An introduction for object-oriented designers. *Software Engineering Notes*, 19(1), 1994.

[54] Tae Eun Lee. Architectural design and spiritual life: Illustration and parallels. In *30th International Seminar on the Integration of Faith and Learning*, Sahmyook University, Seoul Korea, June 16-28 2002.

[55] Steve Lipner and Michael Howard. The trustworthy computing security development lifecycle. Microsoft Corporation, March 2005. `http://msdn.microsoft.com/library`.

[56] Z. Michalewicz and D. B. Fogel. *How to solve it: Modern Heuristics.* Springer-Verlag, 2000.

[57] James W Moore. *Software Engineering Standards: A User's Road Map.* IEEE Computer Society Press, 1997.

[58] Mark Moriconi and Xiaolei Qian. Correctness and composition of software architecture. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, 1994.

[59] A. Newell and H. A Simon. Computer science as empirical enquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, March 1976.

[60] Donald Norman. *The Design of Everyday Things.* Doubleday, 1990.

[61] OpenBRR.org. Business readiness ration for open source BBR 2005 - RFC 1. `http://www.openbrr.org`.

[62] Mark C Paulk. Extreme programming from a CMM perspective. *IEEE Software*, November/December 2001.

[63] PMI Standards Committee. *A Guide to the Project Management Body of Knowledge (PMBOK Guide).* 2000. `http://http://www.pmi.org`.

[64] G Polya. *How to Solve It.* Princeton University Press, 2nd edition, 1957.

[65] Eric S Raymond. *The Art of Unix Programming.* Addison-Wesley, 2003. `http://www.catb.org/~esr/writings/taoup`.

[66] Resenfeld and Morville. *Information Architecture for the World Wide Web.* O'Reilly, 2002.

[67] ROC. The Berkeley/Stanford recovery-oriented computing (ROC) project. `http://roc.cs.berkeley.edu`.

[68] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[69] Bruce Schneier. *Secrets and Lies*. John Wiley & Sons, 2000.

[70] Lui Sha. Using simplicity to control complexity.

[71] Shanks, Tansley, and Weber. Using ontology to validate conceptual models. *Comunications of the ACM*, 46(10), October 2003.

[72] Ben Shneiderman. *Designing the User Interface*. 3rd edition.

[73] Ben Shneiderman. Creating creativity: User interfaces for supporting innovation. *ACM Transactions on Computer-Human Interaction*, 7(1):114–138, March 2000.

[74] Ben Shneiderman. Understanding human activities and relationships. *Interactions*, 2002.

[75] Ben Shneiderman. Universal usability with multi-layer interface design. In *Proceedings of the 2003 Conference on Universal Usability*, November 2003.

[76] Joel Sklar. *Principles of Web Design*. ITP, 2000.

[77] Software Engineering Institute. `http://www.sei.cmu.edu/ata`.

[78] Software Engineering Institute (SEI). Risk management. `http://www.sei.cmu.edu/risk/`.

[79] Software Engineering Institute (SEI). Software acquisition capability maturity model. `http://www.sei.cmu.edu/arm/SA-CMM.html`, 2002.

[80] Software Engineering Standards Committee (SESC). *IEEE Standards Software Engineering*, volume 1-4. IEEE, 1999.

[81] Software Program Managers Network. `http://www.spmn.com`.

[82] Ian Sommerville. *Software Engineering*. 5th edition.

[83] D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering*, April 1995.

[84] Diomidis Spinellis. Reading, writing, and code. *ACM Queue*, October 2003.

[85] Daniel H Steinberg and Daniel W Palmer. *Extreme Software Engineering: a hands on approach*. Pearson Education Inc, 2004.

[86] SWEBOK. *Guide to the Software Engineering Body of Knowledge*. May 2001. Stone Man Trial Version 1.00.

[87] Marlene L. Theriault and William Heney. *Oracle Security.* O'Reilly and Associates, 1998.

[88] Bruce Tognazzini. First principles of interaction design. `http://www.asktog.com/basics/firstPrinciples.html`.

[89] University of Strathclyde (Glasgow). Risk management of e-learning project. `http://www.predict.strath.ac.uk/projects/risk/`.

[90] URLs.

- Security Forum - Security Design Patterns `http://www.opengroup.org/security/gsp.htm`
- Security Patterns.Org `http://www.securitypatterns.org`
- Common Front Group `http://cfg.cit.cornell.edu/cfg/design/contents.html`
- Tog. Interactive Design Solutions. `http://www.asktog.com`
- IBM Ease of Use `http://www.ibm.com/ibm/easy`
- Web Accessibility Initiative `http://www.w3.org/WAI`

.

[91] Wim. Van Grembergen. The Balanced Scorecard and IT Governance. *Information Systems Control Journal.*

[92] John Viega and Gary McGraw. *Building Secure Software.* Addison-Wesley, 2002.

[93] Richard A Watson. *Compositional Evolution.* MIT Press, 2006.

[94] Welie.com. Web and GUI design patterns. `http://www.welie.com`.