SOFTWARE DESIGN & DEVELOPMENT METHODOLOGIES

UNIT 2: SOFTWARE DESIGN, CODING PRACTICE, AND SOFTWARE PLANNING

2.1 Introduction

If Unit 1 explains **what software engineering is** and **how software projects are organized**, Unit 2 begins to answer a new question:

Once requirements are known, how do we shape the solution?
That shaping activity is called **software design**.

Software design stands between requirements and code. Requirements describe what the system should do. Code implements the system in a programming language. Design is the bridge. It turns needs into a structured technical solution.

A weak design often produces software that is difficult to understand, hard to modify, costly to test, and fragile in operation. A good design, by contrast, makes the system easier to build, easier to test, easier to maintain, and easier to extend in future. This is why design is one of the most important activities in software development.

Unit 2 also moves beyond design into **coding discipline** and **software planning**. Students sometimes assume that once a design exists, the rest is straightforward programming. In professional software development, this is not true. Code quality, documentation quality, review discipline, testing discipline, and realistic planning all determine whether the project succeeds.

This chapter therefore studies three tightly connected areas:

1. **Software Design**: how the solution is structured,
2. **Coding Practice**: how the design is translated into maintainable code,
3. **Software Planning**: how effort, size, cost, and schedule are estimated and controlled.

PART I: SOFTWARE DESIGN

2.2 Definition of Software Design

Software design is the activity of transforming requirements into a blueprint for implementation.

A strong definition is:

Software design is the process of defining the architecture, components, interfaces, and data for a software system to satisfy specified requirements.

This definition is important because it shows that design is not just about drawing diagrams. It includes decisions about:

- the major parts of the system,
- how those parts communicate,
- how data is organized,
- how users interact with the system,
- how the system can be built and maintained effectively.

2.2.1 Design as a bridge

A useful way to explain software design is through a construction analogy.

- **Requirements** are like the statement: "Build a hospital with emergency care, wards, operating rooms, and laboratories."
- **Design** is the set of drawings and structural plans showing where everything goes and how everything is connected.
- **Implementation** is the actual construction.

Without design, implementation becomes guesswork.

2.2.2 Why design matters

Design matters because software systems must do more than simply run. They must also be:

- understandable,
- maintainable,
- efficient,
- reusable,
- testable,
- secure,
- scalable where needed.

Many failures in software systems are not caused by syntax mistakes in code. They are caused by poor design decisions made long before coding begins.

2.3 Objectives of Software Design

The objective of software design is not merely to "produce diagrams." The objective is to produce a solution structure that best satisfies requirements while supporting quality.

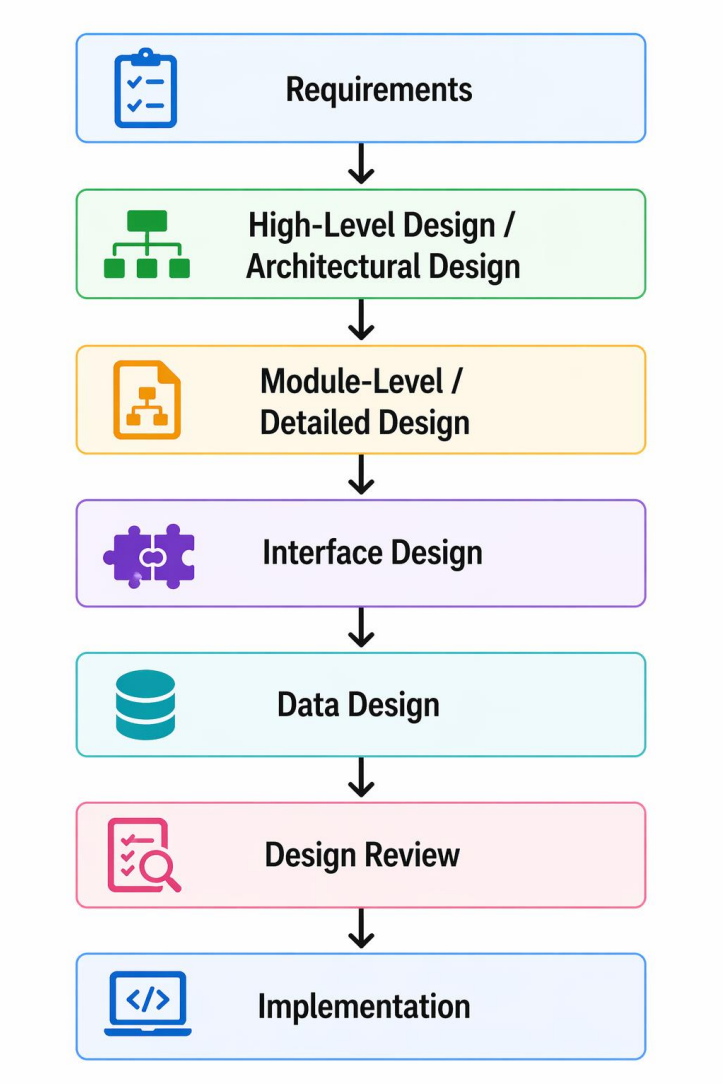
A good design aims to achieve the following.

i. **Correctness:** The design must support all stated requirements. A beautiful design that fails to satisfy user needs is still a bad design.
ii. **Simplicity:** A good design avoids unnecessary complexity. Simplicity improves understanding, reduces defects, and lowers maintenance cost.
iii. **Modularity:** The system should be divided into manageable parts so that different pieces can be developed, tested, and modified independently.
iv. **Maintainability:** Most software spends a large portion of its life being corrected, adapted, or enhanced. Design should make future changes easier.
v. **Reusability:** Where possible, components should be designed so they can be reused in other parts of the system or in future projects.
vi. **Efficiency:** Design must support acceptable use of time, memory, storage, and network resources.
vii. **Reliability:** The design should reduce the chances of failure and help the system behave predictably under normal and abnormal conditions.
viii. **Testability:** The design should make it practical to verify that the system works correctly.
ix. **Security:** The design should protect data and operations against unauthorized access and misuse.
x. **Usability:** Where users interact with the system, design must support clarity, ease of learning, and reduced error.

2.4 Process of Software Design

Software design is a process, not a single act. Designers move from general structure to detailed decisions.

A common view of the process is:



1. **Understanding the problem:** Before design begins, requirements must be studied carefully. If the requirements are unclear, the design will rest on weak foundations.
2. **High-level design:** At this level, designers decide the major subsystems and their interactions.
3. **Detailed design:** At this level, the logic of individual modules, functions, and data structures is refined.
4. **Interface design:** The boundaries between modules and between system and user are defined.
5. **Validation and review:** Design is reviewed before coding to identify defects early.

Important principle

Design is rarely perfect in one attempt. It evolves through review, refinement, and comparison of alternatives.

2.5 Architectural Design

Architectural design defines the overall structure of the system. It answers questions such as:

- What are the main subsystems?
- How do they communicate?
- Where is data stored?
- Which components depend on which others?
- How does the system interact with external systems?

2.5.1 What is software architecture?

Software architecture is the high-level organization of a software system. It identifies major components and the relationships among them.

A software architecture is to a program what the structural plan of a building is to construction.

2.5.2 Why architecture matters

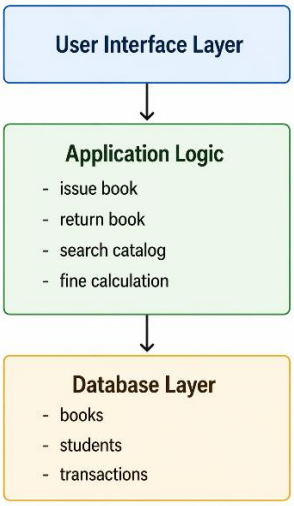
Architecture has long-term effects on:

- maintainability,
- scalability,
- performance,
- reliability,
- deployment choices,
- team organization.

A poorly chosen architecture can make later improvements very difficult.

2.5.3 Simple example: Library management system

A basic architectural decomposition may be:



This is a layered architecture.

2.5.4 Common architectural thinking

Even in diploma-level study, students should understand that architecture is about decomposition at the system level. It may be organized by:

- layers,
- client-server interaction,
- modules,
- services,
- data flow.

2.5.5 Architectural quality questions

A good architecture should help answer:

- Can the system be changed easily?
- Can teams work on parts independently?
- Can failures be isolated?
- Can performance bottlenecks be managed?
- Can security boundaries be enforced?

2.6 Modular Design

Modular design means dividing a software system into smaller parts called **modules**, where each module performs a specific responsibility.

2.6.1 Why modules are needed

No large software system should be built as one giant block of code. That would create confusion and high error rates.

Modules help because they:

- reduce complexity,
- improve readability,
- support division of labor,
- ease testing,
- localize changes,
- promote reuse.

2.6.2 Characteristics of a good module

A good module should:

- perform one well-defined job,
- hide internal details,
- interact through clear interfaces,
- avoid unnecessary dependence on other modules.

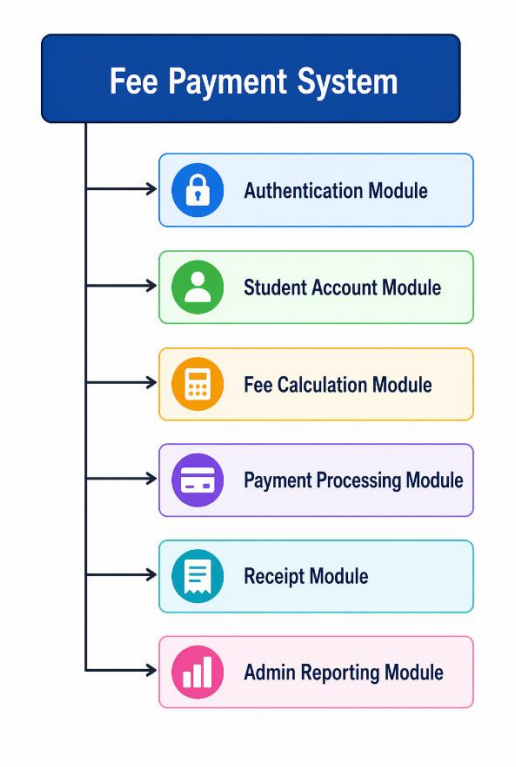
2.6.3 Example: Online fee payment system

Instead of one huge program, the system can be divided into:

- Login module,
- Student profile module,
- Fee calculation module,
- Payment gateway module,
- Receipt generation module,
- Reporting module.

This division makes development and maintenance far easier.

2.6.4 Diagram of modular decomposition



2.6.5 Module independence

The more independent the modules are, the easier it becomes to modify one without damaging others. This idea leads directly to the important concepts of **coupling** and **cohesion**.

2.7 Structure Chart

A structure chart is a design tool that shows the hierarchical organization of modules and their calling relationships.

It is especially useful for explaining top-down functional decomposition.

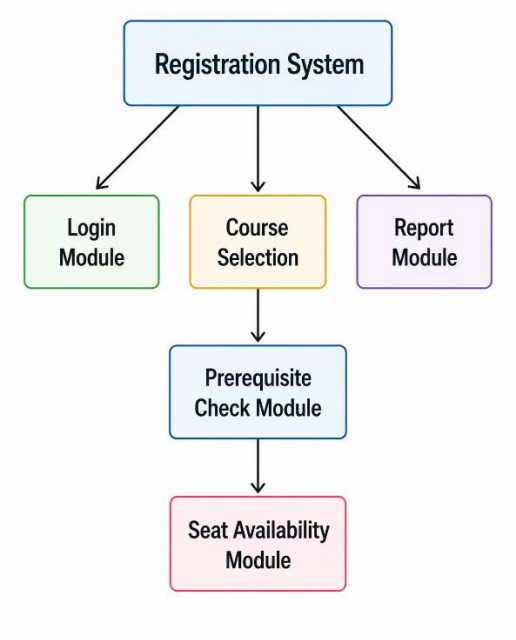
2.7.1 What a structure chart shows

A structure chart shows:

- the main module,
- subordinate modules,
- calling relationships,
- data passed between modules.

2.7.2 Example structure chart

For a student registration system:



2.7.3 Value of structure charts

Structure charts help students and developers see:

- how the system is decomposed,
- what controls what,
- how functionality is broken down.

2.7.4 Limitations

Structure charts are helpful, but they do not capture every aspect of modern systems. They are best understood as a classical design representation rather than a complete modern modelling language.

2.8 Coupling and Cohesion

Coupling and cohesion are among the most important software design concepts in the entire subject.

2.8.1 Cohesion

Cohesion measures how strongly the elements within a module belong together.

A module with high cohesion performs one focused task.

2.8.2 Coupling

Coupling measures the degree of interdependence between modules.

Low coupling is generally preferred because modules become easier to change independently.

2.8.3 The golden design rule

Good design aims for high cohesion and low coupling.

2.9 Types of Cohesion

Cohesion ranges from weak to strong. Different textbooks may order and name them slightly differently, but the classic progression is usually from poor cohesion to strong cohesion.

2.9.1 Coincidental cohesion

This is the weakest form. Unrelated activities are placed in the same module without a meaningful relationship.

Example: One module prints reports, validates login, and calculates salary.

This is poor design.

2.9.2 Logical cohesion

Elements are grouped because they are logically similar, but the module performs different operations depending on control input.

Example: One module handles all input operations, but branches internally for keyboard, mouse, and file input.

Better than coincidental cohesion, but still not ideal.

2.9.3 Temporal cohesion

Elements are grouped because they occur at the same time.

Example: An initialization module that opens files, sets counters, and loads configuration during startup.

This is sometimes acceptable.

2.9.4 Procedural cohesion

Elements are grouped because they follow a sequence of steps.

Example: A module performs steps A, B, and C in order, though the tasks may not be strongly related by data.

2.9.5 Communicational cohesion

Elements are grouped because they use the same data.

Example: A module reads a student record and updates multiple values related to that same record.

2.9.6 Sequential cohesion

The output of one part becomes the input of the next part.

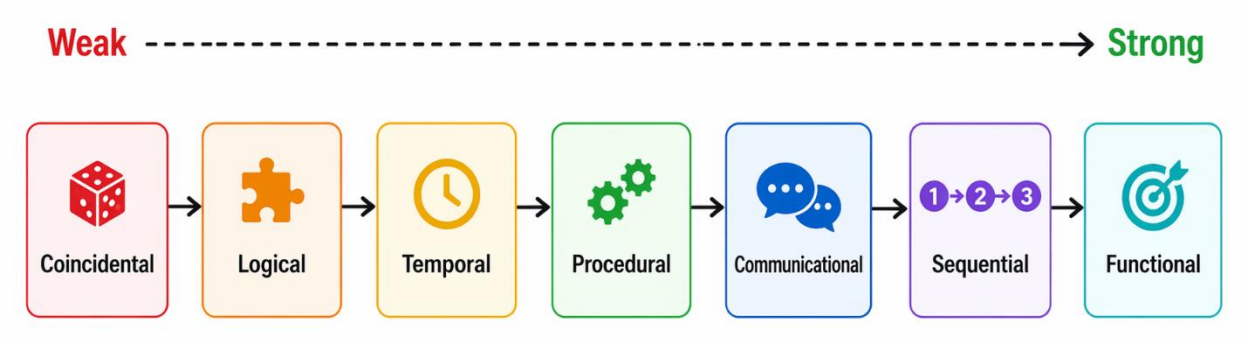
Example: one part validates data, the next formats it, and the next stores it.

2.9.7 Functional cohesion

This is the strongest and most desirable form. Every element in the module contributes to one clearly defined function.

Example: a module whose sole purpose is to calculate semester GPA.

2.9.8 Simple ranking diagram



2.10 Types of Coupling

Coupling also ranges from strong and undesirable to weak and desirable.

1. **Content coupling**

One module directly modifies or depends on the internal workings of another module. This is the worst form.

2. **Common coupling**

Multiple modules share the same global data. This creates hidden dependencies and makes debugging difficult.

3. **External coupling**

Modules depend on externally imposed formats, protocols, or devices. Sometimes unavoidable, but should be controlled.

4. **Control coupling**

One module passes control information that directs the logic of another. Example: passing a flag that tells a module which of many unrelated actions to perform.

5. **Stamp coupling**

A whole data structure is passed even though only part of it is needed. This increases dependency unnecessarily.

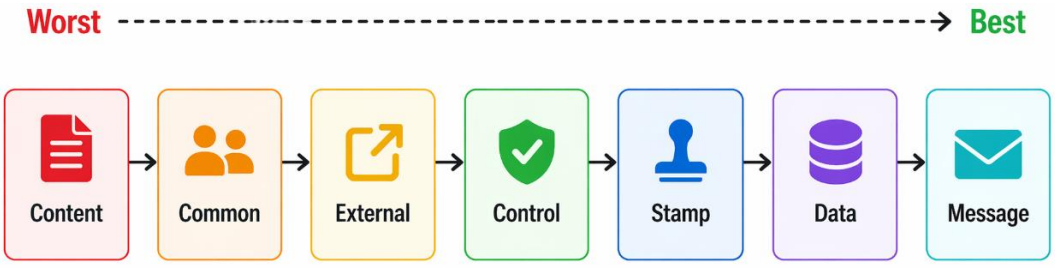
6. **Data coupling**

Modules communicate by passing only the data actually needed. This is much better.

7. **Message coupling**

In object-oriented contexts, modules or objects interact through messages or method calls with minimal knowledge of each other. This is often considered one of the weakest and most desirable forms.

2.10.8 Ranking diagram



2.10.9 Example

Consider two modules:

- CalculateFee(studentData)
- GenerateReceipt(feeAmount)

If the receipt module receives only the exact fee amount needed, coupling is lower than if it receives the entire internal student record structure when most of that data is irrelevant.

2.11 Interface Design

An interface defines how two entities interact. In software, this can mean:

- interaction between modules,
- interaction between software and hardware,
- interaction between software and other software,
- interaction between user and system.

2.11.1 Why interface design matters

A system may have good internal logic but still be difficult to use or integrate if its interfaces are poorly designed.

Good interface design improves:

- communication clarity,
- correctness of interaction,
- maintainability,
- usability,
- interoperability.

2.11.2 Module interface design

For modules, interface design concerns:

- what services a module provides,
- what inputs it expects,
- what outputs it produces,
- what errors it may return,

- what assumptions it makes.

2.11.3 Example

- Module: CalculateFine
- Input: returnDate, dueDate, fineRate
- Output: fineAmount
- Error: invalid date format

This is clearer than allowing other modules to manipulate its internal logic directly.

2.11.4 Good interface principles

A good interface should be:

- simple,
- clear,
- minimal,
- consistent,
- well documented.

2.11.5 Interface errors

Many system defects happen not because modules are internally wrong, but because one module misunderstands another module's interface.

2.12 Design of Human-Computer Interface

The design of Human Computer Interface means the design of the interaction between people and software.

2.12.1 What is HCI in this context?

HCI design focuses on how users see, understand, and operate the system.

Examples include:

- screen layout,
- menus,
- forms,
- buttons,
- labels,

- feedback messages,
- navigation flow.

2.12.2 Why HCI matters

Even a technically correct system can fail if users find it confusing or frustrating.

Poor HCI causes:

- slow task completion,
- frequent user errors,
- low adoption,
- training difficulty,
- dissatisfaction.

2.12.3 Principles of good user interface design

i. **Consistency:** Similar actions should look and behave similarly.
ii. **Clarity:** Instructions, labels, and messages should be easy to understand.
iii. **Simplicity:** The interface should not overload the user.
iv. **Feedback:** The system should tell the user what is happening.
v. **Error prevention:** The design should reduce the chance of mistakes.
vi. **Error recovery:** If mistakes occur, the user should be able to correct them easily.
vii. **Visibility:** Important functions should be easy to find.
viii. **Accessibility:** The design should consider users with different abilities where possible.

2.12.4 Example: student registration screen

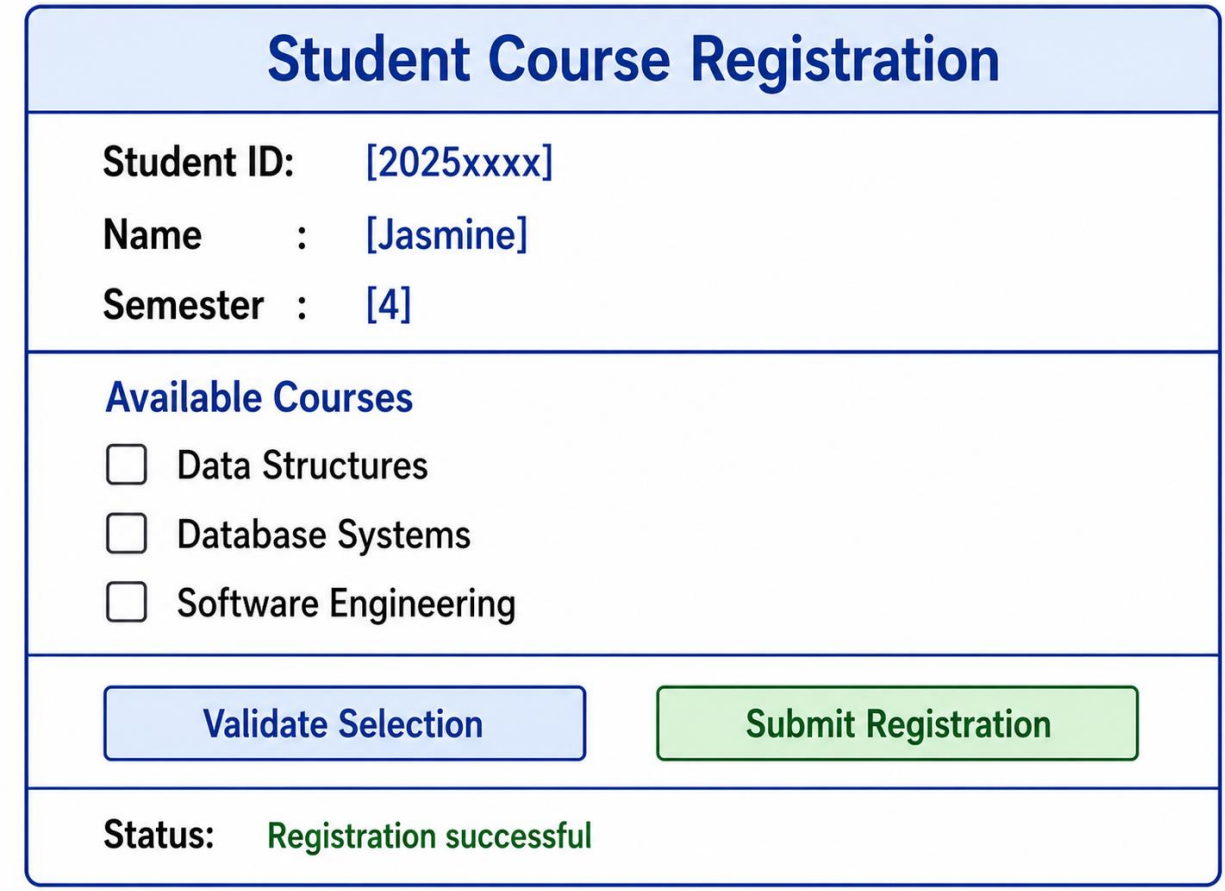
A good registration screen should:

- clearly show available courses,
- show seat availability,
- prevent invalid selections,
- confirm successful registration,
- warn before cancellation.

2.12.5 Bad interface example

If the “Submit” and “Delete” buttons are placed close together with unclear labels, students may accidentally destroy data. This is not just a visual issue. It is a software quality issue.

2.12.6 Simple screen layout concept



PART II: CODING

The Unit 2 syllabus next shifts from design into coding-related discipline: information hiding, programming style, internal documentation, monitoring and control for coding, structured programming, error and exception handling, execution of agile sprints, scrum meetings, code reviews, and unit testing.

2.13 Coding as an Engineering Activity

It is important to understand that coding is not merely "typing instructions." Good coding is disciplined construction based on design.

A professional approach to coding includes:

- following standards,
- using meaningful names,
- documenting logic,
- handling errors safely,
- reviewing work,
- testing modules,
- integrating code carefully.

Coding quality strongly influences maintainability and defect rates.

2.14 Information Hiding

Information hiding is a major design and coding principle. Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.

2.14.1 Meaning

A module should hide internal details and expose only what other parts of the system truly need to know. This reduces dependency and protects the integrity of the module.

2.14.2 Why it matters

If other modules rely on internal implementation details, then changing one module may break many others.

2.14.3 Example

Suppose a BankAccount class stores balance in a certain way. Other classes should not directly manipulate its internal variables carelessly. Instead, they should use controlled methods like:

- deposit(amount)
- withdraw(amount)
- getBalance()

2.14.4 Relation to modularity

Information hiding supports:

- low coupling,
- safer modification,
- better abstraction,
- easier debugging.

Note:

A good module tells the outside world **what it can do**, not **how it does internally**.

2.15 Programming Style

Programming style refers to a set of consistent rules and guidelines used when writing source code to improve human understanding and reduce errors. In other words, it refers to the conventions and habits used to write readable, consistent, and maintainable code.

2.15.1 Why style matters

Programs are read many more times than they are written. Code is read by:

- the original programmer,
- teammates,
- testers,
- maintainers,
- future developers.

Good style reduces confusion and error.

2.15.2 Elements of good programming style

i. **Meaningful names:** Variable and function names should describe their purpose.

   Bad:

   ```
   x = x + y
   ```

   Better:

   ```
   totalFee = totalFee + lateFine
   ```

ii. **Consistent indentation:** Indentation reveals structure.

iii. **Small, focused functions:** Large functions are harder to test and understand.

iv. **Avoiding unnecessary cleverness:** Readable code is better than tricky code.

v. **Clear comments where needed:** Comments should explain intent, not repeat obvious syntax.

vi. **Consistent formatting:** A coding standard should be followed across the team.

2.15.3 Example

Bad style:

```
a=b+c*d/e-f;
```

Better style:

```
netAmount = basicAmount + taxAmount - discountAmount;
```

The second version communicates meaning more clearly.

2.16 Internal Documentation

Internal documentation means explanatory material within the code itself. It consists of information embedded directly within the source code to explain its internal logic and intent to other developers.

It includes:

- comments,
- header notes,
- function descriptions,
- variable naming,
- module summaries.

2.16.1 Why internal documentation matters

Without internal documentation, maintainers may not understand:

- why a certain algorithm was used,
- what assumptions exist,
- what special cases must be handled,
- what each parameter means.

2.16.2 Good documentation practice

A function description may include:

- purpose,
- input,
- output,
- assumptions,
- exceptions,
- author or revision record if organizationally required.

2.16.3 Example

Function: calculateScholarship()

Purpose : Determine scholarship amount based on attendance and marks

Input : attendancePercentage, examScore

Output : scholarshipAmount

2.16.4 Warning

Too many useless comments are not good documentation. Comments should clarify, not create clutter.

2.17 Monitoring and Control for Coding

Monitoring and control in coding means tracking progress, quality, and adherence to standards during implementation.

2.17.1 Why it is needed

If coding proceeds without monitoring, teams may face:

- inconsistent style,
- duplicated work,
- hidden delays,
- poor-quality code,
- integration problems.

2.17.2 What is monitored?

- Module completion status,
- Defect counts,
- Code review outcomes,
- Test results,
- Adherence to coding standards,
- Unresolved issues,
- Sprint commitments in agile settings.

2.17.3 Control actions

When problems are detected, teams may:

- reassign tasks,
- review design decisions,
- refactor code,
- improve test coverage,
- enforce standards more strictly.

Monitoring and control are management activities, but they directly affect coding quality.

2.18 Structured Programming

Structured programming is a disciplined programming approach that emphasizes clear control flow. In other words, Structured programming can be defined as a programming paradigm that organizes code into logical, predictable flows to improve software clarity, quality, and maintainability.

2.18.1 Core idea

Programs should be built using well-understood control structures such as:

- Sequence,
- Selection,
- Iteration.

This avoids chaotic control flow.

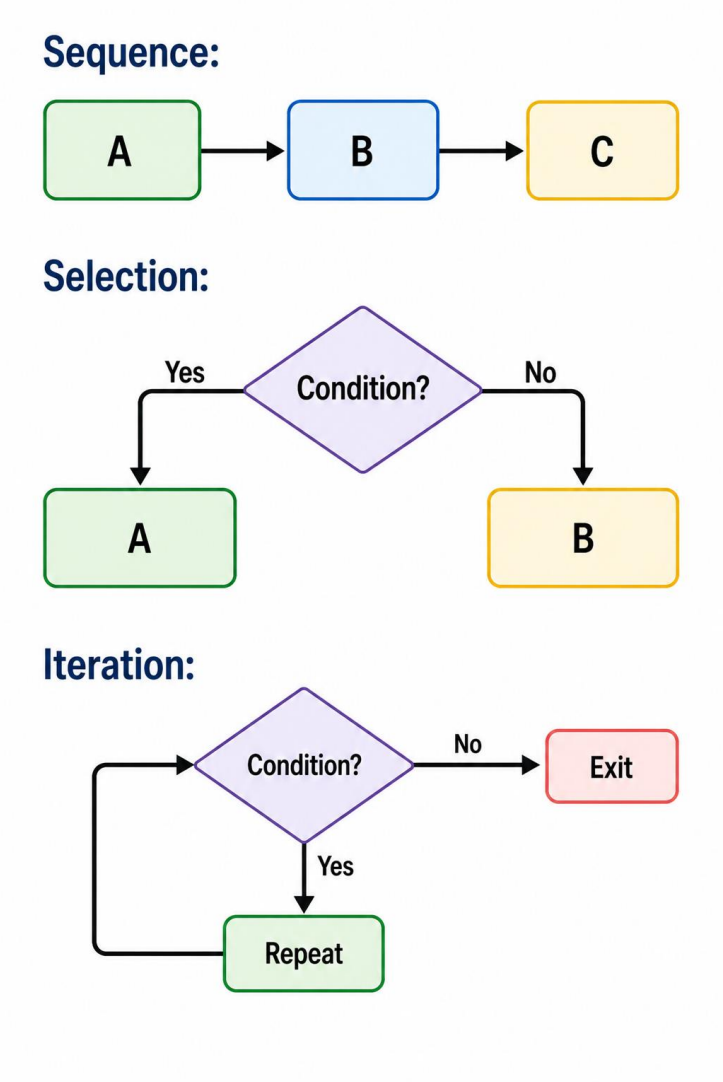
2.18.2 Classical principle

Structured programming rejects unnecessary use of unrestrained jumps in logic because they make programs difficult to follow and debug.

2.18.3 Basic control structures

i. **Sequence:** Instructions execute in order.
ii. **Selection:** A decision chooses one of several paths.
iii. **Iteration:** A block repeats until a condition changes.

2.18.4 Diagram



2.18.5 Benefits of Structured programming

- Easier understanding,
- Easier debugging,
- Easier testing,
- Better maintainability.

2.19 Error and Exception Handling

No real software system can assume that everything will always go right. Therefore, handling errors and exceptions is essential.

2.19.1 Error versus exception

An **error** refers to a fault condition or incorrect situation, while an **exception** in programming often refers to a mechanism used to signal abnormal conditions during execution.

2.19.2 Why error handling matters

Software may face:

- Invalid input,
- Network failure,
- Missing files,
- Divide-by-zero situations,
- Database disconnection,
- Unauthorized access,
- Unavailable resources.

If such conditions are ignored, the system may crash or corrupt data.

2.19.3 Good principles of error handling

- Detect errors early,
- Handle them gracefully,
- Avoid silent failure,
- Give meaningful messages,
- Log important failures,
- Preserve data consistency,
- Never expose sensitive internal information to the wrong user.

2.19.4 Example

Suppose a fee payment system cannot connect to the payment server.

Bad handling:

- Application closes suddenly.

Better handling:

- User sees a message such as “Payment service is temporarily unavailable. Please try again later.”
- Transaction is not falsely marked as completed,
- The event is logged for administrators.

2.19.5 NOTE:

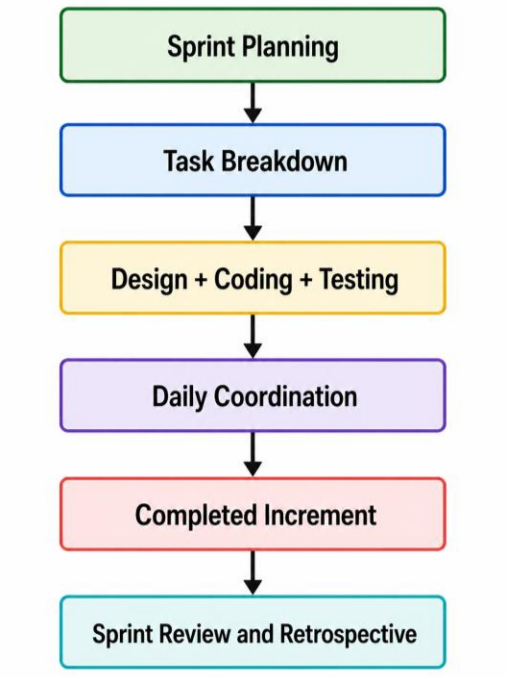
Error handling improves reliability, user trust, and data safety.

2.20 Executing Sprints in Agile

2.20.1 What is a sprint?

A sprint is a short, fixed time period in which a team completes selected backlog items (such as user story, bugs and defects, and technical debt) and produces a usable increment.

2.20.2 Sprint flow



2.20.3 Key idea

A sprint is not “coding only.” It includes all work needed to turn backlog items into a working increment, including design, coding, testing, and review.

2.20.4 Importance of sprint discipline

To execute a sprint well, the team must:

- choose realistic work,
- keep goals clear,
- communicate regularly,
- address blockers quickly,
- ensure completed work is truly tested.

2.20.5 Example

For a library system, a sprint goal might be:

“Implement book search and availability display.”

During that sprint, the team designs the search screen, codes the backend logic, tests results, and demonstrates the feature.

2.21 Scrum Meetings in Agile

Scrum meetings (or events/ceremonies) are structured, time-boxed, and recurring collaborative sessions in Agile software development designed to align the team, track progress, and foster continuous improvement. These meetings ensure transparency, inspection, and adaptation during a sprint (short development cycle). Scrum meetings help synchronize the team during Agile work.

2.21.1 Daily scrum

The daily scrum is a short coordination meeting. Team members commonly discuss:

- what was done,
- what will be done next,
- what obstacles exist.

2.21.2 Sprint planning meeting

The team selects backlog items for the sprint and defines the sprint goal. It defines what can be delivered in the sprint and how that work will be achieved.

2.21.3 Sprint review

It is held at the end of the sprint to demonstrate completed work (completed increment) to stakeholders and gather feedback.

2.21.4 Retrospective

A session for the team to reflect on the process and identify improvements for the next sprint. The team reflects on what went well, what went poorly, and how to improve the next sprint.

2.21.5 Why meetings matter

These meetings improve visibility, accountability, issue detection, and adaptation.

2.22 Importance of Code Reviews

A code review is a systematic examination of code by someone other than the original author.

2.22.1 Why code reviews are valuable

They help find:

- Logic errors,
- Style violations,
- Security weaknesses,
- Design inconsistency,
- Missing test cases,
- Maintainability problems.

2.22.2 Educational value

Reviews also improve team learning. Developers learn from each other's approaches.

2.22.3 Good review questions

- Is the code correct?
- Is it readable?

- Does it follow design decisions?
- Are edge cases handled?
- Is the naming clear?
- Are tests adequate?

2.22.4 Important note

A code review is not a personal attack. It is a quality assurance practice.

2.23 Unit Testing

Unit testing means testing the smallest logical pieces of software, usually individual functions, methods, or modules.

2.23.1 Purpose

Unit testing checks whether a small component behaves as expected in isolation.

2.23.2 Why unit testing is important

- Defects are found early,
- Debugging becomes easier,
- Refactoring becomes safer,
- Confidence in code increases.

2.23.3 Example

A unit test for calculateFine(daysLate) may check:

- 0 days late returns 0,
- 1 day late returns correct fine,
- negative input is rejected or handled,
- very large delay is processed correctly.

2.23.4 Position in quality process

Unit testing is not the only testing needed, but it is a very important starting point.

PART III: SOFTWARE PLANNING

The Unit 2 syllabus also includes software metrics, product and process metrics, Agile-relevant metrics, function point and feature point metrics, project estimation, steps in estimation, reasons for poor estimation, guidelines, estimation models, COCOMO, automated estimation tools, and sprint planning.

2.24 Introduction to Software Planning

Software planning is the activity of deciding:

- how much work is involved,
- how long it may take,
- how many people are needed,
- what resources are required,
- what risks may affect the plan.

A project without planning is difficult to control. But planning is also difficult because software is intangible and complex. Unlike a physical object, software size and effort are not always obvious at the beginning.

2.25 Software Metrics

A software metric is a quantitative measure related to software product, process, or project.

2.25.1 Why metrics matter

Without measurement, management becomes guesswork. Metrics help teams understand:

- Size,
- Quality,
- Productivity,
- Effort,
- Defect trends,
- Schedule progress.

2.25.2 Definition

A metric is a measurable attribute used to assess some aspect of software or the software process.

2.25.3 Two broad categories of Metrics

There are two types of metrics to be discussed in our context:

- Product metrics
- Process metrics

2.26 Product Metrics

Product metrics describe characteristics of the software product itself.

Examples include:

- Size of software,
- Complexity,
- Performance,
- Number of defects,
- Reliability indicators,
- Usability-related measures.

2.26.1 Example

For a student portal, product metrics may include:

- Response time for login,
- Number of defects per release,
- Number of modules,
- Memory usage,
- Average page load time.

2.27 Process Metrics

Process metrics describe the effectiveness or quality of the development process.

Examples include:

- Defect detection rate,
- Rework percentage,
- Review coverage,
- Test execution rate,

- Effort spent per phase,
- Sprint completion rate.

2.27.1 Example

If a team consistently finds most defects only during final testing, that suggests the earlier phases may be weak. Process metrics help reveal such problems.

2.28 Relevant Metrics in Agile

Agile projects still use measurement, but often focus on practical team-level indicators.

Examples may include:

- sprint burndown,
- completed story count,
- defect leakage,
- test pass rate,
- cycle time,
- team velocity.

NOTE: Agile does not mean “no metrics.” It means metrics should support learning and delivery rather than becoming rigid bureaucracy (system).

2.29 Function Point Metric

Function Point Analysis estimates software size by measuring the functionality provided to the user.

2.29.1 Why function points are used

Lines of code depend heavily on programming language and style. Function points try to measure software size in a more language-independent way.

2.29.2 Basic idea

Function point analysis looks at user-visible functions such as:

- Inputs,
- Outputs,
- Inquiries,

- Internal files,
- External interfaces.

Each is weighted according to complexity, and the total gives a measure of system size.

2.29.3 Why it is useful

Function points are useful in early estimation because they can be considered before coding begins.

2.29.4 Example intuition

In a college administration system, the following may contribute to function size:

- input student registration form,
- output fee report,
- query attendance record,
- maintain student database file,
- connect to payment gateway.

Each contributes to functional size.

2.30 Feature Point Metric

Feature points are related to function points but are intended to better represent certain systems, especially where algorithmic complexity is higher. It is used to estimate the size, complexity, and development effort of systems with high algorithmic complexity.

2.30.1 Main idea

Feature points extend the idea of function points by recognizing that some systems are not well represented by transaction counts alone.

2.30.2 Usefulness

They are more relevant in systems where processing complexity matters strongly, not only data movement.

2.31 Software Project Estimation

Estimation is the activity of predicting the size, effort, duration, and cost of a project.

2.31.1 Why estimation is difficult

Software projects deal with uncertainty:

- Incomplete requirements,
- Human productivity differences,
- Technology learning curves,
- Integration surprises,
- Changing scope.

2.31.2 What is estimated?

- Size,
- Effort,
- Schedule,
- Staffing,
- Cost,
- Risk exposure.

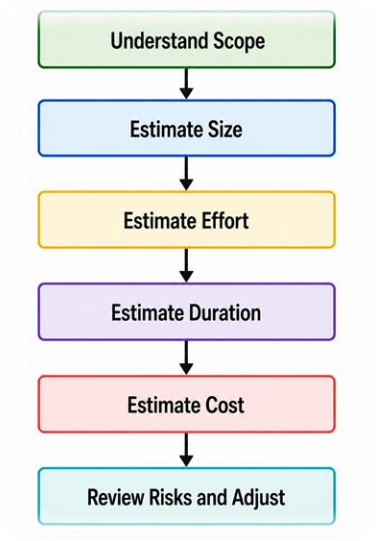
2.31.3 Why estimation matters

Poor estimation affects:

- Deadlines,
- Budgeting,
- Customer trust,
- Staff pressure,
- Quality.

2.32 Steps for Estimation

A practical estimation sequence is:



i. **Understand scope:** Clearly define what the system will and will not include. If requirements are unclear or incomplete, any estimates made later are likely to be inaccurate and may lead to delays or cost overruns.

ii. **Estimate size:** Determine how large the project is in terms of functionality or complexity. Techniques such as function points, feature points, breaking the system into smaller parts (decomposition), or using data from past projects help make this estimation more reliable.

iii. **Estimate effort:** Calculate the amount of work required to complete the project. This is typically expressed in person-hours, person-days, or person-months and reflects how much human effort is needed.

iv. **Estimate duration:** Determine how long the project will take. Duration depends on effort but also on team size, dependencies, and coordination. Increasing team size does not always reduce time proportionally due to communication and management overhead.

v. **Estimate cost:** Convert effort and resource needs into monetary terms. This includes salaries, tools, infrastructure, and other operational expenses required to complete the project.

vi. **Review risk:** Identify uncertainties and potential problems (such as new technology or unclear requirements). High-risk areas should be carefully considered and may require adjustments to effort, time, and cost estimates to avoid unexpected issues.

2.33 Reasons for Poor and Inaccurate Estimation

Poor estimation may result from:

- Unclear requirements,
- Optimistic assumptions,
- Ignoring historical data,
- Underestimating complexity,
- Unplanned changes,
- Weak technical knowledge,
- Political pressure to promise too early,
- Ignoring risk,
- Assuming best-case productivity.

2.33.1 Common misunderstanding

People sometimes think inaccurate estimation is caused only by "wrong calculation." In reality, wrong estimation is often caused earlier by wrong assumptions and incomplete understanding.

2.34 Project Estimation Guidelines

Good estimation practice includes the following.

i. **Break work into smaller parts:** Smaller pieces are easier to estimate than one huge block.
ii. **Use historical data:** Past project information improves realism.
iii. **Involve technical staff:** People who will do the work often understand complexity better than outsiders.
iv. **Include risk allowance:** Not everything goes according to the best-case scenario.
v. **Re-estimate when requirements change:** An estimate is not permanently valid if scope changes.
vi. **Use multiple methods where possible:** Comparing expert judgment, decomposition, and model-based estimation may produce a better result than relying on one approach alone.

2.35 Models for Estimation

Estimation models provide systematic ways to predict effort and cost.

Common approaches include:

- Expert judgment,
- Analogy-based estimation,
- Decomposition methods,
- Algorithmic models such as COCOMO.

Each has strengths and limitations.

2.36 COCOMO Model

COCOMO stands for **Constructive Cost Model**. Constructive Cost Model (COCOMO) is a software cost estimation model that predicts the effort, time, and cost required for a Software Development project. It uses factors like project size, complexity, and team experience to provide accurate estimates, helping with planning, budgeting, and resource allocation. It is a well-known software cost estimation model.

2.36.1 Basic idea

COCOMO uses project size and other factors to estimate development effort and schedule.

2.36.2 Why COCOMO matters academically

It is one of the classic models in software engineering because it provides a more structured alternative to guesswork.

2.36.3 Core concept

Effort tends to increase with project size, but not always in a simple linear way. Project type and development conditions also matter.

2.36.4 Important note:

COCOMO says that software effort can be estimated mathematically using known project characteristics rather than intuition alone.

2.36.5 Caution

COCOMO is an **estimation model**, not as a guarantee of perfect prediction.

2.37 Automated Tools for Estimation

2.37.1 Purpose of tools

Tools help teams:

- calculate estimates faster,
- organize project data,
- compare scenarios,
- track progress against plan,
- reuse historical information.

2.37.2 Limitation

A tool is only as good as the input assumptions. Poor input produces poor estimates.

2.37.3 Important note:

Estimation tools support judgment; they do not replace professional understanding.

2.38 Sprint Planning in Agile

Sprint planning is a collaborative meeting in Agile methodologies where teams decide which tasks to complete in an upcoming sprint and how that work will be achieved. This meeting is typically facilitated by the product owner or Scrum master. Sprint planning connects software planning with Agile execution.

2.38.1 What happens in sprint planning?

The team decides:

- what backlog items to take,
- what sprint goal to pursue,
- how the work will be approached,
- whether the chosen work is realistically achievable.

2.38.2 Why sprint planning matters

Without careful sprint planning:

- Teams overcommit,

- Work remains incomplete,
- Quality drops,
- Trust decreases.

2.38.3 Example

Suppose a sprint lasts two weeks. The team selects:

- login enhancement,
- attendance dashboard,
- report export feature.

If the work is far greater than team capacity, the sprint will fail. Good sprint planning matches ambition with available time and skill.

2.39 Integrated Example: Unit 2 as One Continuous Story

To help you see the whole unit as one connected body of knowledge, consider this example.

Problem

A college wants to build an online internal portal for course registration, attendance display, and fee status.

Design stage

The team decides on a layered architecture:

- user interface,
- application services,
- database layer.

Then it performs modular design:

- login module,
- registration module,
- attendance module,
- fee module,
- reporting module.

A structure chart is drawn to show module hierarchy.

The team checks for good design qualities:

- high cohesion in each module,
- low coupling between modules.

The interface design ensures clear module inputs and outputs.

The human-computer interface is designed so students can use the portal easily.

Coding stage

Developers apply information hiding. They use proper naming and internal documentation. They handle errors such as invalid login or unavailable database connections. They perform code reviews and unit testing.

Planning stage

The team estimates size and effort. Metrics are tracked. Sprint planning is used for incremental delivery. Actual progress is compared with estimates.

This example shows that Unit 2 is not a set of disconnected headings. It is the full path from solution structure to disciplined implementation and realistic planning.

2.40 Short Revision Summary

Software design transforms requirements into a technical solution structure. Major design activities include architectural design, modular decomposition, interface definition, and human-computer interface planning. Good design should be correct, simple, modular, maintainable, testable, and efficient.

Structure charts help visualize module hierarchy. Modular design is strengthened by aiming for **high cohesion** within modules and **low coupling** between them. Interface design ensures that modules and users interact clearly and safely.

Coding is an engineering discipline, not just programming. Important coding concerns include information hiding, programming style, internal documentation, structured programming, error handling, code reviews, and unit testing.

Software planning uses metrics and estimation to predict and control project effort, time, and cost. Product metrics describe the software itself, while process metrics describe how development is

performed. Function points and feature points help estimate system size. COCOMO is a classic model for effort estimation. In Agile work, sprint planning helps convert backlog items into realistic short-term commitments.