

UNIT 2

Software Development Process Models

(8 Hours) — Software Engineering (CACS253)

2.1 Software Process

A **software process** is a structured set of activities required to develop a software system. Every process involves four fundamental activities:

- **Specification** – defining what the system should do (functions and constraints).
- **Design and implementation** – organising the system and converting the design into an executable program.
- **Validation** – checking that the system does what the customer wants.
- **Evolution** – changing the system in response to changing customer needs.

A **software process model** is an abstract representation of a process – a simplified description from a particular perspective. Models are generic frameworks that can be extended and adapted to create more specific processes. Process models can be **plan-driven** (all activities planned in advance) or **agile** (planning is incremental and continuous).

2.2 The Waterfall Model

The waterfall model, also called the **linear sequential life cycle**, is the classical plan-driven model. The phases are separate and sequential: the output of one phase is the input to the next, and each phase must be complete before the next begins.

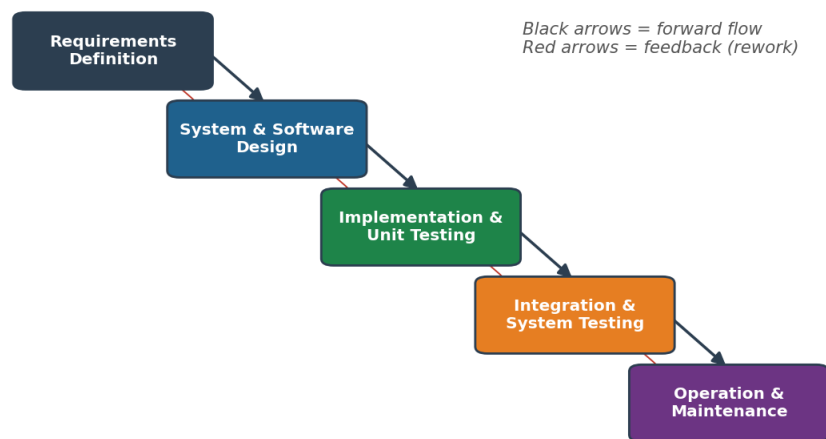


Figure 2.1: The waterfall model with feedback between phases

Phase	Activity and Output
Requirements definition	Services, constraints and goals are established with users. Output: requirements document (SRS).
System and software design	Overall architecture and fundamental abstractions. Output: design documents.
Implementation and unit testing	Design realised as programs; each unit tested against its specification.
Integration and system testing	Units integrated and tested as a complete system, then delivered.
Operation and maintenance	Longest phase: system installed, errors corrected, enhancements added.

Advantages and Problems of Waterfall

Advantages	Disadvantages
Simple, easy to understand and manage	Inflexible partitioning into distinct stages makes it difficult to respond to changing customer requirements
Well-documented; each phase has deliverables	A working version appears late – customers see nothing until near the end
Clear milestones suit large projects with fixed requirements	Cost of late-discovered errors is high
Works well when requirements are well understood	Not suitable when requirements are uncertain or change often

Note: The waterfall model should be used only when requirements are well understood and unlikely to change significantly – for example, in large systems engineering projects developed at several sites, or safety-critical systems.

2.3 Evolutionary Development

Evolutionary development interleaves specification, development and validation. An initial version is quickly developed from abstract specifications, exposed to user comments and then refined through many versions until an adequate system emerges.

- **Exploratory development** – work with customers to explore requirements and deliver a final system. Starts with the best-understood requirements and adds features proposed by the customer.
- **Throw-away prototyping** – the objective is to understand the customer's requirements and develop a better requirements definition. The prototype concentrates on poorly understood requirements and is discarded afterward.

Strengths	Problems
Specification developed incrementally; user needs discovered as work proceeds	The process is not visible – managers need regular deliverables to measure progress
Effective for small and medium interactive systems	Systems are often poorly structured because continual change corrupts the structure
Quick user feedback reduces the risk of building the wrong system	Special skills (e.g., in rapid-prototyping languages) may be required

2.4 Component-Based Software Engineering (CBSE)

CBSE is a reuse-based approach in which systems are assembled from existing, independently designed software components. Reuse is the dominant approach for building many types of business systems.

- **Component analysis** – search for components matching the requirements; usually only partial match is found.
- **Requirements modification** – requirements are analysed using discovered component information and modified to reflect available components.
- **System design with reuse** – the framework is designed or an existing framework reused, considering the components.
- **Development and integration** – software that cannot be bought is developed and integrated with components (“glue code”).

Advantages	Disadvantages
Reduced development cost and effort	Requirements compromises are inevitable; the system may not meet real user needs

Advantages	Disadvantages
Faster delivery	Loss of control over the evolution of reused components
Reduced risk because components are already tested	Integration (mismatch) problems between components

2.5 Process Iteration

System requirements always evolve; therefore, iteration is part of larger systems. Two approaches explicitly support iteration:

A. Incremental Delivery

Instead of delivering the system as a single delivery, the development and delivery is broken down into increments, with each increment delivering part of the required functionality. User requirements are prioritised and the highest-priority requirements are included in early increments. Once development of an increment starts, requirements are frozen, though requirements for later increments can continue to evolve.

Advantages	Disadvantages
Customer value delivered with each increment, so functionality is available early	Difficult to identify common facilities needed by all increments
Early increments act as prototypes that help elicit requirements for later increments	Iterative processes can be hard when replacing an old system whose users need full functionality
Lower risk of overall project failure; highest-priority services tested most	The specification is developed with the software, which conflicts with procurement models requiring full spec

B. Spiral Development (Boehm)

The process is represented as a **spiral** rather than a sequence of activities. Each loop in the spiral represents a phase of the process – feasibility, requirements definition, system design and so on. There are no fixed phases; loops are chosen as needed. Risks are explicitly assessed and resolved throughout the process.

Boehm's Spiral Model (cost & progress grow outward)



Figure 2.2: The spiral model – four sectors per loop

Sector	Activity
Objective setting	Specific objectives for this phase are identified; constraints and alternatives are listed.
Risk assessment and reduction	Risks are assessed; activities (e.g., a prototype) are put in place to reduce key risks.
Development and validation	A development model for the system is chosen and applied.
Planning	The project is reviewed and the next loop of the spiral is planned.

Note: Spiral model is risk-driven. Its main drawback is that it requires risk-assessment expertise and can be costly for small projects.

2.6 Rapid Software Development

Businesses operate in a fast-changing environment so it is practically impossible to produce a stable set of software requirements. Software must evolve quickly to reflect changing business needs. Rapid development and delivery are therefore often the most critical requirement for new software systems.

A. Agile Methods

Agile methods were developed in the late 1990s to reduce overheads in the process (e.g., by limiting documentation) and to respond quickly to changing requirements without excessive rework. The **Agile Manifesto** values:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

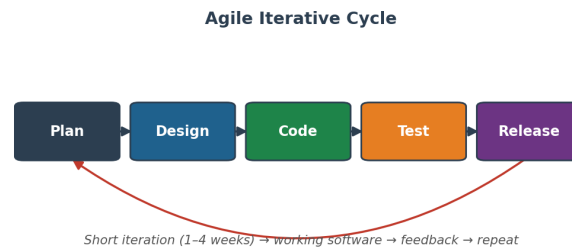


Figure 2.3: Agile iterative cycle

Agile Principle	Description
Customer involvement	Customers should be closely involved throughout; they provide and prioritise requirements and evaluate iterations.
Incremental delivery	Software developed in increments; customer specifies requirements in each increment.
People not process	Team skills recognised and exploited; team members develop their own ways of working.
Embrace change	Expect requirements to change and design the system to accommodate change.
Maintain simplicity	Focus on simplicity in both software and process; eliminate complexity actively.

B. Extreme Programming (XP)

XP is the best-known agile method. It takes an “extreme” approach to iterative development: new versions may be built several times per day, increments are delivered to customers roughly every two weeks, all tests are run for every build, and the build is accepted only if tests run successfully.

XP Practice	Description
Incremental planning / user stories	Requirements recorded on story cards; stories are selected for a release based on time and priority.
Small releases	Minimal useful functionality is developed first and released early; later releases add functionality.
Simple design	Enough design to meet current requirements and no more.

XP Practice	Description
Test-first development	Automated unit test framework used to write tests <i>before</i> the new functionality.
Refactoring	All developers continually refactor code as soon as improvements are found, keeping code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing support.
Collective ownership	Pairs work on all areas so no islands of expertise develop.
Continuous integration	Work integrated into the whole system as soon as a task is complete; all tests must pass.
Sustainable pace	Large amounts of overtime are not acceptable as it reduces code quality.
On-site customer	A representative of the end user should be available full time for the XP team.

C. Rapid Application Development (RAD)

RAD is a linear-sequential-style model with an extremely **short development cycle (60–90 days)** that uses component-based construction. It is suited to business information systems that can be modularised. Team-based development uses reusable components and automated tools (4GL, GUI builders, code generators).

- **Business modelling** – information flow among business functions is modelled.
- **Data modelling** – data objects needed are defined with attributes and relations.
- **Process modelling** – data objects are transformed to achieve the information flow.
- **Application generation** – automated tools and reusable components build the software.
- **Testing and turnover** – reused components are already tested, reducing overall testing time.

Note: RAD needs sufficient human resources for multiple teams, commitment from both developers and customers, and a system that can be modularised. It is unsuitable for high technical risk.

D. Software Prototyping

A **prototype** is an initial version of a system used to demonstrate concepts, try out design options and find out more about the problem and its solutions. Rapid iterative development of the prototype is essential so that costs are controlled.

Benefits of Prototyping	Problems
Misunderstandings between developers and users are identified	Prototype may not be used the same way as the final system
Missing user services and confusing services are detected	Users may pressure for delivery of the prototype as the product
Working system available early to demonstrate feasibility and usefulness	Non-functional requirements (performance, reliability) usually ignored in the prototype
Prototype can be basis for writing the system specification	Prototypes usually cannot be tuned to meet non-functional requirements

2.7 Rational Unified Process (RUP)

RUP is a modern generic process model derived from the work on UML and the associated Unified Software Development Process. It brings together elements of all generic process models, supports iteration and illustrates good practice in specification and design. RUP is normally described from three perspectives: a **dynamic** perspective (phases), a **static** perspective (process activities) and a **practice** perspective (good practices).

RUP Phase	Goal
Inception	Establish the business case for the system; identify external entities and their interactions; assess contribution to business. Ends with go / no-go decision.
Elaboration	Develop understanding of the problem domain, establish an architectural framework, develop the project plan and identify key risks.
Construction	System design, programming and testing; parts developed in parallel and integrated. Outcome is a working software system and documentation.
Transition	Move the system from the development community to the user community and make it work in a real environment (beta testing, training, deployment).

Six best practices of RUP: develop software iteratively; manage requirements; use component-based architectures; visually model software (UML); verify software quality; control changes to software.

2.8 Computer-Aided Software Engineering (CASE)

Overview of the CASE Approach

CASE refers to software used to support software process activities such as requirements engineering, design, program development and testing. CASE tools automate routine tasks, reduce errors, improve consistency and support team communication. CASE can support

software process activities either as *tools* (support individual tasks), *workbenches* (support a phase such as design) or *environments* (support all or most of the process).

- Automatic generation of documentation and code from design models
- Maintaining a central repository so all team members share consistent information
- Checking consistency and completeness of models and specifications
- Increased productivity and better quality, though returns are limited because software engineering is a creative, design-centred activity

Classification of CASE Tools

Category	Purpose	Examples
Upper CASE tools	Support early phases – planning, requirements, analysis and design	Requirement editors, UML/ER modelling tools, prototyping tools
Lower CASE tools	Support later phases – coding, testing, debugging, maintenance	Code generators, compilers, debuggers, test tools
Integrated CASE (I-CASE)	Cover the entire life cycle with a shared repository	Full development environments
Tools	Support individual process tasks	Compilers, editors, checking tools
Workbenches	Support a process phase such as specification or design	Analysis and design workbenches, testing workbench
Environments	Support all or a substantial part of the process	Integrated development environments

Choosing a Process Model – Example

A college wants a student-registration system. Requirements are clear and stable, and the user community is small.

Suggested model: Waterfall or incremental delivery. For a startup building a new mobile app with uncertain needs → **Agile / prototyping**. For a large, high-risk defence system → **Spiral**.

Key Takeaways

- ✓ Every process includes specification, design and implementation, validation and evolution.
- ✓ **Waterfall** is sequential and document-driven; good only for stable requirements.
- ✓ **Evolutionary** and **prototyping** approaches help when requirements are unclear.
- ✓ **CBSE** assembles systems from reusable components to cut cost and time.
- ✓ **Incremental delivery** gives customers early value; **Spiral** adds explicit risk analysis in every loop.
- ✓ **Agile / XP** emphasise working software, customer collaboration, test-first, pair programming, refactoring.
- ✓ **RAD** offers 60–90 day delivery using reusable components. **RUP** has four phases: inception, elaboration, construction, transition.
- ✓ **CASE** tools automate process activities – classified as upper, lower and integrated CASE.

— *End of Unit 2* —