SEN 212 SOFTWARE REQUIREMENT AND DESIGN LECTURE NOTE

SOFTWARE REQUIREMENTS

INTRODUCTION

The Software Requirements is concerned with the elicitation, analysis, specification, and validation of software requirements as well as the management of requirements during the whole life cycle of the software product. It is widely acknowledged amongst researchers and industry practitioners that software projects are critically vulnerable when the requirements-related activities are poorly performed.

Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem.

Software Requirements is related closely to the Software Design, Software Testing, Software Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, Software Engineering Models and Methods, and Software Quality.

Software Requirements Fundamentals

Definition of a Software Requirement

At its most basic, a software requirement is a property that must be exhibited by something in order to solve some problem in the real world. It may aim to automate part of a task for someone to support the business processes of an organization, to correct shortcomings of existing software, or to control a device—to name just a few of the many problems for which software solutions are possible. The ways in which users, business processes, and devices function are typically complex. By extension, therefore, the requirements on particular software are typically a complex combination from various people at different levels of an organization, and who are in one way or another involved or connected with this feature from the environment in which the software will operate.

An essential property of all software requirements is that they be verifiable as an individual feature as a functional requirement or at the system level as a nonfunctional requirement. It may be difficult or costly to verify certain software requirements. For example, verification of the throughput requirement on a call center may necessitate the development of simulation software. Software requirements, software testing, and quality personnel must ensure that the requirements can be verified within available resource constraints.

Product and Process Requirements

A product requirement is a need or constraint on the software to be developed (for example, "The software shall verify that a student meets all prerequisites before he or she registers for a course").

A process requirement is essentially a constraint on the development of the software (for example, "The software shall be developed using a RUP process").

Some software requirements generate implicit process requirements. The choice of verification technique is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce faults that can lead to inadequate reliability. Process requirements may also be imposed directly by the development organization, their customer, or a third party such as a safety regulator.

Functional and Non-functional Requirements

Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or features. A functional requirement can also be described as one for which a finite set of test steps can be written to validate its behavior.

Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, security requirements, interoperability requirements or one of many other types of software requirements

Emergent Properties

Some requirements represent emergent properties of software—that is, requirements that cannot be addressed by a single component but that depend on how all the software components interoperate. The throughput requirement for a call center would, for example, depend on how the telephone system, information system, and the operators all interacted under actual operating conditions. Emergent properties are crucially dependent on the system architecture.

Quantifiable Requirements

Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements that depend for

their interpretation on subjective judgment ("the software shall be reliable"; "the software shall be user-friendly"). This is particularly important for nonfunctional requirements. Two examples of quantified requirements are the following: a call center's software must increase the center's throughput by 20%; and a system shall have a probability of generating a fatal error during any hour of operation of less than 1 * 10 −8. The throughput requirement is at a very high level and will need to be used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture.

System Requirements and Software Requirements

System means an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements, as defined by the International Council on Software and Systems Engineering (INCOSE).

System requirements are the requirements for the system as a whole. In a system containing software components, software requirements are derived from system requirements.

We will define "user requirements" in a restricted way, as the requirements of the system's customers or end users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

Requirements Process

This sub-topic introduces the software requirements process, orienting the remaining five topics and showing how the requirements process dovetails with the overall software engineering process.

Process Models

The objective of this topic is to provide an understanding that the requirements process

- is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project that continues to be refined throughout the life cycle;
- identifies software requirements as configuration items and manages them using the same software configuration management practices as other products of the software life cycle processes;
- needs to be adapted to the organization and project context.

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints. The topic also includes activities that provide input into the requirements process, such as marketing and feasibility studies.

Process Actors

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but will always include users/operators and customers (who need not be the same).

Typical examples of software stakeholders include (but are not restricted to) the following:

- **Users:** This group comprises those who will operate the software. It is often a heterogeneous group involving people with different roles and requirements.
- **Customers:** This group comprises those who have commissioned the software or who represent the software's target market.
- **Market analysts:** A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- **Regulators:** Many application domains, such as banking and public transport, are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- **Software engineers:** These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in or from other products. If, in this scenario, a customer of a particular product has specific requirements that compromise the potential for component reuse, the software engineers must carefully weigh their own stake against those of the customer.

Specific requirements, particularly constraints, may have major impact on project cost or delivery because they either fit well or poorly with the skill set of the engineers. Important tradeoffs among such requirements should be identified. It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the software engineer's job to negotiate tradeoffs that are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for this is that all the stakeholders be identified, the nature of their "stake" analyzed, and their requirements elicited.

Process Support and Management

This section introduces the project management resources required and consumed by the requirements process. It establishes the context for the first topic (Initiation and Scope Definition) of the Software Engineering Management KA. Its principal purpose is to make the link between the process activities identified in 2.1 and the issues of cost, human resources, training, and tools.

Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality and Software Engineering Process, comprising

- requirements process coverage by process improvement standards and models;
- requirements process measures and benchmarking;
- improvement planning and implementation;
- security/CIA improvement/planning and implementation

Requirements Elicitation

Requirements elicitation is concerned with the origins of software requirements and how the software engineer can collect them. It is the first stage in building an understanding of the problem the software is required to solve. It is fundamentally a human activity and is where the stakeholders are identified and relationships established between the development team and the customer. It is variously termed "requirements capture," "requirements discovery," and "requirements acquisition."

One of the fundamental principles of a good requirements elicitation process is that of effective communication between the various stake- holders. This communication continues through the entire Software Development Life Cycle (SDLC) process with different stakeholders at different points in time. Before development begins, requirements specialists may form the conduit for this communication. They must mediate between the domain of the software users (and other stakeholders) and the technical world of the software engineer. A set of internally consistent models at different levels of abstraction facilitate communications between software users/stake holders and software engineers.

A critical element of requirements elicitation is informing the project scope. This involves providing a description of the software being specified and its purpose and prioritizing the deliverables to ensure the customer's most important business needs are satisfied first. This minimizes the risk of requirements specialists spending time eliciting requirements that are of low importance or those that turn out to be no longer relevant when the software is delivered. On the other hand, the description must be scalable and extensible to accept further requirements not expressed in the first formal lists and compatible with the previous ones as contemplated in recursive methods.

Requirements Sources

Requirements have many sources in typical software, and it is essential that all potential sources be identified and evaluated. This topic is designed to promote awareness of the various sources of software requirements and of the frameworks for managing them. The main points covered are as follows:

- **Goals:** The term "goal" (sometimes called "business concern" or "critical success factor") refers to the overall, high-level objectives of the software. Goals provide the motivation for the software but are often vaguely formulated. Software engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this
- **Domain knowledge:** The software engineer needs to acquire or have available knowledge about the application domain. Domain knowledge provides the background against which all elicited requirements knowledge must be set in order to understand it. It's a good practice to emulate an ontological approach in the knowledge domain. Relations between relevant concepts within the application domain should be identified.
- **Stakeholders:** Much software has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense of others. Hence, the delivered software is difficult to use, or subverts the cultural or political structures of the customer organization. The software engineer needs to identify, represent, and manage the "viewpoints" of many different types of stakeholders.
- **Business rules:** These are statements that define or constrain some aspect of the structure or the behavior of the business itself. "A student cannot register in next semester's courses if there remain some unpaid tuition fees" would be an example of a business rule that would be a requirement source for a university's course-registration software.
- **The operational environment:** Requirements will be derived from the environment in which the software will be executed. These maybe, for example, timing constraints in real-time software or performance constraints in a business environment. These must be sought out actively because they can greatly affect software feasibility and cost as well as restrict design choices.
- **The organizational environment:** Software is often required to support a business process, the selection of which may be conditioned by the structure, culture, and internal politics of the organization. The software engineer needs to be sensitive to these since, in general, new software should not force unplanned change on the business process.

Elicitation Techniques

An elicitation technique is any of a number of data collection techniques used in anthropology, cognitive science, counseling, education, knowledge engineering, linguistics, management, philosophy, psychology, or other fields to gather knowledge or information from people. Recent work in behavioral economics has purported that elicitation techniques can be used to control subject misconceptions and mitigate errors from generally accepted experimental design practices. Elicitation, in which knowledge is sought directly from human beings, is usually distinguished from indirect methods such as gathering information from written sources.

A person who interacts with human subjects in order to elicit information from them may be called an elicitor, an analyst, experimenter, or knowledge engineer, depending on the field of study.

Elicitation techniques include interviews, observation of either naturally occurring behavior (including as part of participant observation) or behavior in a laboratory setting, or the analysis of assigned tasks.

List of elicitation techniques

Interviews

Existing System

Project Scope

Brain Storming

Focus Groups

Exploratory Prototypes

User Task Analysis

Observation

Surveys

Questionnaire

Story Board

Requirements Analysis

This topic is concerned with the process of analyzing requirements to

- detect and resolve conflicts between requirements;
- discover the bounds of the software and how it must interact with its organizational and operational environment;

- elaborate system requirements to derive software requirements.

The traditional view of requirements analysis has been that it be reduced to conceptual modeling using one of a number of analysis methods, such as the structured analysis method. While conceptual modeling is important, we include the classification of requirements to help inform tradeoffs between requirements (requirements classification) and the process of establishing these tradeoffs (requirements negotiation).

Care must be taken to describe requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

Requirements Classification

Requirements can be classified on a number of dimensions. Examples include the following:

- Whether the requirement is functional or nonfunctional
- Whether the requirement is derived from one or more high-level requirements or an emergent property or is being imposed directly on the software by a stakeholder or some other source.
- Whether the requirement is on the product or the process (see section 1.2, Product and Process Requirements). Requirements on the process can constrain the choice of contractor, the software engineering process to be adopted, or the standards to be adhered to.
- The requirement priority. The higher the priority, the more essential the requirement is for meeting the overall goals of the software. Often classified on a fixed-point scale such as mandatory, highly desirable, desirable, or optional, the priority often has to be balanced against the cost of development and implementation.
- The scope of the requirement. Scope refers to the extent to which a requirement affects the software and software components. Some requirements, particularly certain nonfunctional ones, have a global scope in that their satisfaction cannot be allocated to a discrete component. Hence, a requirement with global scope may strongly affect the software architecture and the design of many components, whereas one with a narrow scope may offer a number of design choices and have little impact on the satisfaction of other requirements

Conceptual Modeling

The development of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding the situation in which the problem occurs, as well as depicting a solution. Hence, conceptual models comprise models of entities from the problem domain, configured to reflect their real-world relationships and dependencies.

Several kinds of models can be developed. These include use case diagrams, data flow models, state models, goal-based models, user interactions, object models, data models, and many others. Many of these modeling notations are part of the Unified Modeling Language (UML).

Use case diagrams, for example, are routinely used to depict scenarios where the boundary separates the actors (users or systems in the external environment) from the internal behavior where each use case depicts a functionality of the system.

The factors that influence the choice of modeling notation include these:

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously. For example, state and parametric models, which are part of SysML, are likely to be more important for real-time software than for information systems, while it would usually be the opposite for object and activity models.
- The expertise of the software engineer. It is often more productive to adopt a modeling notation or method with which the software engineer has experience.
- The process requirements of the customer. Customers may impose their favored notation or method or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

Architectural Design and Requirements Allocation

At some point, the solution architecture must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands that the architecture/design components that will be responsible for satisfying the requirements be identified. This is requirements allocation–the assignment to architecture components responsible for satisfying the requirements.

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem. As an example, requirements for a particular braking performance for a car (braking distance,

safety in poor driving conditions, smoothness of application, pedal pressure required, and so on) may be allocated to the braking hardware (mechanical and hydraulic assemblies) and an antilock braking system (ABS). Only when a requirement for an antilock braking system has been identified, and the requirements allocated to it, can the capabilities of the ABS, the braking hardware, and emergent properties (such as car weight) be used to identify the detailed ABS software requirements.

Requirements Negotiation

Another term commonly used for this subtopic is "conflict resolution." This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and nonfunctional requirements, for example. In most cases, it is unwise for the software engineer to make a unilateral decision, so it becomes necessary to consult with the stakeholder(s) to reach a consensus on an appropriate tradeoff. It is often important, for contractual reasons, that such decisions be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis.

Requirements prioritization is necessary, not only as a means to filter important requirements, but also in order to resolve conflicts and plan for staged deliveries, which means making complex decisions that require detailed domain knowledge and good estimation skills. However, it is often difficult to get real information that can act as a basis for such decisions. In addition, requirements often depend on each other, and priorities are relative. In practice, software engineers perform requirements prioritization frequently without knowing about all the requirements. Requirements prioritization may follow a cost value approach that involves an analysis from the stakeholders defining in a scale the benefits or the aggregated value that the implementation of the requirement brings them, versus the penalties of not having implemented a particular requirement. It also involves an analysis from the software engineers estimating in a scale the cost of implementing each requirement, relative to other requirements. Another requirements prioritization approach called the analytic hierarchy process involves comparing all unique pairs of requirements to determine which of the two is of higher priority, and to what extent.

Formal Analysis

Formal analysis has made an impact on some application domains, particularly those of high integrity systems. The formal expression of requirements requires a language with formally defined semantics. The use of a formal analysis for requirements expression has two benefits.

First, it enables requirements expressed in the language to be specified precisely and unambiguously, thus (in principle) avoiding the potential for misinterpretation.

Secondly, requirements can be reasoned over, permitting desired properties of the specified software to be proven. Formal reasoning requires tool support to be practicable for anything other than trivial systems, and tools generally fall into two types: theorem provers or model checkers.

In neither case can proof be fully automated, and the level of competence in formal reasoning needed in order to use the tools restricts the wider application of formal analysis. Most formal analysis is focused on relatively late stages of requirements analysis. It is generally counterproductive to apply formalization until the business goals and user requirements have come into sharp focus through means such as those described elsewhere in section 4. However, once the requirements have stabilized and have been elaborated to specify concrete properties of the software, it may be beneficial to formalize at least the critical requirements. This permits static validation that the software specified by the requirements does indeed have the properties (for example, absence of deadlock) that the customer, users, and software engineer expect it to have.

Requirements Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. In software engineering, "software requirements specification" typically refers to the production of a document that can be systematically reviewed, evaluated, and approved. For complex systems, particularly those involving substantial nonsoftware components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required. All three documents are described here, with the understanding that they may be combined as appropriate.

System Definition Document

This document (sometimes known as the user requirements document or concept of operations document) records the system requirements. It defines the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market driven software), so its content must be couched in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment, and a statement of the constraints, assumptions, and nonfunctional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios, and the principal domain entities, as well as workflows.

System Requirements Specification

Developers of systems with substantial software and non-software components—a modern airliner, for example—often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified.

Software Requirements Specification

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do as well as what it is not expected to do. Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules. Organizations can also use a software requirements specification document as the basis for developing effective verification and validation plans.

Software requirements specification provides an informed basis for transferring a software product to new users or software platforms. Finally, it can provide a basis for software enhancement.

Software requirements are often written in natural language, but, in software requirements specification, this may be supplemented by formal or semiformal descriptions. Selection of appropriate notations permits particular requirements and aspects of the software architecture to be described more precisely and concisely than natural language. The general rule is that notations should be used that allow the requirements to be described as precisely as possible. This is particularly crucial for safety-critical, regulatory, and certain other types of dependable software. However, the choice of notation is often constrained by the training, skills, and preferences of the document's authors and readers.

A number of quality indicators have been developed that can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule, and reproducibility. Quality indicators for individual software requirements specification statements include imperatives, directives, weak phrases, options, and continuances. Indicators for the entire software requirements specification document include size, readability, specification, depth, and text structure.

Requirements Validation

The requirements documents may be subject to validation and verification procedures. The requirements may be validated to ensure that the software engineer has understood the requirements; it is also important to verify that a requirements document conforms to company standards and that it is understandable, consistent, and complete. In cases where documented company standards or terminology are inconsistent with widely accepted standards, a mapping between the two should be agreed on and appended to the document.

Formal notations offer the important advantage of permitting the last two properties to be proven (in a restricted sense, at least). Different stakeholders, including representatives of the customer and developer, should review the document(s). Requirements documents are subject to the same configuration management practices as the other deliverables of the software life cycle processes. When practical, the individual requirements are also subject to configuration management, generally using a requirements management tool .

It is normal to explicitly schedule one or more points in the requirements process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements. Requirements validation is concerned with the process of examining the requirements document to ensure that it defines the right software (that is, the software that the users expect).

Requirements Reviews

Perhaps the most common means of validation is by inspection or reviews of the requirements document(s). A group of reviewers is assigned a brief to look for errors, mistaken assumptions, lack of clarity, and deviation from standard practice. The composition of the group that conducts the review is important (at least one representative of the customer should be included for a customer-driven project, for example), and it may help to provide guidance on what to look for in the form of checklists. Reviews may be constituted on completion of the system definition document, the system specification document, the software requirements specification document, the baseline specification for a new release, or at any other step in the process.

Prototyping

Prototyping is commonly a means for validating the software engineer's interpretation of the software requirements, as well as for eliciting new requirements. As with elicitation, there is a range of prototyping techniques and a number of points in the process where prototype validation may be appropriate. The

advantage of prototypes is that they can make it easier to interpret the software engineer's assumptions and, where needed, give useful feedback on why they are wrong. For example, the dynamic behavior of a user interface can be better understood through an animated prototype than through textual description or graphical models. The volatility of a requirement that is defined after prototyping has been done is extremely low because there is agreement between the stakeholder and the software engineer— therefore, for safety-critical and crucial features prototyping would really help. There are also disadvantages, however. These include the danger of users' attention being distracted from the core underlying functionality by cosmetic issues or quality problems with the prototype. For this reason, some advocate prototypes that avoid software, such as flip-chart-based mockups. Prototypes may be costly to develop. However, if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified. Early prototypes may contain aspects of the final solution. Prototypes may be evolutionary as opposed to throwaway.

Model Validation

It is typically necessary to validate the quality of the models developed during analysis. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects that, in the stakeholders' domain, exchange data. If formal analysis notations are used, it is possible to use formal reasoning to prove specification properties.

Acceptance Tests

An essential property of a software requirement is that it should be possible to validate that the finished product satisfies it. Requirements that cannot be validated are really just "wishes." An important task is therefore planning how to verify each requirement. In most cases, designing acceptance tests does this for how end-users typically conduct business using the system.

Identifying and designing acceptance tests may be difficult for nonfunctional requirements. To be validated, they must first be analyzed and decomposed to the point where they can be expressed quantitatively.

Practical Considerations

The requirements process spans the whole software life cycle. Change management and the maintenance of the requirements in a state that accurately mirrors the software to be built, or that has been built, are key to the success of the software engineering process. Not every organization has a culture of documenting and managing requirements. It is common in dynamic start-up companies, driven by a strong "product

vision" and limited resources, to view requirements documentation as unnecessary overhead. Most often, however, as these companies expand, as their customer base grows, and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management are key to the success of any requirements process.

Iterative Nature of the Requirements Process

There is general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive, market-driven sectors. Moreover, most projects are constrained in some way by their environment, and many are upgrades to, or revisions of, existing software where the architecture is a given. In practice, therefore, it is almost always impractical to implement the requirements process as a linear, deterministic process in which software requirements are elicited from the stakeholders, baselined, allocated, and handed over to the software development team. It is certainly a myth that the requirements for large software projects are ever perfectly understood or perfectly specified.

Instead, requirements typically iterate towards a level of quality and detail that is sufficient to permit design and procurement decisions to be made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This risks expensive rework if problems emerge late in the software engineering process. However, software engineers are necessarily constrained by project management plans and must therefore take steps to ensure that the "quality" of the requirements is as high as possible given the available resources. They should, for example, make explicit any assumptions that underpin the requirements as well as any known problems.

For software products that are developed iteratively, a project team may baseline only those requirements needed for the current iteration. The requirements specialist can continue to develop requirements for future iterations, while developers proceed with design and construction of the current iteration. This approach provides customers with business value quickly, while minimizing the cost of rework.

In almost all cases, requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point in understanding software requirements is that a significant proportion of the requirements will change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the "environment"— for example, the customer's operating or business environment, regulatory processes imposed by the authorities, or the market into which software must sell. Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by

ensuring that proposed changes go through a defined review and approval process and by applying careful requirements tracing, impact analysis, and software configuration management. Hence, the requirements process is not merely a front-end task in software development, but spans the whole software life cycle. In a typical project, the software requirements activities evolve over time from elicitation to change management. A combination of top-down analysis and design methods and bottom-up implementation and refactoring methods that meet in the middle could provide the best of both worlds. However, this is difficult to achieve in practice, as it depends heavily upon the maturity and expertise of the software engineers.

Change Management

Change management is central to the management of requirements. This topic describes the role of change management, the procedures that need to be in place, and the analysis that should be applied to proposed changes.

Requirements Attributes

Requirements should consist not only of a specification of what is required, but also of ancillary information, which helps manage and interpret the requirements. Requirements attributes must be defined, recorded, and updated as the software under development or maintenance evolves. This should include the various classification dimensions of the requirement and the verification method or relevant acceptance test plan section. It may also include additional information, such as a summary rationale for each requirement, the source of each requirement, and a change history. The most important requirements attribute, however, is an identifier that allows the requirements to be uniquely and unambiguously identified.

Requirements Tracing

Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backward to the requirements and stakeholders that motivated it (from a software requirement back to the system requirement(s) that it helps satisfy, for example). Conversely, a requirement should be traceable forward into the requirements and design entities that satisfy it (for example, from a system requirement into the software requirements that have been elaborated from it, and on into the code modules that implement it, or the test cases related to that code and even a given section on the user manual which describes the actual functionality) and into the test case that verifies it.

The requirements tracing for a typical project will form a complex directed acyclic graph (DAG) of requirements. Maintaining an up-to-date graph or traceability matrix is an activity that must be considered

during the whole life cycle of a product. If the traceability information is not updated as changes in the requirements continue to happen, the traceability information becomes unreliable for impact analysis.

Measuring Requirements

As a practical matter, it is typically useful to have some concept of the "volume" of the requirements for a particular software product. This number is useful in evaluating the "size" of a change in requirements, in estimating the cost of a development or maintenance task, or simply for use as the denominator in other measurements. Functional size measurement (FSM) is a technique for evaluating the size of a body of functional requirements.

Software Requirements Tools

Tools for dealing with software requirements fall broadly into two categories: tools for modeling and tools for managing requirements.

Requirements management tools typically support a range of activities—including documentation, tracing, and change management—and have had a significant impact on practice. Indeed, tracing and change management are really only practicable if supported by a tool. Since requirements management is fundamental to good requirements practice, many organizations have invested in requirements management tools, although many more manage their requirements in more ad hoc and generally less satisfactory ways (e.g., using spreadsheets).

SOFTWARE DESIGN

INTRODUCTION

Design is defined as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process". Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction. A software design (the result) describes the software architecture—that is, how software is decomposed and organized into components—and the interfaces between those components. It should also describe the components at a level of detail that enables their construction.

Software design plays an important role in developing software: during software design, software engineers produce various models that form a kind of blueprint of the solution to be implemented. We can analyze and evaluate these models to determine whether or not they will allow us to fulfill the various requirements. We can also examine and evaluate alternative solutions and tradeoffs. Finally, we can use the resulting models to plan subsequent development activities, such as system verification and validation, in addition to using them as inputs and as the starting point of construction and testing.

In a standard list of software life cycle processes, such as that in ISO/IEC/IEEE Std. 12207, Software Life Cycle Processes, software design consists of two activities that fit between software requirements analysis and software construction:

i. Software architectural design (sometimes called high-level design): develops top-level structure and organization of the software and identifies the various components.
ii. Software detailed design: specifies each component in sufficient detail to facilitate its construction.

Software Design Fundamentals

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of software design.

General Design Concepts

In the general sense, design can be viewed as a form of problem solving. For example, the concept of a wicked problem—a problem with no definitive solution—is interesting in terms of understanding the limits of design. A number of other notions and concepts are also of interest in understanding design in its general sense: goals, constraints, alternatives, representations, and solutions.

Context of Software Design

Software design is an important part of the software development process. To understand the role of software design, we must see how it fits in the software development life cycle. Thus, it is important to understand the major characteristics of software requirements analysis, software design, software construction, software testing, and software maintenance.

Software Design Process

Software design is generally considered a two-step process:

- Architectural design (also referred to as high-level design and top-level design) describes how software is organized into components.
- Detailed design describes the desired behavior of these components.

The output of these two processes is a set of models and artifacts that record the major decisions that have been taken, along with an explanation of the rationale for each nontrivial decision. By recording the rationale, long-term maintainability of the software product is enhanced.

Software Design Principles

A *principle* is "a comprehensive and fundamental law, doctrine, or assumption". Software design principles are key notions that provide the basis for many different software design approaches and concepts. Software design principles include abstraction; coupling and cohesion; decomposition and modularization; encapsulation/information hiding; separation of interface and implementation; sufficiency, completeness, and primitiveness; and separation of concerns.

- ***Abstraction*** is "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information". In the context of software design, two key abstraction mechanisms are parameterization and specification. Abstraction by parameterization abstracts from the details of data representations by representing the data as named parameters. Abstraction by specification leads to three major kinds of abstraction: procedural abstraction, data abstraction, and control (iteration) abstraction.
- ***Coupling and Cohesion***. Coupling is defined as “a measure of the interdependence among modules in a computer program,” whereas cohesion is defined as “a measure of the strength of association of the elements within a module”.
- ***Decomposition and modularization***. Decomposing and modularizing means that large software is divided into a number of smaller named components having well-defined interfaces that describe

component interactions. Usually the goal is to place different functionalities and responsibilities in different components.

- ***Encapsulation and information hiding*** means grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities.
- ***Separation of interface and implementation*.** Separating interface and implementation involves defining a component by specifying a public interface (known to the clients) that is separate from the details of how the component is realized (see encapsulation and information hiding above).
- ***Sufficiency, completeness, and primitiveness*.** Achieving sufficiency and completeness means ensuring that a software component captures all the important characteristics of an abstraction and nothing more. Primitiveness means the design should be based on patterns that are easy to implement.
- ***Separation of concerns*.** A concern is an "area of interest with respect to a software design". A design concern is an area of design that is relevant to one or more of its stakeholders. Each architecture view frames one or more concerns. Separating concerns by views allows interested stakeholders to focus on a few things at a time and offers a means of managing complexity.

2 Key Issues in Software Design

A number of key issues must be dealt with when designing software. Some are quality concerns that all software must address—for example, performance, security, reliability, usability, etc. Another important issue is how to decompose, organize, and package software components. This is so fundamental that all design approaches address it in one way or another. In contrast, other issues “deal with some aspect of software’s behavior that is not in the application domain, but which addresses some of the supporting domains”. Such issues, which often crosscut the system’s functionality, have been referred to as aspects, which “tend not to be units of software’s functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways”. A number of these key, crosscutting issues are discussed in the following sections (presented in alphabetical order).

Concurrency

Design for concurrency is concerned with decomposing software into processes, tasks, and threads and dealing with related issues of efficiency, atomicity, synchronization, and scheduling.

Control and Handling of Events

This design issue is concerned with how to organize data and control flow as well as how to handle reactive and temporal events through various mechanisms such as implicit invocation and call-backs.

Data Persistence

This design issue is concerned with how to handle long-lived data.

Distribution of Components

This design issue is concerned with how to distribute the software across the hardware (including computer hardware and network hardware), how the components communicate, and how middleware can be used to deal with heterogeneous software.

Error and Exception Handling and Fault Tolerance

This design issue is concerned with how to prevent, tolerate, and process errors and deal with exceptional conditions.

Interaction and Presentation

This design issue is concerned with how to structure and organize interactions with users as well as the presentation of information (for example, separation of presentation and business logic using the Model-View-Controller approach). Note that this topic does not specify user interface details, which is the task of user interface design.

Security

Design for security is concerned with how to prevent unauthorized disclosure, creation, change, deletion, or denial of access to information and other resources. It is also concerned with how to tolerate security-related attacks or violations by limiting damage, continuing service, speeding repair and recovery, and failing and recovering securely. Access control is a fundamental concept of security, and one should also ensure the proper use of cryptology.

Software Structure and Architecture

In its strict sense, a software architecture is "the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both". During the mid-1990s, however, software architecture started to emerge as a broader discipline that involved the study of software structures and architectures in a more generic way. This gave rise to a number of interesting concepts about software design at different levels of abstraction. Some of these concepts can be useful during the architectural design (for example, architectural styles) as well as during the detailed design (for example, design patterns). These design concepts can also be used to design families of programs (also known as product lines). Interestingly, most of these concepts can be seen as attempts to describe, and thus reuse, design knowledge.

Architectural Structures and Viewpoints

Different high-level facets of a software design can be described and documented. These facets are often called views: “A view represents a partial aspect of a software architecture that shows specific properties of a software system”. Views pertain to distinct issues associated with software design—for example, the logical view (satisfying the functional requirements) vs. the process view (concurrency issues) vs. the physical view (distribution issues) vs. the development view (how the design is broken down into implementation units with explicit representation of the dependencies among the units). Various authors use different terminologies—like behavioral vs. functional vs. structural vs. data modeling views. In summary, a software design is a multifaceted artifact produced by the design process and generally composed of relatively independent and orthogonal views.

Architectural Styles

An architectural style is “a specialization of element and relation types, together with a set of constraints on how they can be used”. An architectural style can thus be seen as providing the software’s high-level organization. Various authors have identified a number of major architectural styles:

- General structures (for example, layers, pipes and filters, blackboard)
- Distributed systems (for example, client-server, three-tiers, broker)
- Interactive systems (for example, Model-View-Controller, Presentation-Abstraction-Control)
- Adaptable systems (for example, microkernel, reflection)
- Others (for example, batch, interpreters, process control, rule-based).

Design Patterns

Succinctly described, a pattern is "a common solution to a common problem in a given context". While architectural styles can be viewed as patterns describing the high-level organization of software, other design patterns can be used to describe details at a lower level. These lower level design patterns include the following:

- Creational patterns (for example, builder, factory, prototype, singleton)
- Structural patterns (for example, adapter, bridge, composite, decorator, façade, flyweight, proxy)
- Behavioral patterns (for example, command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor).

Architecture Design Decisions

Architectural design is a creative process. During the design process, software designers have to make a number of fundamental decisions that profoundly affect the software and the development process. It is useful to think of the architectural design process from a decision-making perspective rather than from an activity perspective. Often, the impact on quality attributes and tradeoffs among competing quality attributes are the basis for design decisions.

Families of Programs and Frameworks

One approach to providing for reuse of software designs and components is to design families of programs, also known as software product lines. This can be done by identifying the commonalities among members of such families and by designing reusable and customizable components to account for the variability among family members. In object-oriented (OO) programming, a key related notion is that of a framework: a partially completed software system that can be extended by appropriately instantiating specific extensions (such as plug-ins).

User Interface Design

User interface design is an essential part of the software design process. User interface design should ensure that interaction between the human and the machine provides for effective operation and control of the machine. For software to achieve its full potential, the user interface should be designed to match the skills, experience, and expectations of its anticipated users.

General User Interface Design Principles

- ***Learnability***. The software should be easy to learn so that the user can rapidly start working with the software.
- ***User familiarity*****.** The interface should use terms and concepts drawn from the experiences of the people who will use the software.
- ***Consistency*****.** The interface should be consistent so that comparable operations are activated in the same way.
- ***Minimal surprise*****.** The behavior of software should not surprise users.
- ***Recoverability***. The interface should provide mechanisms allowing users to recover from errors.
- ***User guidance*****.** The interface should give meaningful feedback when errors occur and provide context-related help to users.
- ***User diversity*****.** The interface should provide appropriate interaction mechanisms for diverse types of users and for users with different capabilities (blind, poor eyesight, deaf, colorblind, etc.).

User Interface Design Issues

User interface design should solve two key issues:

- How should the user interact with the software?
- How should information from the software be presented to the user?

User interface design must integrate user interaction and information presentation. User interface design should consider a compromise between the most appropriate styles of interaction and presentation for the software, the background and experience of the software users, and the available devices.

The Design of User Interaction Modalities

User interaction involves issuing commands and providing associated data to the software. User interaction styles can be classified into the following primary styles:

- ***Question-answer***. The interaction is essentially restricted to a single question-answer exchange between the user and the software. The user issues a question to the software, and the software returns the answer to the question.
- ***Direct manipulation*****.** Users interact with objects on the computer screen. Direct manipulation often includes a pointing device (such as a mouse, trackball, or a finger on touch screens) that manipulates an object and invokes actions that specify what is to be done with that object.
- ***Menu selection*****.** The user selects a command from a menu list of commands.

- ***Form fill-in.*** The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.
- ***Command language.*** The user issues a command and provides related parameters to direct the software what to do.
- ***Natural language.*** The user issues a command in natural language. That is, the natural language is a front end to a command language and is parsed and translated into software commands.

The Design of Information Presentation

Information presentation may be textual or graphical in nature. A good design keeps the information presentation separate from the information itself. The MVC (Model-View-Controller) approach is an effective way to keep information presentation separating from the information being presented. Software Design Software engineers also consider software response time and feedback in the design of information presentation. Response time is generally measured from the point at which a user executes a certain control action until the software responds with a response. An indication of progress is desirable while the software is preparing the response. Feedback can be provided by restating the user's input while processing is being completed. Abstract visualizations can be used when large amounts of information are to be presented. According to the style of information presentation, designers can also use color to enhance the interface. There are several important guidelines:

- Limit the number of colors used.
- Use color change to show the change of software status.
- Use color-coding to support the user's task.
- Use color-coding in a thoughtful and consistent way.
- Use colors to facilitate access for people with color blindness or color deficiency (e.g., use the change of color saturation and color brightness, try to avoid blue and red combinations).
- Don't depend on color alone to convey important information to users with different capabilities (blindness, poor eyesight, colorblindness, etc.).

User Interface Design Process

User interface design is an iterative process; interface prototypes are often used to determine the features, organization, and look of the software user interface. This process includes three core activities:

- ***User analysis.*** In this phase, the designer analyzes the users' tasks, the working environment, other software, and how users interact with other people.

- ***Software prototyping*.** Developing prototype software help users to guide the evolution of the interface.
- ***Interface evaluation*.** Designers can observe users' experiences with the evolving interface.

Localization and Internationalization

User interface design often needs to consider internationalization and localization, which are means of adapting software to the different languages, regional differences, and the technical requirements of a target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without major engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating the text. Localization and internationalization should consider factors such as symbols, numbers, currency, time, and measurement units.

Metaphors and Conceptual Models

User interface designers can use metaphors and conceptual models to set up mappings between the software and some reference system known to the users in the real world, which can help the users to more readily learn and use the interface. For example, the operation "delete file" can be made into a metaphor using the icon of a trash can. When designing a user interface, software engineers should be careful to not use more than one metaphor for each concept. Metaphors also present potential problems with respect to internationalization, since not all metaphors are meaningful or are applied in the same way within all cultures.

Software Design Quality Analysis and Evaluation

This section includes a number of quality analysis and evaluation topics that are specifically related to software design.

Quality Attributes

Various attributes contribute to the quality of a software design, including various "-ilities" (maintainability, portability, testability, usability) and "-nesses" (correctness, robustness). There is an interesting distinction between quality attributes discernible at runtime (for example, performance, security, availability, functionality, usability), those not discernible at runtime (for example, modifiability, portability, reusability, testability), and those related to the architecture's intrinsic qualities (for example, conceptual integrity, correctness, completeness).

Quality Analysis and Evaluation Techniques

Various tools and techniques can help in analyzing and evaluating software design quality.

- Software design reviews: informal and formalized techniques to determine the quality of design artifacts (for example, architecture reviews, design reviews, and inspections; scenario-based techniques; requirements tracing). Software design reviews can also evaluate security. Aids for installation, operation, and usage (for example, manuals and help files) can be reviewed.
- Static analysis: formal or semiformal static (non-executable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking). Design vulnerability analysis (for example, static analysis for security weaknesses) can be performed if security is a concern.

  Formal design analysis uses mathematical models that allow designers to predicate the behavior and validate the performance of the software instead of having to rely entirely on testing. Formal design analysis can be used to detect residual specification and design errors (perhaps caused by imprecision, ambiguity, and sometimes other kinds of mistakes).
- Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototypes).

Measures

Measures can be used to assess or to quantitatively estimate various aspects of a software design; for example, size, structure, or quality. Most measures that have been proposed depend on the approach used for producing the design. These measures are classified in two broad categories:

- Function-based (structured) design measures: measures obtained by analyzing functional decomposition; generally represented using a structure chart (sometimes called a hierarchical diagram) on which various measures can be computed.
- Object-oriented design measures: the design structure is typically represented as a class diagram, on which various measures can be computed. Measures on the properties of the internal content of each class can also be computed.

Software Design Notations

Many notations exist to represent software design artifacts. Some are used to describe the structural organization of a design, others to represent software behavior. Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used for both

purposes. In addition, some notations are used mostly in the context of specific design methods. Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

Structural Descriptions (Static View)

The following notations, mostly but not always graphical, describe and represent the structural aspects of a software design—that is, they are used to describe the major components and how they are interconnected (static view):

- Architecture description languages (ADLs): textual, often formal, languages used to describe software architecture in terms of components and connectors.
- Class and object diagrams: used to represent a set of classes (and objects) and their interrelationships.
- Component diagrams: used to represent a set of components ("physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces") and their interrelationships.
- Class responsibility collaborator cards (CRCs): used to denote the names of components (class), their responsibilities, and their collaborating components' names.
- Deployment diagrams: used to represent a set of (physical) nodes and their interrelationships, and, thus, to model the physical aspects of software.
- Entity-relationship diagrams (ERDs): used to represent conceptual models of data stored in information repositories.
- Interface description languages (IDLs): programming-like languages used to define the interfaces (names and types of exported operations) of software components.
- Structure charts: used to describe the calling structure of programs (which modules call, and are called by, which other modules).

Behavioral Descriptions (Dynamic View)

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software systems and components. Many of these notations are useful mostly, but not exclusively, during detailed design. Moreover, behavioral descriptions can include a rationale for design decision such as how a design will meet security requirements.

- Activity diagrams: used to show control flow from activity to activity. Can be used to represent concurrent activities.
- Communication diagrams: used to show the interactions that occur among a group of objects; emphasis is on the objects, their links, and the messages they exchange on those links.

- Data flow diagrams (DFDs): used to show data flow among elements. A data flow diagram provides "a description based on modeling the flow of information around a network of operational elements, with each element making use of or modifying the information flowing into that element". Data flows (and therefore data flow diagrams) can be used for security analysis, as they offer identification of possible paths for attack and disclosure of confidential information.
- Decision tables and diagrams: used to represent complex combinations of conditions and actions.
- Flowcharts: used to represent the flow of control and the associated actions to be performed.
- Sequence diagrams: used to show the interactions among a group of objects, with emphasis on the time ordering of messages passed between objects.
- State transition and state chart diagrams: used to show the control flow from state to state and how the behavior of a component changes based on its current state in a state machine.
- Formal specification languages: textual languages that use basic notions from mathematics (for example, logic, set, sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and postconditions.
- Pseudo code and program design languages (PDLs): structured programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method.

Software Design Strategies and Methods

There exist various general strategies to help guide the design process. In contrast with general strategies, methods are more specific in that they generally provide a set of notations to be used with the method, a description of the process to be used when following the method, and a set of guidelines for using the method. Such methods are useful as a common framework for teams of software engineers.

General Strategies

Some often-cited examples of general strategies useful in the design process include the divide and-conquer and stepwise refinement strategies, top-down vs. bottom-up strategies, and strategies making use of heuristics, use of patterns and pattern languages, and use of an iterative and incremental approach.

Function-Oriented (Structured) Design

This is one of the classical methods of software design, where decomposition centers on identifying the major software functions and then elaborating and refining them in a hierarchical top-down manner. Structured design is generally used after structured analysis, thus producing (among other things) data flow diagrams and associated process descriptions. Researchers have proposed various strategies (for example,

transformation analysis, transaction analysis) and heuristics (for example, fan-in/fan-out, scope of effect vs. scope of control) to transform a DFD into a software architecture generally represented as a structure chart.

Object-Oriented Design

Numerous software design methods based on objects have been proposed. The field has evolved from the early object-oriented (OO) design of the mid-1980s (noun = object; verb = method; adjective = attribute), where inheritance and polymorphism play a key role, to the field of component-based design, where meta-information can be defined and accessed (through reflection, for example). Although OO design's roots stem from the concept of data abstraction, responsibility-driven design has been proposed as an alternative approach to OO design.

Data Structure-Centered Design

Data structure-centered design starts from the data structures a program manipulates rather than from the function it performs. The software engineer first describes the input and output data structures and then develops the program's control structure based on these data structure diagrams. Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures.

Component-Based Design (CBD)

A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse. Reused and off-the-shelf software components should meet the same security requirements as new software. Trust management is a design concern; components created as having a certain degree of trustworthiness should not depend on less trustworthy components or services.

Other Methods

Other interesting approaches also exist (see the Software Engineering Models and Methods KA). Iterative and adaptive methods implement software increments and reduce emphasis on rigorous software requirement and design. Aspect-oriented design is a method by which software is constructed using aspects to implement the crosscutting concerns and extensions that are identified during the software requirements

process. Service-oriented architecture is a way to build distributed software using web services executed on distributed computers. Software systems are often constructed by using services from different providers because standard protocols (such as HTTP, HTTPS, SOAP) have been designed to support service communication and service information exchange.

Software Design Tools

Software design tools can be used to support the creation of the software design artifacts during the software development process. They can support part or whole of the following activities:

- to translate the requirements model into a design representation;
- to provide support for representing functional components and their interface(s);
- to implement heuristics refinement and partitioning;
- to provide guidelines for quality assessment.