

UNIT 4

Software Design

(10 Hours) — Software Engineering (CAC253)

4.1 Design Concepts

Software design is the process of transforming the requirements specification into a blueprint for constructing the software. Design is where the requirements are translated into a “finished” software product: it provides the architecture, data structures, interfaces and components necessary to implement the system. Good design produces software that is correct, understandable, maintainable and efficient.

Abstraction

Abstraction allows designers to focus on the essential details at a higher level and ignore lower-level details. Two kinds are common:

- **Procedural (functional) abstraction** – a named sequence of instructions with a specific, limited function; e.g., *open* a door hides walking, reaching, grasping and turning.
- **Data abstraction** – a named collection of data describing a data object; e.g., *door* has type, dimensions, weight, opening mechanism. Basis for abstract data types and classes.

Architecture

Software architecture is the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. It defines the components (modules), their properties and the relationships between them. Architecture is a key factor in performance, robustness, distributability and maintainability.

Patterns

A **design pattern** describes a proven solution to a recurring design problem in a particular context. It records the problem, the solution, consequences and when to use it. Patterns help designers reuse experience.

Pattern Category	Purpose	Examples
Creational	Object creation	Singleton, Factory, Builder

Pattern Category	Purpose	Examples
Structural	Class and object composition	Adapter, Facade, Decorator
Behavioural	Interaction and responsibility between objects	Observer, Strategy, Command

Modularity

Software is divided into separately named and addressable components called **modules** that are integrated to satisfy problem requirements. Modularity is the single attribute that allows a program to be intellectually manageable. However, if modules are too small, the cost of integration rises; too large, they become complex – so there is an optimum number of modules.

Cohesion and Coupling

Cohesion measures how strongly the elements within a module are related – a cohesive module does one thing well. **Coupling** measures the degree of interdependence between modules. The aim is **high cohesion and low coupling**.

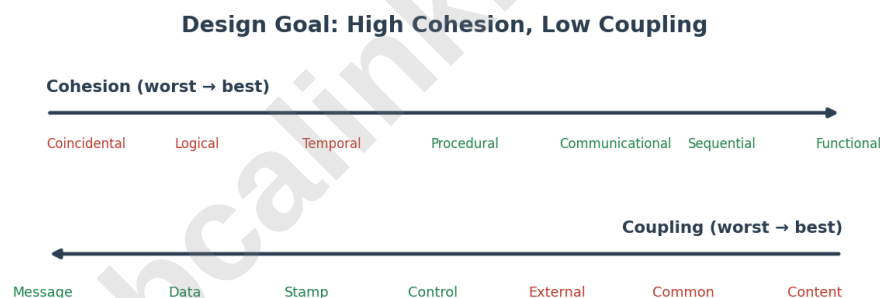


Figure 4.1: Cohesion and coupling scales

Cohesion Type	Description
Coincidental (worst)	Parts are grouped arbitrarily with no meaningful relationship.
Logical	Elements perform similar logical functions selected by a flag (e.g., all input routines).
Temporal	Elements are executed at the same time (e.g., initialisation module).
Procedural	Elements follow a specific execution sequence of control.
Communicational	Elements operate on the same input or produce the same output data.
Sequential	Output of one element is input to the next.
Functional (best)	All elements contribute to a single, well-defined task.

Coupling Type	Description
Content (worst)	One module modifies or relies on the internal workings of another.
Common	Modules share global data.
External	Modules share an externally imposed format, protocol or device.
Control	One module passes control flags that dictate the behaviour of another.
Stamp	Modules share a composite data structure but use only part of it.
Data (good)	Modules communicate only through simple parameters.
Message (best)	Modules communicate only via messages, with no shared data.

Information Hiding

Modules should be specified and designed so that the information (procedure and data) contained within a module is **inaccessible** to other modules that have no need for it. Only a small, well-defined interface is exposed. This limits the effect of changes, reducing errors introduced during maintenance.

Functional Independence

Functional independence is achieved by developing modules with a “single-minded” function and an aversion to excessive interaction with other modules. It is a direct result of modularity, abstraction and information hiding and is measured by cohesion and coupling. Independent modules are easier to develop, test and maintain.

Refinement

Stepwise refinement (Niklaus Wirth) is a top-down design strategy. A program is developed by successively refining levels of procedural detail. It begins with a statement of function stated at a high level of abstraction and elaborates it step by step until programming-language statements can be written.

Stepwise Refinement – Example

Level 0: Process student results.

Level 1: Read marks → compute total and grade → print result sheet.

Level 2 (compute grade): total = sum of subject marks; if total ≥ 40% then pass else fail; assign division.

4.2 Architectural Design

Architectural design is concerned with understanding how a system should be organised and designing the overall structure of that system. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. Architecture affects performance, security, availability and maintainability.

Repository Model

Shared data is held in a central database or repository that may be accessed by all sub-systems. Sub-systems exchange data via the repository rather than directly. Used when large amounts of data are to be shared (e.g., IDEs, CASE toolsets).

Advantages	Disadvantages
Efficient way to share large amounts of data	Sub-systems must agree on a repository data model – an inevitable compromise
Sub-systems need not be concerned with how data is produced	Data evolution is difficult and expensive
Centralised management (backup, security, access control)	Repository is a single point of failure; may be inefficient to organise all communication through it

Client–Server Model

A distributed system model showing how data and processing are distributed across a range of components. It consists of a set of **servers** that offer services, a set of **clients** that call on these services and a **network** that allows clients to access servers.

Layered Model

Organises the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest layers represent core services likely to be used throughout the system. It supports the incremental development of sub-systems in different layers; when a layer interface changes, only the adjacent layer is affected.

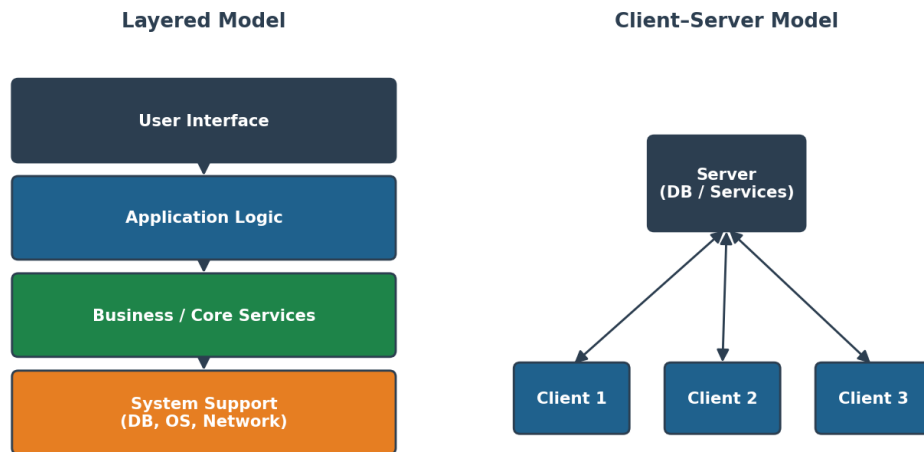


Figure 4.2: Layered model and client-server model

Model	Best Used When	Weakness
Repository	Large volumes of shared data; tools cooperate through data	Single point of failure; data model compromise
Client-Server	Data shared across a network; distributed access needed	Server overload; network dependence; no shared naming service
Layered	Building on existing systems; multiple teams; security across levels	Clean separation is difficult; performance penalty due to multiple layers

Modular Decomposition

After the overall system architecture is chosen, sub-systems are decomposed into modules. Two strategies are used:

- **Object-oriented decomposition** – the system is decomposed into a set of communicating objects.
- **Function-oriented pipelining (data-flow)** – the system is decomposed into functional modules that accept input data and transform it into output data (e.g., the batch-processing pipeline). Each transformation is a separate module that processes data as it flows; suitable when the sequence of processing is clear (payroll, invoice processing).

4.3 Procedural Design Using Structured Methods

Structured design (structured analysis and design) is a function-oriented approach. Its primary tool is the **data flow diagram (DFD)**, which models how data flows through the system and is transformed by processes.

DFD Symbol	Meaning
Circle or rounded rectangle	Process – transforms input data into output data
Open-ended rectangle / parallel lines	Data store – repository of data at rest
Rectangle	External entity – source or destination of data outside the system
Arrow	Data flow – movement of data (labelled)

- **Level 0 (context diagram)** shows the entire system as a single process with external entities.
- **Level 1, 2 ...** DFDs decompose processes into sub-processes (balancing: inputs and outputs must match the parent).
- **Structure charts** are derived from DFDs by transform or transaction analysis. They show module hierarchy, calling relationships and data/control passed between modules.
- **Design notations:** flowcharts, pseudo-code (program design language, PDL), decision tables and structured English.

Pseudo-code (PDL) – Example

PROCEDURE IssueBook(member, book)

IF book.available = TRUE **AND** member.loans < LIMIT **THEN**

 record loan; set due date = today + 14; book.available = FALSE

ELSE display refusal message

ENDIF

END PROCEDURE

4.4 User Interface Design

User interface (UI) design creates an effective communication medium between a human and a computer. Poor interface design is a frequent reason why users reject otherwise good systems. Design should be centred on the needs, abilities and experience of the user.

Human–Computer Interaction (HCI) Principles

Principle	Meaning
User familiarity	Use terms and concepts drawn from the experience of the people who will use the system.
Consistency	Interface should be consistent so comparable operations are activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.

Principle	Meaning
Recoverability	Include mechanisms allowing users to recover from errors (undo, confirmation for destructive actions).
User guidance	Provide meaningful feedback and context-sensitive help.
User diversity	Provide appropriate interaction facilities for different types of users (e.g., large text, keyboard shortcuts).

Interaction Styles

- **Direct manipulation** – interact with objects directly (drag and drop).
- **Menu selection** – select a command from a list.
- **Form fill-in** – fill in fields of a form.
- **Command language** – issue commands with parameters; powerful for experts.
- **Natural language** – speech or typed sentences.

Information Presentation

Information presentation is concerned with presenting system information to users. The software manages the information (model) and a separate presentation component displays it (Model–View–Controller idea), so views can change without affecting the data.

- **Static information** (initialised at the start of a session) versus **dynamic** (changes during a session and must be communicated).
- Use **graphs, charts and tables** for numeric trends; use **digital displays** for precise values.
- Colour: use a small number of colours (no more than about 4–7), do not depend on colour alone (colour blindness), and use colours consistently.
- Use warnings and error messages that are polite, concise, consistent and constructive.

Interface Evaluation

Interface evaluation assesses the usability of an interface and checks that it meets user requirements. Methods include questionnaires, observation of users at work with the system, video snapshots of typical use, and software that collects information about the most-used features and common errors. Prototypes and usability testing with real users are important; heuristic evaluation by experts is also used.

Usability Attribute	Description
Learnability	How long does a new user take to become productive?
Speed of operation	How well does the system response match the user's work practice?
Robustness	Tolerance of user error
Recoverability	Ability to recover from user errors
Adaptability	How closely the system is tied to a single model of work

4.5 Design Notation

Design notations are graphical or textual languages used to record and communicate design decisions. Common notations:

Notation	Use
Data flow diagrams (DFD)	Function-oriented view of data movement and processing
Structure charts	Hierarchy of modules and their interfaces
Flowcharts	Control flow of algorithms
Pseudo-code / PDL	Language-independent procedural design
Decision tables / trees	Complex conditional logic
ER diagrams	Data design – entities, attributes and relationships
UML class, sequence, state diagrams	Object-oriented design – structure and behaviour

Key Takeaways

- ✓ Key design concepts: abstraction, architecture, patterns, modularity, information hiding, functional independence, refinement.
- ✓ **Aim for high cohesion and low coupling**; functional cohesion and data/message coupling are best.
- ✓ Architectural models: **repository** (shared data), **client-server** (distributed services), **layered** (incremental levels of service).
- ✓ Modular decomposition can be object-oriented or function-oriented (data-flow pipeline).
- ✓ Structured design uses DFDs, structure charts and pseudo-code.
- ✓ UI design principles: familiarity, consistency, minimal surprise, recoverability, guidance, diversity.
- ✓ Evaluate interfaces using questionnaires, observation and usability testing.

— *End of Unit 4* —

bcalink.com