Lecture 04A: Software Design

After the requirements specification is produced through requirement analysis, it undergoes the software design process.

Software design is the process of defining the architecture, components, interfaces, and other characteristics of a system or component and planning for a software solution.

It is an iterative process through which requirements are analyzed in order to produce a description ("blueprint") of the software's internal structure that will serve as the basis for its construction.

Specifically, a software design should describe:

- The software architecture and the interfaces between those components.
- The components at a level of detail that enable their construction.

Software design is a two step process, although both steps could be performed in parallel so as to achieve a design that will meet the quality requirements and also allow for construction of the system:

- Software architectural design (*top-level design or logical design*): Requirements are partitioned into various components describing software's top-level structure and fundamental organization (*what they do*) and their relationships with each other (*how they interact with each other*) and the environment in order to meet the system's quality goals. This results in an architectural design document which is also an **architectural model** consisting of the specifications of the components which describe what each component must do by specifying the interfaces of the components.
- Software detailed design (*physical*): describes the specific behaviour of each component sufficiently to allow for its construction according to the top-level design. The result of this process is the detailed design document, a design model which describes:
    - the design of the functions of each component, describing how each component provides its required functions,
    - the design of the interface of each component, describing "how" each component provides its services to other components.

Design Principles

1. **The design process should not suffer from "tunnel vision."**
   A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.

2. **The design should be traceable to the analysis model.**
   Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

3. **The design should not reinvent the wheel.**
   Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

4. **The design should "minimize the intellectual distance" [DAV95] between the software and the problem as it exists in the real world.**
   That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.

5. **The design should exhibit uniformity and integration.**
   A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

6. **The design should be structured to accommodate change.**
   Make use of design concepts to enable a design to achieve this principle.

7. **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.**
   Well designed software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

8. **Design is not coding, coding is not design.**
   Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

9. **The design should be assessed for quality as it is being created, not after the fact.**
   Use available design concepts and design measures to assess for quality

10. **The design should be reviewed to minimize conceptual (semantic) errors.**
    There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

Design Concepts

Also referred to as "*Enabling Techniques*", design concepts provide the software designer with guidelines, approaches and key notions to apply so as to obtain good software design. They also provide the basic criteria for design quality.

1. **Abstraction**
   Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose; 'Low level details' are disregarded and an 'element is represented only by its essential characteristics that distinguish it from all other kinds of objects..'. It is an essential principle used to deal with complexity.
   - Data Abstraction
     - This is a named collection of data that describes a data object. E.g. A door is an abstraction of the low level details behind it which may be manufacturer, model number, weight, swing direction.
   - Procedural Abstraction
     - Instructions are given in a named sequence
     - Each instruction has a limited function
   - Control Abstraction
     - A program control mechanism without specifying internal details, e.g., semaphore, rendezvous

2. **Refinement**
   It is a process where one or several instructions of the program are decomposed into more detailed instructions.
   Stepwise refinement is a top-down strategy where a hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions.

   *Abstraction and Refinement are complementary concepts.

3. **Modularity**
   This involves decomposition of a system (Data and Structure) into modules. The objectives of modularity in a design method are:
   - Modular Decomposability
     - To provide a systematic mechanism to decompose a problem into sub problems
   - Modular Composability
     - To promote product flexibility as it enables reuse of existing components to be assembled into a new system.
   - Modular Understandability
     - To promote understanding, since each module can be grasped at a time.
   - Modular Continuity

- If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of the side effects is reduced.

- Modular Protection
  - If there is an error in the module, then those errors are localized and not spread to other modules.

4. Functional Independence - Cohesion and Coupling

Functional Independence is critical in dividing system into independently implementable parts. It is measured (whether the design was well divided into modules) by the two qualitative criteria; coupling and cohesion.

a) Cohesion is the strength of relation between the tasks performed by a module. It is an *intra-module* measure. It measures the degree to which a module performs one and only one function. There are different levels of cohesion:
   a. Coincidental: Occurs when modules are grouped together for no reason at all
   b. Logical: Modules have no actual connection in data but have a shared control flag that will indicate which task will be performed during execution.
   c. Procedural: Modules have procedural cohesion if they perform a series of actions related because they are executed in the specified order.
   d. Temporal: Module functions are bound together because they must be used/executed at approximately the same time.
   e. Communications: Functions are grouped together because they access the same data and are not related in any other way.
   f. Sequential: Tasks are grouped because they belong to the same sequence of operations, they share data at each step, but they don't make up a complete task when done together.
   g. Functional: Tasks are grouped due to the similarity of their functions.

   *In order of most to least desirable on a software design: functional, sequential, communicational, temporal, procedural, logical, and coincidental.

b) Coupling is the degree to which a module is "connected" to other modules in the system. It is the strength of interdependence between software modules. *The more dependent module A is on module B, the stronger the coupling between A and B.*
   Strong coupling means that the involved modules are harder to understand as they have to be understood together, harder to change as the changes affect more than one module, and harder to correct because a problem cited may span over multiple modules other than just one. Levels of coupling include:
   a. No coupling: Modules do not communicate at all with one another
   b. Message Coupling: The loosest type of coupling. Modules do not depend on each other. They use a public interface to exchange parameter-less messages.
   c. Data coupling: Occurs when one module passes local data values to another as parameters.
   d. Stamp coupling: Occurs when part of a data structure is passed to another module as a parameter.
   e. Control Coupling: Occurs when control parameters are passed between modules.

f. Common Coupling: Occurs when multiple modules access common data areas (global).
g. Content Coupling: Occurs when a module accesses data (local) in another module.

* The cohesion of a module may determine how tightly it will be coupled to other modules. The better (i.e., higher) the cohesion, the better (i.e., looser) the coupling.

Asst1: Read more on Cohesion and coupling and make notes on them: description, examples, strengths and weaknesses and any other considerations.

5. Software Architecture

It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. Software architecture is the development work product that gives the highest return on investment with respect to quality, schedule and cost.

The desired properties of an architectural design include:

- Structural Properties
  - This defines the components of a system and the manner in which these interact with one another.
- Extra Functional Properties
  - This addresses how the design architecture achieves requirements for performance, reliability and security
- Families of Related Systems
  - The ability to reuse architectural building blocks.

6. Control Hierarchy

Also called program structure, it represents the organization of program components (modules), and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

For example: The control relationship among modules is expressed in the following way: A module that controls another module is said to be superordinate to it, and conversely, a module controlled by another is said to be subordinate to the controller.

7. Structural Partitioning

The program structure can be divided both horizontally and vertically.

- Horizontal partitions define separate branches of modular hierarchy for each major program function. This is makes the software easier to test and add new features.
- Vertical partitioning: Control and work modules are distributed top down in the program structure. Top level modules perform control functions while lower level modules perform computations.

8. Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture. Data

structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information.

9. **Software Procedure**
   Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

10. **Encapsulation/Information Hiding**
    This concept involves concealing the details of a system entity's implementation from its clients e.g. Modules may be specified and designed so that information contained within the module is inaccessible to other modules that have no need for such information.

Characteristics of a good design

According to Mc Glaughlin:

- A good design must implement all of the explicit requirements contained in the analysis model and accommodate all the implicit requirements desired by the customer.
- A good design must be a readable, understandable guide for code generators, testers and support engineers.
- A good design should provide a complete picture of the software: data, functional and behavioural domains addressed from an implementation perspective.

Software Architecture

A. Software Quality Attributes

Runtime System Qualities: Measured as the system executes:

- **Functionality:** The ability of the system to do the work for which it was intended.
- **Performance:** The response time, utilization, and throughput behaviour of the system. This is not the same as the human performance or system delivery time.
- **Security:** A measure of the system's ability to resist unauthorized attempts at usage or behaviour modification, while still providing service to legitimate users.
- **Availability:** The measure of time that the system is up and running correctly; the length of time between failures and the length of time needed to resume operation after a failure.
- **Usability:** The ease of use and of training the end users of the system. This encompasses learn-ability, efficiency, affect, helpfulness, control qualities.
- **Interoperability:** The ability of two or more systems to cooperate at runtime.

Non-Runtime System Qualities: These cannot be measured as the system executes:

- **Modifiability**: The ease with which a software system can accommodate changes to its software.
- **Portability**: The ability of a system to run under different computing environments. The environment types can be either hardware or software, but is usually a combination of the two.
- **Reusability**: The degree to which existing applications can be reused in new applications.
- **Integrability**: The ability to make the separately developed components of the system work correctly together.
- **Testability**: The ease with which software can be made to demonstrate its faults.

Business Qualities

- **Cost and Schedule:** The cost of the system with respect to time to market, expected project lifetime, and utilization of legacy systems.
- **Marketability:** The use of the system with respect to market competition.
- **Appropriateness for Organization:** Availability of the human input, allocation of expertise, and alignment of team and software structure.

Architecture Qualities

- **Conceptual Integrity:** The integrity of the overall structure that is composed from a number of small architectural structures.
- **Correctness:** Accountability for satisfying all requirements of the system.
- **Completeness**
- **Buildability**

Architectural Views and Viewpoints

According to IEEE, a *view* is a representation of a whole system from the perspective of a related set of concerns. Views are therefore high-level facets of a software design. They represent a partial aspect of a software architecture that shows specific properties of a software system.

A *viewpoint* on the other hand according to IEEE is a specification of the conventions for constructing and using a view. It defines the perspective from which a view is taken. More specifically, it is a pattern or template from which to develop individual views that defines:

- how to construct and use a view (by means of an appropriate schema or template);
- the information that should appear in the view;
- the modelling techniques for expressing and analyzing the information;
- and a rationale for these choices (e.g., by describing the purpose and intended audience of the view).

Architecture can be represented from a variety of viewpoints, all of which can be combined to create a holistic view of the system.
As with the architecture of a building, it is normally necessary to develop multiple views of the architecture of an information system, to enable the architecture to be communicated to, and understood by, the different stakeholders in the system.
To choose the appropriate set of views, there is need to identify the stakeholders who depend on software architecture documentation and the information that they need.

Rationale

Quite often it may not be feasible to capture all that we want to model in a single model or document as this would mean creation of a big, complex, unreadable model. The trick therefore, is to capture a 'subset' of the design decisions by creating several coordinated models and this subset is often organized around a particular concern or other selection criteria. The view – view point concept, the subset-model is called a 'view' and the concern or criteria a 'viewpoint'.

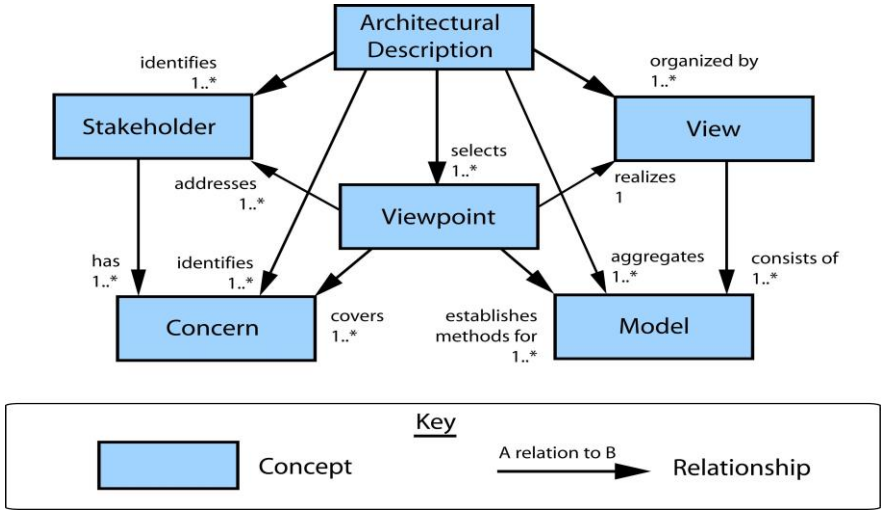

Figure 1: Adapted from "IEEE Recommended Practice for Architectural Description of Software Intensive Systems". IEEE (2000)

Concept Description

Stakeholders: A person, group or organization, with interests in the system e.g. developers.

Concern: A non-functional requirement of the system e.g. delivery, ease of building?

Model: A description of part of the system based on set rules defined by the viewpoint.

View: A representation of the whole system based on a set of related concerns.
E.g. Behavioural View for run-time structure.

Viewpoint: A specification of how to develop views, and the models included. Defines the audience, purpose and techniques for creation and analysis.

E.g. Each architectural view addresses some specific set of concerns, specific to stakeholders in the development process: users, designers, managers, system engineers, maintainers, and so on.

Illustration:

In a Video Library Information system, the view of the user is comprised of all the ways in which she/he interacts with the system, without seeing any details e.g applications or Database Management Systems.

The view of the developer on the other hand is one of productivity and tools, and doesn't include things such as actual live data and connections with consumers.

However, there are things that are shared, such as descriptions of the processes that are enabled by the system and/or communications protocols set up for users to communicate problems directly to development.

In this example, one viewpoint is the description of how the user sees the system, and the other viewpoint is how the developer sees the system. Users describe the system from their perspective, using a model of availability, response time, and access to information while developers describe the system differently, using a model of software connected to hardware distributed over a network, etc.

Several Viewpoint models have been developed based on the framework of the IEEE 1471-2000 (IEEE 2000) standard shown in Figure 1 for example:

- "4+1" View Model, Kruchten, P. (1997).
  This was incorporated into Rational ADS and includes an **iterative design process**.
- SEI View Model, Clements, P. et al. (2002b).
  Developed by the Software Engineering Institute (SEI) of Carnegie Mellon University, it depicts relatively **independent viewpoints** and the focus on documentation for various stakeholders.
- ISO RM-ODP, ISO (1994).
  Developed by the International Standards Organization (ISO), this is a Reference Model of Open Distributed Processing. Its focus is on **Interoperability** of systems and has explicit mention of only **Developers** and **Standards writers** and n**o defined translations** between viewpoints.
- Siemens Four View Model, Soni, D. et al. (1995).
  This model focuses on the **design process** and is explicitly for **Architects** to **design** software architecture.
- Rational ADS, Norris, D. (2004).

Table 1: Summary of Models

	Focus	**Features**	**Viewpoints**
SEI	Communication	Independent viewpoints	3 viewpoints
"4+1"	Design	Iterative design process	5 views
Siemens	Design	Flow of information through viewpoints	4 views
RM-ODP	Re-use	Defines a common vocabulary	5 viewpoints
Rational ADS	Design	Defined mappings between views	4 viewpoints

Case Study: The Phillipe Kruchten '4+1' Model

Basing on the Phillipe Kruchten '4+1' Model, distinct views pertain to distinct issues associated with software design:

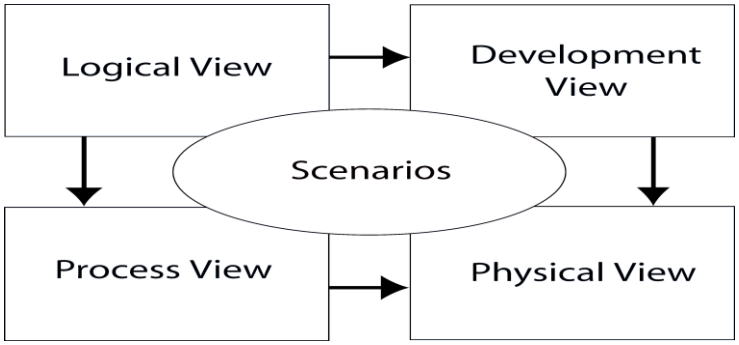

Figure 2: The Krutchen '4+1' View Model

Description of each view

1. Logical view: Capture the logical (often software) entities in a system and how they are interconnected. This is a view that serves to satisfy the functional requirements and system services.
2. Process View: This view deals with concurrency issues mapping functional components into run-time processes.
3. Physical View: This view captures the physical (often hardware) entities in a system and how they are interconnected thus mapping processes & modules to the physical hardware.
4. Developmental View: Capture how logical entities (functional components) are mapped onto physical entities (development modules). It describes how the design is broken down into implementation units.
5. Scenarios: Describe major services to provide to the end users.

Interrelation and Process:

1. Iterative process of design from Scenarios to Physical view.
2. Each iteration adds additional scenarios to validate and expand the architecture.

***In Summary:**

1. A View is what you see. A viewpoint is where you are looking from - the vantage point or perspective that determines what you see. Every view has an associated viewpoint that describes it.
2. A *viewpoint* identifies the set of *concerns* and the *representations*/*modelling techniques*, etc. used to describe the *architecture* to address those *concerns* and a *view* is the result of applying a viewpoint to a particular system [IEEE 1471 Contribution]