SOFTWARE EVOLUTION

Prepared By: Natabar Khatri

New Summit College

Unit 9

INTRODUCTION

- Software evolution is the continuous process of changing, updating, and maintaining software systems after their initial development and deployment.
- Unlike physical systems that wear out, software systems degrade in a different way: they become less useful and more costly to change if they are not actively adapted to their evolving environment.
- Evolution is not an exception but a fundamental and dominant aspect of the software lifecycle.
- Software must evolve for a multitude of reasons, which can be categorized as follows:
  - **Changing Business Needs and User Expectations:** As a business grows, enters new markets, or adapts its processes, its software must change to support these new requirements.
  - **Error Correction (Bug Fixing):** Defects are inevitably discovered during operational use and must be corrected.
  - **Environmental Adaptations:** Changes to the underlying hardware, operating systems, libraries, or other platform components force the software to adapt to remain functional.
  - **Performance and Quality Improvements:** Systems may need to be optimized for speed, reliability, or other non-functional characteristics.
  - **Competitive Pressure:** For commercial software products and apps, new features must be added to keep pace with or surpass competitors.

INTRODUCTION

The Cost of Evolution

- Historical data suggests that **between 60% and 90% of total software costs are dedicated to evolution.** This is because software is a critical business asset, and companies must invest in its change to maintain its value.
- Most large companies spend more on maintaining and evolving existing systems than on developing new ones.

Models of Software Evolution

The content describes several models for how evolution is managed:

- **The Spiral Development and Evolution Model**
- **The Discontinuous Model (Software Maintenance)**
- **The Rajlich and Bennett Lifecycle Model**

MODELS OF SOFTWARE EVOLUTION

The Spiral Development and Evolution Model:

- This is a seamless process where the same company is responsible for the software throughout its life. Development is a continuous spiral of requirements, design, implementation, and testing, leading to regular releases.
- The time between these releases has shrunk dramatically - from years to, in some cases, weeks - due to competitive pressures and the need for rapid user feedback.

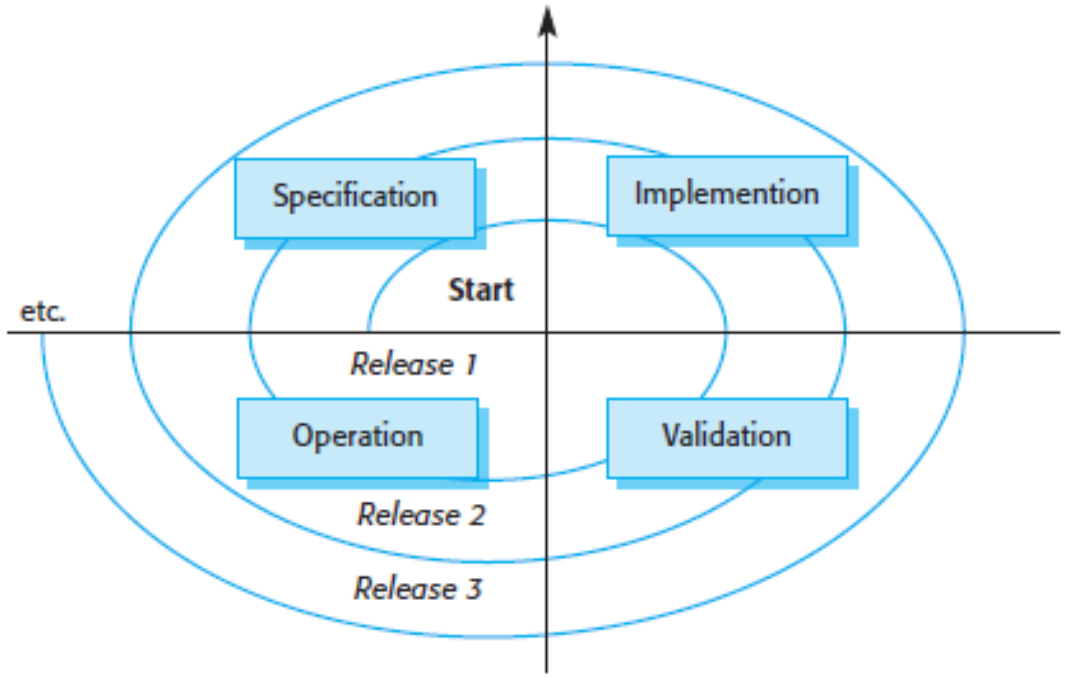

Figure 9.1: A spiral model of development and evolution

MODELS OF SOFTWARE EVOLUTION

The Discontinuous Model (Software Maintenance):

- For custom software, the transition from development to evolution is often not seamless. The customer may take over support using in-house staff or hire a different company.
- The discontinuity introduces extra challenges, such as a lack of proper documentation, leading to additional activities like "program understanding." This process is more accurately termed **software maintenance**.

The Rajlich and Bennett Lifecycle Model:

This model proposes a distinct phased approach:

- **Evolution Phase:** The initial period after deployment where significant, strategic changes are made to the architecture and functionality. The software is highly adaptable.
- **Servicing Phase:** As the software's structure degrades from repeated changes, it becomes increasingly expensive to modify. At this point, only small, tactical, and essential changes are made. The company typically begins planning for the system's replacement.
- **Final Stage and Retirement:** The software is still used, but changes are minimal. Eventually, it is retired, often incurring costs for data migration to a new system.

THE SOFTWARE EVOLUTION PROCESS

- The software evolution process is the structured set of activities used to manage, design, implement, and release changes to an existing software system.
- Unlike initial development, which starts from a blank slate, evolution must work within the constraints of an operational system, making it a cyclical and often complex endeavor.
- There is no single standard process; it varies based on the system's criticality, organizational practices, and the skills of the team involved, ranging from the informal (e.g., for mobile apps) to the highly formalized (e.g., for embedded critical systems).

THE SOFTWARE EVOLUTION PROCESS

The Cyclical Driver: Change Proposals

The entire process is initiated and driven by **change proposals**. These formal or informal suggestions for modification can originate from various sources:

- Unimplemented existing requirements
- Requests for new features and functionality
- Bug reports from users and stakeholders
- New ideas for improvement from the development team itself

As shown in Figure 9.2, the processes of change identification and system evolution form a continuous cycle that persists throughout the system's lifetime.

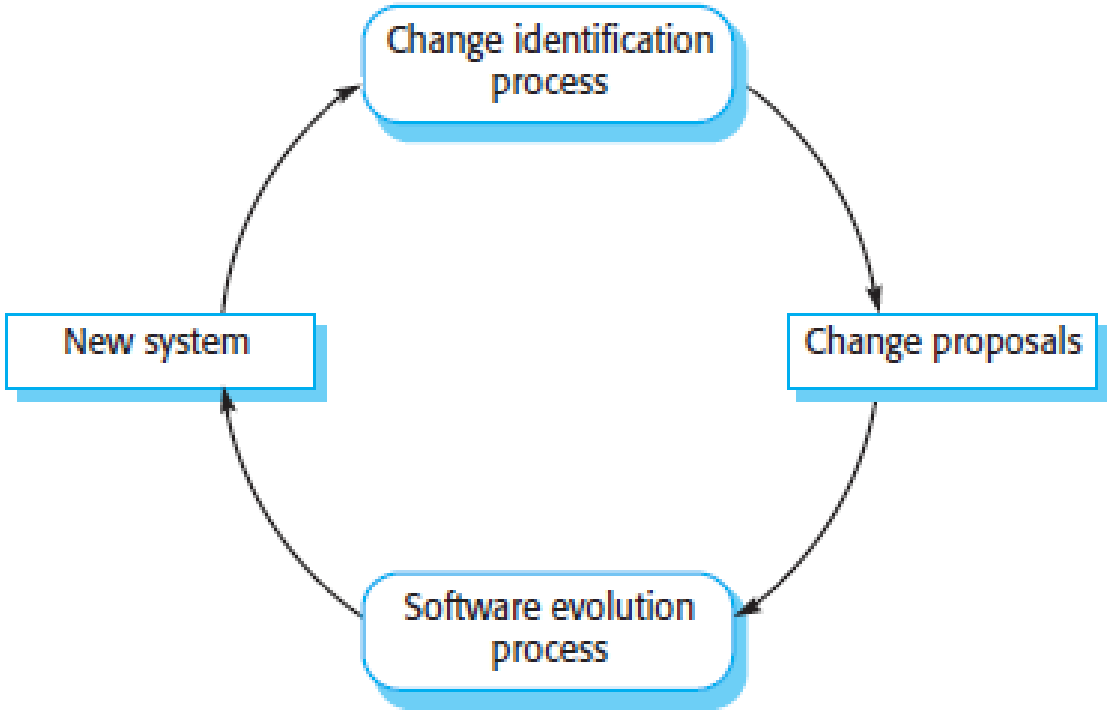

Figure 9.2: Change identification and evolution processes

A GENERAL MODEL OF THE EVOLUTION PROCESS

- Figure 9.3 outlines a general model for managing these changes in a structured, planned manner.
- The key stages are:
  - **Change Requests & Impact Analysis**
  - **Release Planning**
  - **Change Implementation**
  - **System Release**

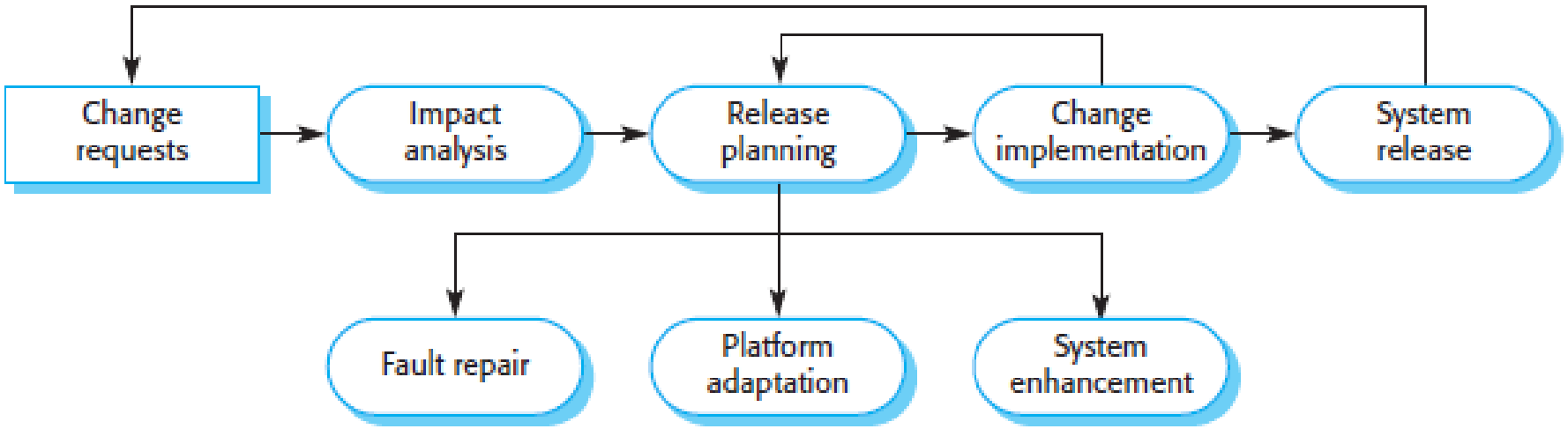

Fig 9.3: A general model of the software evolution process

A GENERAL MODEL OF THE EVOLUTION PROCESS

- **Change Requests & Impact Analysis:** Once a change is proposed, it is not immediately implemented. First, a critical analysis is conducted to determine:
  - Which software components are affected.
  - The potential cost of implementation.
  - The broader impact of the change on the system's stability and functionality.
    This step is essential for informed decision-making and effective change management.
- **Release Planning:** In this phase, all proposed changes—whether they are **fault repairs, platform adaptations, or system enhancements**—are collectively reviewed. Decisions are made about which changes will be bundled into the next version of the system, balancing urgency, resource availability, and strategic goals.

A GENERAL MODEL OF THE EVOLUTION PROCESS

- **Change Implementation:** The approved changes are designed, coded, and validated. This stage can take two forms:
  - **Seamless Evolution:** When the same team handles development and evolution, this is simply another iteration of the development process.
  - **Discontinuous Evolution (Maintenance):** If a different team takes over, the first critical step is **program understanding**. New developers must comprehend the system's structure to ensure their changes do not introduce new problems.
- **System Release:** The modified, tested, and validated system is released to users. The process then immediately iterates, beginning again with new change proposals for the subsequent release.

THE CHALLENGE OF EMERGENCY CHANGES

Urgent changes is a significant real-world challenge. These are emergency fixes required to address:

- Serious system faults or security vulnerabilities.
- Unexpected disruptions caused by environmental changes.
- Unanticipated business changes (e.g., new legislation).
- In these cases, the formal process is shortcut. As shown in Figure 9.4, developers make an emergency fix directly to the code to resolve the immediate problem, often deferring documentation updates. The danger is that this leads to inconsistencies between the code, design, and requirements, accelerating **software ageing** and making future changes more difficult and expensive. While refactoring the quick fix later is ideal, it is often neglected due to ongoing pressures.

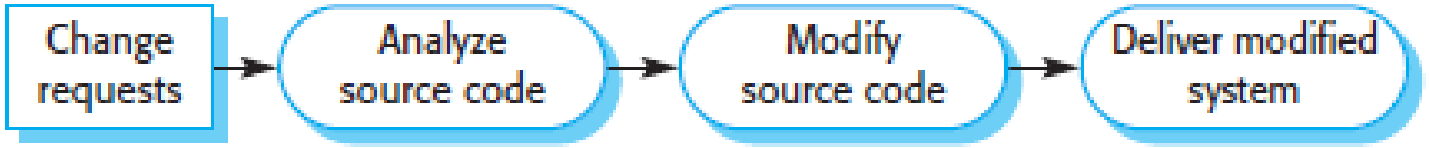

Figure 9.4: The emergency repair process

AGILE METHODS IN EVOLUTION

- Agile methods can be highly effective for evolution, creating a seamless transition from agile development to agile evolution. Techniques like test-driven development, user stories, and Scrum's product backlog are well-suited for prioritizing and implementing incremental changes.
- However, two key challenges can arise during a handover:
- An **agile development team** handing over to a **plan-based evolution team** can cause friction, as the latter may expect detailed documentation that was never created.
- A **plan-based system** being taken over by an **agile evolution team** can be problematic if the code lacks the comprehensive automated tests and refactored structure that agile methods rely on.
- Furthermore, pure agile practices may need adaptation for evolution. It can be difficult to involve users as closely, short cycles may be interrupted by emergency fixes, and release frequencies might need to be adjusted to avoid disrupting business operations.

LEGACY SYSTEMS

- A legacy system is an old, yet still critical, software system that continues to be used because it reliably supports core business functions. However, these systems are built on technologies, programming languages, and hardware that are often obsolete.
- Legacy systems have typically been maintained for decades, and their internal structure may have degraded due to numerous changes, making them expensive and risky to modify or replace.

LEGACY SYSTEMS

The Components of a Legacy System

- **Business Processes, Policies, and Rules:** These are the core of the business. The legacy system often embeds critical business knowledge and rules (e.g., how to calculate an insurance risk). Business processes have frequently evolved to work *with* the system, becoming constrained by its functionality.
- **Application Software:** The actual programs that provide business services. This is often a collection of programs (e.g., millions of lines of COBOL code) developed at different times.
- **Application Data:** Vast volumes of data accumulated over the system's lifetime. This data is often inconsistent, duplicated, and spread across multiple, sometimes incompatible, files and databases.
- **Support Software:** The obsolete operating systems, compilers, and databases on which the application software depends. These may no longer be supported by their original vendors.
- **System Hardware:** Often older mainframe computers that are expensive to maintain, incompatible with modern IT policies, and difficult to integrate with newer systems.

These layers are interdependent. A change in one layer, such as upgrading the hardware to improve performance, can necessitate costly changes in the software layers above and below it.

THE LEGACY SYSTEM DILEMMA: WHY NOT JUST REPLACE THEM?

- Despite their challenges, businesses are often reluctant to replace legacy systems due to the immense cost and risk involved.
- The key reasons for this reluctance are:
  - **Lack of Complete Specification:** There is rarely an up-to-date specification of what the system actually does, making it impossible to specify a functionally identical replacement.
  - **Intertwined Business Processes:** Business processes have adapted to the legacy system's quirks. Replacing the system forces a costly and disruptive re-engineering of these processes.
  - **Embedded Business Rules:** Critical business rules exist only within the code. If not properly extracted and documented during replacement, they can be lost, leading to serious business consequences.
  - **Inherent Risk of New Development:** New software projects are risky and may fail to deliver on time, on budget, or to meet business needs. Sticking with a known, working system avoids this uncertainty.

THE CHALLENGES OF MAINTAINING LEGACY SYSTEMS

While keeping the system avoids replacement risk, maintenance becomes progressively more difficult and expensive due to:

- **Degraded Structure:** Years of changes corrupt the original software architecture.
- **Obsolete Technology:** Reliance on old languages like COBOL leads to a shortage of skilled programmers.
- **Inadequate Documentation:** Often, the source code is the only reliable documentation.
- **Poor Data Quality:** Data is often inconsistent, duplicated, and spread across incompatible structures.
- **Security Vulnerabilities:** Systems designed before the internet era may have inherent security weaknesses.

LEGACY SYSTEM MANAGEMENT

Given the limited budgets for maintaining legacy systems, organizations must strategically decide how to manage their portfolio. This involves a systematic assessment from two perspectives, leading to one of four strategic options.

1. The Assessment Process

- Decisions are based on an evaluation of both **Business Value** and **Technical Quality**.
  - **Assessing Business Value:**
    This determines how critical the system is to the business. Key questions include:
    - **Use:** How often and by how many people is the system used? Is it essential, even if used infrequently?
    - **Supported Processes:** Does the system support efficient, modern business processes, or does it force the use of obsolete ones?
    - **Dependability:** How do system failures impact customers and employees?
    - **Outputs:** How important are the system's outputs to the business's success?

LEGACY SYSTEM MANAGEMENT

Assessing Technical Quality and Environment:
This evaluates the health of the system itself and its platform.

- **Environment:** Assesses the hardware and support software, including supplier stability, failure rates, age, and maintenance costs.
- **Application:** Assesses the software's understandability, documentation, data structure, and the availability of skilled personnel to maintain it.

LEGACY SYSTEM MANAGEMENT

2. The Strategic Options

By plotting systems on a chart of Business Value versus Technical Quality, organizations can identify four clusters, each with a recommended strategy:

- **Scrap the System:** For systems with **Low Quality, Low Business Value**. These are costly to maintain and contribute little. They should be decommissioned.
- **Continue Maintenance:** For systems with **High Quality, Low Business Value** (not worth replacing) or **High Quality, High Business Value** (critical and healthy). Normal maintenance continues unless expensive changes become necessary.
- **Reengineer the System:** For systems with **Low Quality, High Business Value**. These are business-critical but expensive to maintain. The goal is to restructure them to improve maintainability, potentially by wrapping components with modern interfaces.
- **Replace the System:** For systems with **Low Quality, High Business Value** (if reengineering is not feasible) or when the environment (e.g., hardware) can no longer be supported. This is a high-risk but sometimes necessary option.

SOFTWARE MAINTENANCE

Software maintenance is the comprehensive process of modifying a software system after it has been delivered and deployed. This term is most commonly applied to custom software where the team responsible for maintenance is different from the original development team. Maintenance encompasses a wide spectrum of changes, from simple bug fixes to significant enhancements that add new functionality.

Types of Software Maintenance

Maintenance activities are generally categorized into three types:

- **Fault Repairs (Corrective Maintenance):** This involves fixing coding errors, design flaws, or specification mistakes. While coding errors are usually cheap to fix, errors in design or requirements are far more expensive as they may require rewriting large portions of the system.
- **Environmental Adaptation (Adaptive Maintenance):** This is necessary when the system's environment changes, such as upgrades to the hardware, operating system, or other support software. The application must be modified to remain operational in this new context.
- **Functionality Addition (Perfective Maintenance):** This involves adding new features or modifying existing ones to support new or changed business requirements. This type of maintenance often requires the most effort and extensive changes to the software.

WHY MAINTENANCE IS SO EXPENSIVE

Adding features during maintenance is typically more expensive than implementing them during initial development for several key reasons:

- **Program Understanding:** A new team must spend significant time understanding the existing system before they can safely make changes.
- **Lack of Incentive for Maintainability:** If development and maintenance are separate, the development team has no incentive to write easily maintainable code, as they will not bear the cost of future changes.
- **Poor Image of Maintenance:** Maintenance is often viewed as less skilled work and is assigned to junior staff, which can lead to inefficient changes and further code degradation.
- **Structural Degradation:** Over time, as changes are made, the original structure of the software degrades, making it increasingly complex, poorly documented, and harder to understand and modify.

MAINTENANCE PREDICTION

To manage costs and resources effectively, organizations practice **maintenance prediction**, which involves forecasting the changes a system will require and identifying the parts that will be most costly to maintain.

This involves assessing:

- **System Change Predictions:** By evaluating the number of system interfaces, the volatility of requirements, and the business processes the system supports, one can predict the number and nature of future change requests.
- **Maintainability Predictions:** The complexity of software components is a key indicator of future maintenance cost. After deployment, process metrics can be used, such as:
  - The number of corrective maintenance requests.
  - The average time required for impact analysis.
  - The average time to implement a change.
  - The number of outstanding change requests.
    An increase in these metrics suggests a decline in the system's maintainability.

SOFTWARE REENGINEERING

- When a system becomes too difficult and expensive to maintain, a solution is **software reengineering**. This process restructures and documents existing software to improve its comprehensibility and reduce future maintenance costs, *without changing its core functionality*.
- **Advantages over Replacement:**
- **Reduced Risk:** Avoids the high risk of failure associated with completely redeveloping a business-critical system.
- **Reduced Cost:** Is often significantly cheaper than building a new system from scratch.

SOFTWARE REENGINEERING

The Reengineering Process includes:

- **Source Code Translation:** Converting the code to a modern language.
- **Reverse Engineering:** Extracting design and specification information from the source code.
- **Program Structure Improvement:** Restructuring the code to improve readability (e.g., simplifying control structures).
- **Program Modularization:** Grouping related functionalities and removing redundancies.
- **Data Reengineering:** Cleaning and restructuring the system's data.

REFACTORING

Refactoring is a complementary, ongoing practice of making small, behavior-preserving improvements to a program's internal structure to reduce complexity and improve readability. It is a form of "preventative maintenance."

Key differences from Reengineering:

- **Reengineering** is a major, one-off project on a legacy system.
- **Refactoring** is a continuous process integrated into the development and evolution lifecycle, as championed by agile methods.

Refactoring addresses "bad smells" in code, such as:

- **Duplicate Code**
- **Long Methods**
- **Overuse of Switch Statements**
- **Data Clumping** (groups of data that always appear together)
- **Speculative Generality** (unused "just-in-case" code)