Module 2 Requirements Analysis and Design

FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.

Functional Requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.
- Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

- Ideally, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services and information required by the user should be defined. Consistency means that requirements should not be contradictory.

Non - Functional Requirements

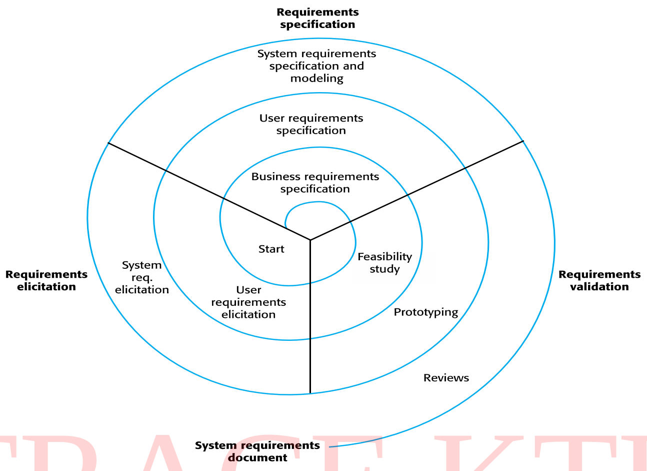
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.
- Non-functional requirements may affect the overall architecture of a system rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
  - It may also generate requirements that restrict existing requirements.

- Product requirements: Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

- Organisational requirements: Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements: Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

REQUIREMENTS ENGINEERING PROCESS

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- In practice, RE is an iterative activity in which these processes are interleaved.


Requirements specification
System requirements specification and modeling
User requirements specification
Business requirements specification
Start
Feasibility study
Requirements elicitation
System req. elicitation
User requirements elicitation
Prototyping
Requirements validation
Reviews
System requirements document

Spiral view of requirements engineering process

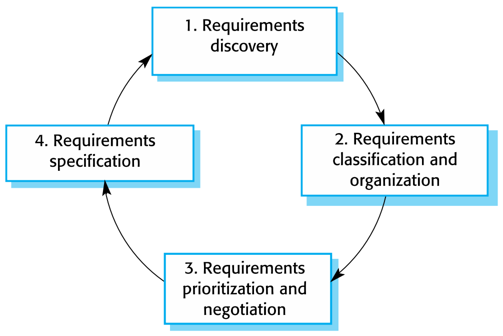
- The output of the RE process is a system requirements document.
- The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

I. Requirements Elicitation

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*
- Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- Stages include:
  - Requirements discovery,
  - Requirements classification and organization,
  - Requirements prioritization and negotiation,
  - Requirements specification.

Problems of requirement elicitation

- Stakeholders don’t know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.


1. Requirements discovery
2. Requirements classification and organization
3. Requirements prioritization and negotiation
4. Requirements specification

Requirements elicitation process

- Requirements discovery: Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- Requirements classification and organisation: Groups related requirements and organises them into coherent clusters.
- Prioritisation and negotiation: Prioritising requirements and resolving requirements conflicts.
- Requirements specification: Requirements are documented and input into the next round of the spiral.

Techniques

1. *Interviewing*

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

a) Closed interviews, where the stakeholder answers a predefined set of questions.
b) Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

2. Ethnography

- Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support these processes.

- The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

II. Requirements Specifications

- The process of writing down the user and system requirements in a requirements document.

- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
    - It is therefore important that these are as complete as possible.

Ways of writing Specifications

Notation	**Description**
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary.

III. Requirements Validation

- Requirements validation is the process of checking that requirements define the system that the customer really wants.
- It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements.
- Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.
- During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.

2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. *Realism checks* By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.

5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

- A number of requirements validation techniques can be used individually or in conjunction with one another:

1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. *Prototyping* This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.

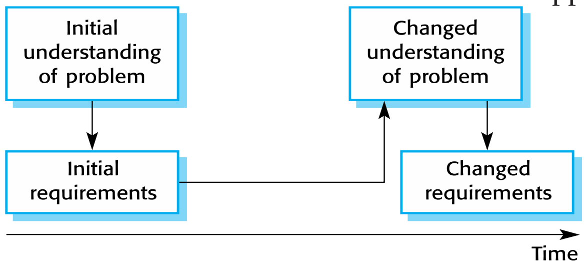
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

IV. Requirements Change

Changing Requirements

- The business and technical environment of the system always changes after installation.
  - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- The people who pay for a system and the users of that system are rarely the same people.
  - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
  - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.


Initial understanding of problem
Changed understanding of problem
Initial requirements
Changed requirements
Time

Requirements Management Planning

- Establishes the level of requirements management detail that is required.
- Requirements management decisions:
    - *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
    - *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
    - *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
    - *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements Change Management

- Deciding if a requirements change should be accepted
    - *Problem analysis and change specification*: During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
    - *Change analysis and costing*: The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
    - Change implementation: The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented

Traceability Matrix:

- *Traceability* is a software engineering term that refers to documented links between software engineering work products (e.g., requirements and test cases).
- A *traceability matrix* allows a requirements engineer to represent the relationship between requirements and other software engineering work products.
- Rows of the traceability matrix are labelled using requirement names and columns can be labeled with the name of a software engineering work product (e.g., a design element or a test case).
- A matrix cell is marked to indicate the presence of a link between the two.
- The traceability matrices can support a variety of engineering development activities.
- They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used.
- Traceability matrices often can be used to ensure the engineering work products have taken all requirements into account.

DEVELOPING USE CASES

- The first step in writing a use case is to defi ne the set of "actors" that will be involved in the story.
- *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.
- Actors represent the roles that people (or devices) play as the system operates.
- Defi ned somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.
- Every actor has one or more goals when using the system.

- *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.
- *Secondary actors* support the system so that primary actors can do their work.

Once actors have been identified, use cases can be developed. Jacobson suggests a number of questions that should be answered by a use case:

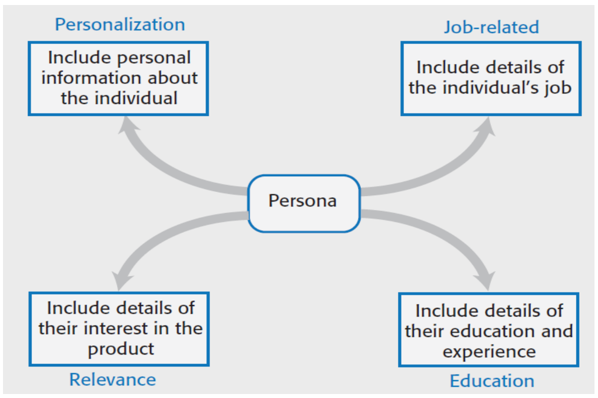
- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

<u>PERSONAS, SCENARIOS, USER STORIES, FEATURE IDENTIFICATION</u>

1. Personas

- Personas are 'imagined users' where you create a character portrait of a type of user that you think might use your product.
- For example, if your product is aimed at managing appointments for dentists, you might create a dentist persona, a receptionist persona and a patient persona.
- Personas of different types of user help you imagine what these users may want to do with your software and how it might be used. They help you envisage difficulties that they might have in understanding and using product features.

- A persona should 'paint a picture' of a type of product user. They should be relatively short and easy-to read.
- You should describe their background and why they might want to use your product.
- You should also say something about their educational background and technical skills.
- These help you assess whether or not a software feature is likely to be useful, understandable and usable by typical product users.


Personalization
Include personal information about the individual
Job-related
Include details of the individual's job
Persona
Include details of their interest in the product
Relevance
Include details of their education and experience
Education

Persona Descriptions

Aspects of Persona Description

- Personalization: You should give them a name and say something about their personal circumstances. This is important because you shouldn't think of a persona as a role but as an individual. It is sometimes helpful to use an appropriate stock photograph to represent the person in the persona. Some studies suggest that this helps project teams use personas more effectively.

- Job-related: If your product is targeted at business, you should say something about their job and (if necessary) what that job involves. For some jobs, such as a teacher where readers are likely to be familiar with the job, this may not be necessary.

- Education: You should describe their educational background and their level of technical skills and experience. This is important, especially for interface design.

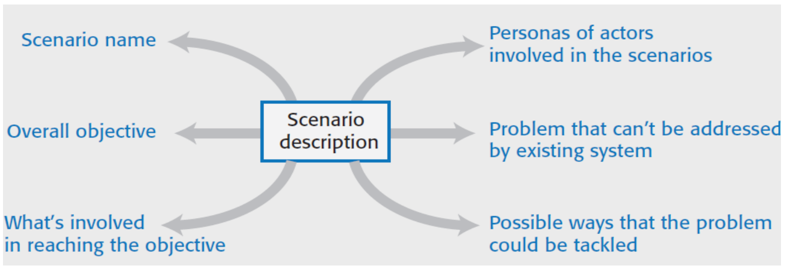
- Relevance: If you can, you should say why they might be interested in using the product and what they might want to do with it.

Persona Benefits

- The main benefit of personas is that they help you and other development team members empathize with potential users of the software.
- Personas help because they are a tool that allows developers to 'step into the user's shoes'.
- Instead of thinking about what you would do in a particular situation, you can imagine how a persona would behave and react.
- Personas can help you check your ideas to make sure that you are not including product features that aren't really needed.
- They help you to avoid making unwarranted assumptions, based on your own knowledge, and designing an over-complicated or irrelevant product.

2. <u>Scenarios</u>

- A scenario is a narrative that describes how a user, or a group of users, might use your system.
- There is no need to include everything in a scenario – the scenario isn't a system specification.
- It is simply a description of a situation where a user is using your product's features to do something that they want to do.
- Scenario descriptions may vary in length from two to three paragraphs up to a page of text.


Scenario name
Personas of actors involved in the scenarios
Overall objective
Scenario description
Problem that can't be addressed by existing system
What's involved in reaching the objective
Possible ways that the problem could be tackled
Elements of scenario descriptions

- A brief statement of the overall objective.
- References to the personas involved so that you can get information about the capabilities and motivation of that user.
- Information about what is involved in doing the activity
- An explanation of problems that can't be readily addressed using the existing system.
- A description of one way that the identified problem might be addressed.

Narrative Scenarios

- Narrative, high-level scenarios, are a means of facilitating communication and stimulating design creativity.
- They are effective in communication because they are understandable and accessible to users and to people responsible for funding and buying the system.
- As with personas, they help developers to agree on a shared understanding of the system that they are creating.

- Scenarios are NOT specifications.
- They lack detail, they may be incomplete, and they may not represent all types of user interactions

Writing Scenarios

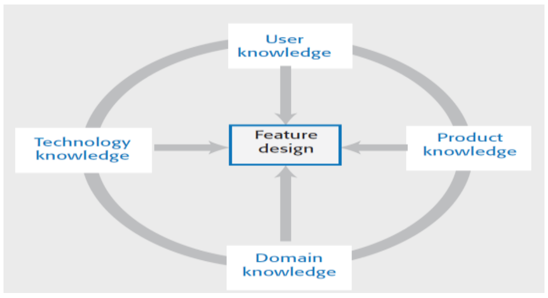
- Scenarios should always be written from the user's perspective and based on identified personas or real users.
- Your starting point for scenario writing should be the personas that you have created. You should normally try to imagine several scenarios from each persona.
- Ideally, scenarios should be general and should not include implementation information.
- However, describing an implementation is often the easiest way to explain how a task is done.
- It is important to ensure that you have coverage of all of the potential user roles when describing a system.

3. <u>User Stories</u>

- User stories are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system.
- An important use of user stories is in planning.
- Many users of the Scrum method represent the product backlog as a set of user stories.
- User stories should focus on a clearly defined system feature or aspect of a feature that can be implemented within a single sprint.
- If the story is about a more complex feature that might take several sprints to implement, then it is called an epic.

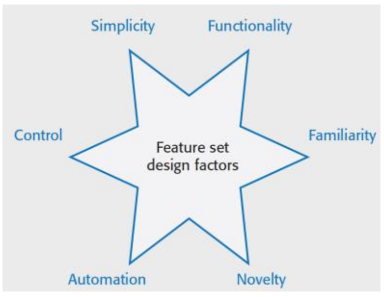
4. <u>Feature Identification</u>

- Your aim in the initial stage of product design should be to create a list of features that define your product.
- A feature is a way of allowing users to access and use your product's functionality so the feature list defines the overall functionality of the system.
- Features should be independent, coherent and relevant:
- Independence
    - Features should not depend on how other system features are implemented and should not be affected by the order of activation of other features.
- Coherence
    - Features should be linked to a single item of functionality. They should not do more than one thing and they should never have side-effects.
- Relevance
    - Features should reflect the way that users normally carry out some task. They should not provide obscure functionality that is hardly ever required.



- User knowledge
    - You can use user scenarios and user stories to inform the team of what users want and how they might use it the software features.

- Product knowledge
    - You may have experience of existing products or decide to research what these products do as part of your development process. Sometimes, your features have to replicate existing features in these products because they provide fundamental functionality that is always required.
- Domain knowledge
    - This is knowledge of the domain or work area(e.g. finance, event booking) that your product aims to support. By understanding the domain, you can think of new innovative ways of helping users do what they want to do.
- Technology knowledge
    - New products often emerge to take advantage of technological developments since their competitors were launched. If you understand the latest technology, you can design features to make use of it.

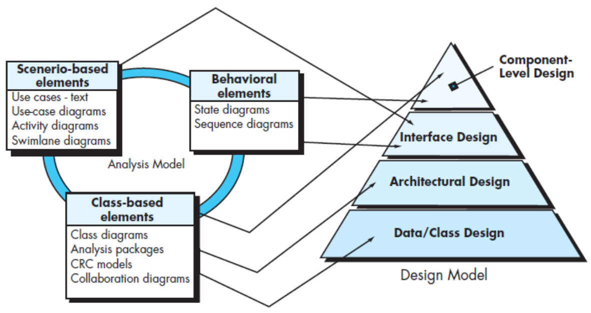


- Simplicity and functionality
    - You need to find a balance between providing a simple, easy-to-use system and including enough functionality to attract users with a variety of needs.
- Familiarity and novelty
    - Users prefer that new software should support the familiar everyday tasks that are part of their work or life. To encourage them to adopt your system, you need to find a balance between familiar features and new features that convince users that your product can do more than its competitors.
- Automation and control
    - Some users like automation, where the software does things for them. Others prefer to have control. You have to think carefully about what can be automated, how it is automated and how users can configure the automation so that the system can be tailored to their preferences.

DESIGN CONCEPTS

Design within concept of Software Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used



- Data/Class Design → transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships provide the basis for the data design activity.

- Architectural Design → defines the relationship between major structural elements of the software, the architectural styles and patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

- Interface Design → describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

- Component-Level Design → transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

Design Process

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.
- Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.
- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.
- These can still be traced to requirements, but the connection is more subtle.

Design Concepts

1. **Abstraction:** When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

- A procedural abstraction refers to a sequence of instructions that have a specific and limited function. A data abstraction is a named collection of data that describes a data object.

2. **Architecture**: *Software architecture* alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

- One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

- *Structural properties* define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." *Extra-functional properties* address " how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

3. Patterns: a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

- The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. **Separation of Concerns:** *Separation of concerns* is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

- A *concern* is a feature or behavior that is specifi ed as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

5. **Modularity:** *Modularity* is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules,* that are integrated to satisfy problem requirements.

6. **Information Hiding:** The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

7. **Functional Independence:** Functional independence is achieved by developing modules with " single - minded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure

8. **Refinement:** Refinement is actually a process of *elaboration.* You begin with a statement of function (or description of information) that is defi ned at a high level of abstraction. That is, the statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

9. **Aspects:** An aspect is a representation of cross cutting.

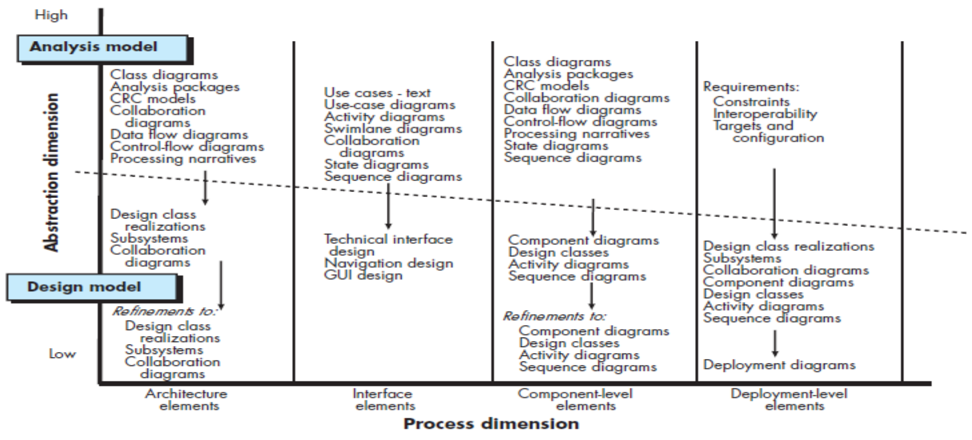
10. **Refactoring:** *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 12.4 . The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refi ned iteratively

Referring to the figure, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

FIGURE 12.4 Dimensions of the design model



Data Design Elements
Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refi ned into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it

Architectural Design Elements
The *architectural design* for software is the equivalent to the floor plan of a house. The architectural model is derived from three sources: (1) information about the application domain for the software to be built; (2) specifi c requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles

Interface Design Elements

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defi ned as part of the architecture. There are three important elements of interface design: (1) the user interface (UI), (2) external interfaces to other systems, devices, networks, or other producers or consumers of information, and (3) internal interfaces between various design components.

Component-Level Design Elements
The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Deployment-Level Design Elements
Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software

<u>SOFTWARE ARCHITECTURE</u>

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

- The architecture is not the operational software. Rather, it is a representation that enables you to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reduce the risks associated with the construction of the software.

three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

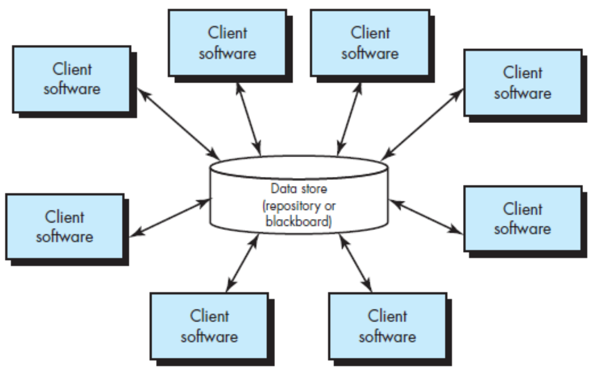
Architectural Styles

- The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system, (2) a set of connectors that enable "communication, coordination and cooperation" among components, (3) constraints that defi ne how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

- An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components

1. Data Centred Architecture:

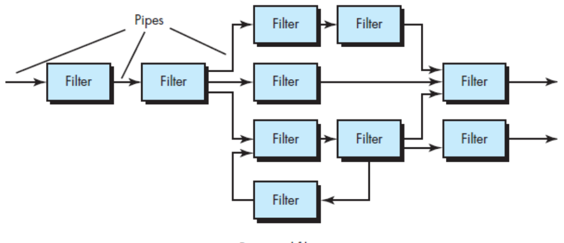
- A data store (e.g., a fi le or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 13.1 illustrates a typical data-centered style.
- Data-centered architectures promote integrability.
- That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently).
- In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients).
- Client components independently execute processes.



2. Data Flow Architecture:

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called filters , connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- However, the filter does not require knowledge of the workings of its neighboring filters.

- If the data flow degenerates into a single line of transforms, it is termed batch sequential .
- This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

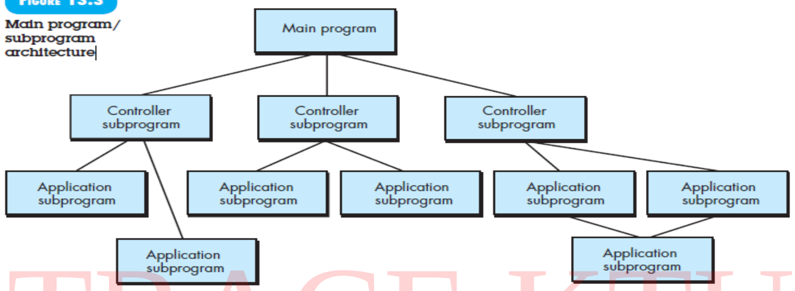


Pipes and filters

3. Call and Return Architecture

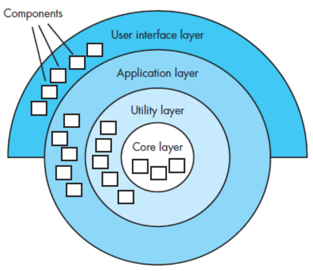
- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:
- Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.
- Remote procedure call architectures. The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

FIGURE 13.3
Main program/ subprogram architecture



4. Object Oriented Architecture: The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

5. Layered Architecture: The basic structure of a layered architecture is illustrated in Figure 13.4 . A number of different layers are defi ned, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



Architectural Consideration

1. Economy —Many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or non functional requirements (e.g., reusability when it serves no purpose). The best software is uncluttered and relies on abstraction to reduce unnecessary detail.

2. Visibility —As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time. Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.

3. Spacing— Separation of concerns in a design without introducing hidden dependencies is a desirable design concept that is sometimes referred to as spacing. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility. Methods like domain-driven design can help to identify what to separate in a design and what to treat as a coherent unit.

4. Symmetry —Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate.

5. Emergence —Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures.

- These considerations do not exist in isolation. They interact with each other and are moderated by each other. For example, spacing can be both reinforced and reduced by economy. Visibility can be balanced by spacing.

COMOPONENT LEVEL DESIGN

A component is a modular building block for computer software. More formally the component is defined as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

- In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.

- In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

- A traditional component, also called a module, resides within the software architecture and serves one of three important roles: (1) a control component that coordinates the invocation of all other problem domain components, (2) a problem domain component that implements a complete or partial function that is required by the customer, or (3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

Designing Class Based Components

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur

1. **The Open-Closed Principle (OCP).** "*A module [component] should be open for extension but closed for modification*". This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

2. **The Liskov Substitution Principle (LSP).** "*Subclasses should be substitutable for their base classes*" This design principle, originally proposed by Barbara Liskov suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the

component instead. LSP demands that any class derived from a base class must honour any implied contract between the base class and the components that use it

3. **Dependency Inversion Principle (DIP).** *"Depend on abstractions. Do not depend on concretions"*

4. **The Interface Segregation Principle (ISP).** *"Many client-specific interfaces are better than one general purpose interface"*

- **The Release Reuse Equivalency Principle (REP).** *"The granule of reuse is the granule of release".* When classes or components are designed for reuse, an implicit contract is established between the developer of the reusable entity and the people who will use it.
- **The Common Closure Principle (CCP).** *" Classes that change together belong together."* Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioural area.
- **The Common Reuse Principle (CRP).** *" Classes that aren't reused together should not be grouped together"* . When one or more classes with a package changes, the release number of the package changes

Design level Guidelines:

1. Components. Naming conventions should be established for components that are specifi ed as part of the architectural model and then refi ned and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model

2. Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC). However, unfettered representation of interfaces tends to complicate component diagrams.

3. Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, components' interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency

Conducting Component level design

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. . Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to defi ne attributes are defi ned within the context of the programming language that is to be used for implementation

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and fi les normally transcend the design description of an individual component.In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented within the context of the computing environment that will house them

Step 7. Refactor every component-level design representation and always consider alternatives. The first component-level model you create will not be as complete, consistent, or accurate as the *n* th iteration you apply to the model. It is essential to refactor as design work is conducted.

Component level design for web apps

- WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability

Content Design at the Component Level

- Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually

Functional Design at the Component Level

- WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure consistency. During architectural design, WebApp content and functionality are combined to create a functional architecture. A *functional architecture* is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.