Software Evolution

Chapter 9, Software Engineering, 9th Edition by Ian Sommerville

Why software evolution is necessary

- Software development does not stop when a system is delivered but continues throughout the lifetime of the system
  - After a system has been deployed, it has to change if it is to remain useful
- Business changes and changes to user expectations generate new requirements for the existing software
  - Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system’s environment
- Erlikh (2000) suggests that **85–90%** of organizational software costs are **evolution costs**

“brownfield software development”

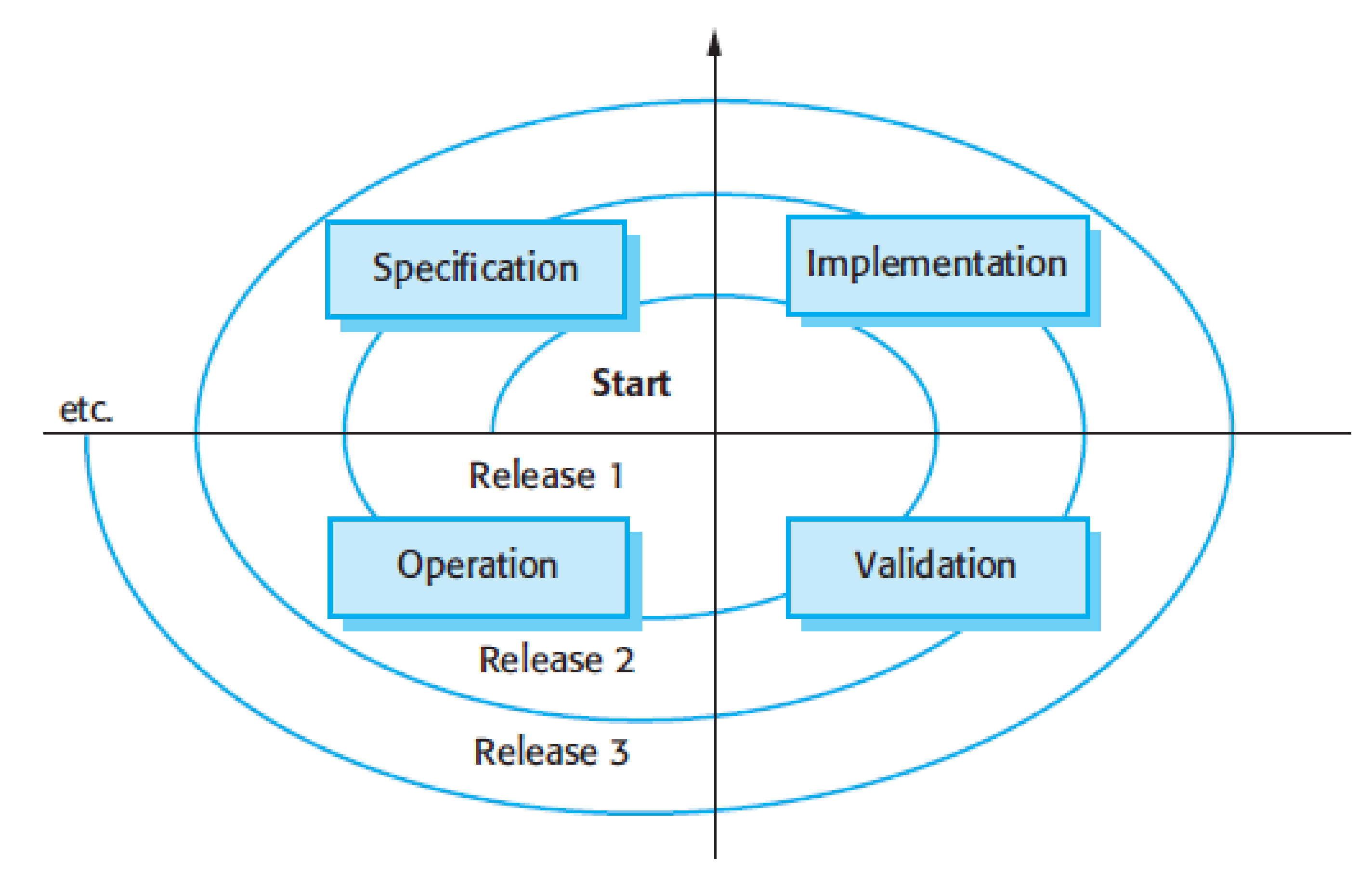
- Hopkins and Jenkins (2008) have coined the term ‘brownfield software development’ to describe **situations in which software systems have to be developed and managed in an environment where they are dependent on many other software systems**
- The evolution of a system can rarely be considered in isolation
- Changes to the environment lead to system change that may then trigger further environmental changes

- Systems that have to evolve increases the difficulties and costs of evolution
- You have to understand and analyze an impact of a proposed change on the system itself
- You may also have to assess how this may affect other systems in the operational environment

Software Lifetime

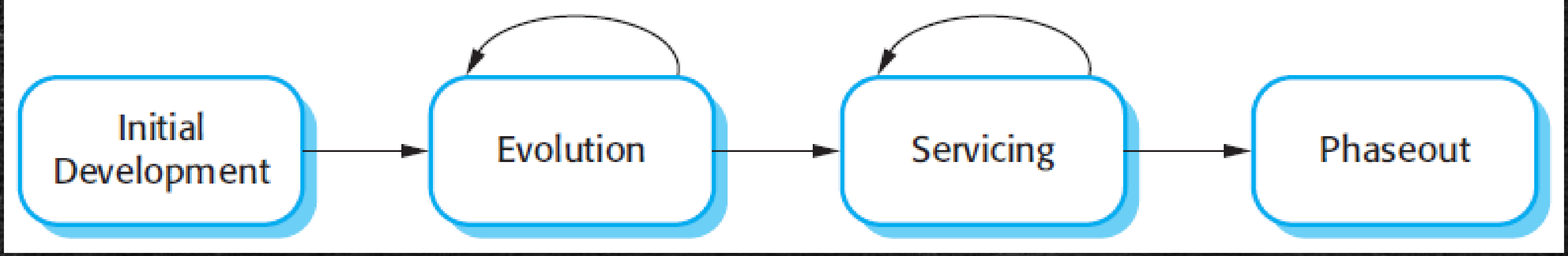
- Useful software systems often have a very long lifetime
- Large military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more
- Business systems are often more than 10 years old

A Spiral Model of Development and Evolution



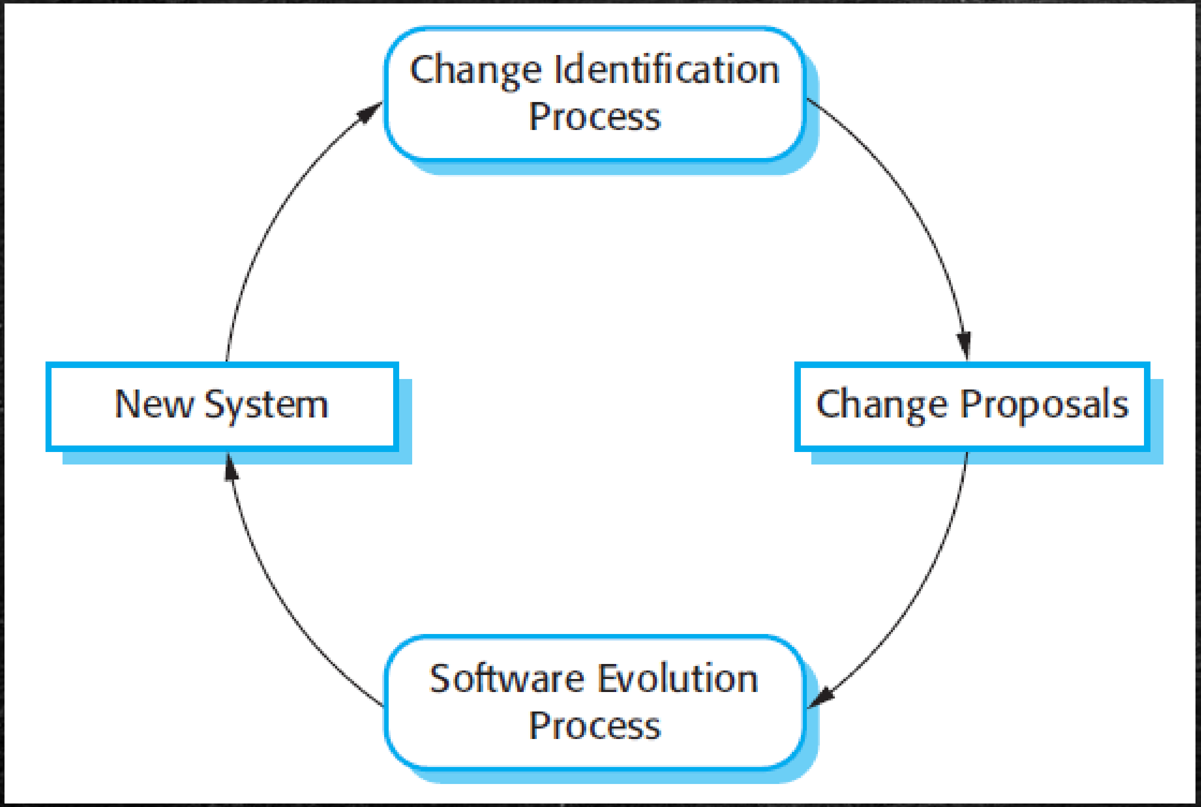
Most packaged software products are developed using this approach

An alternative view of the software evolution life cycle



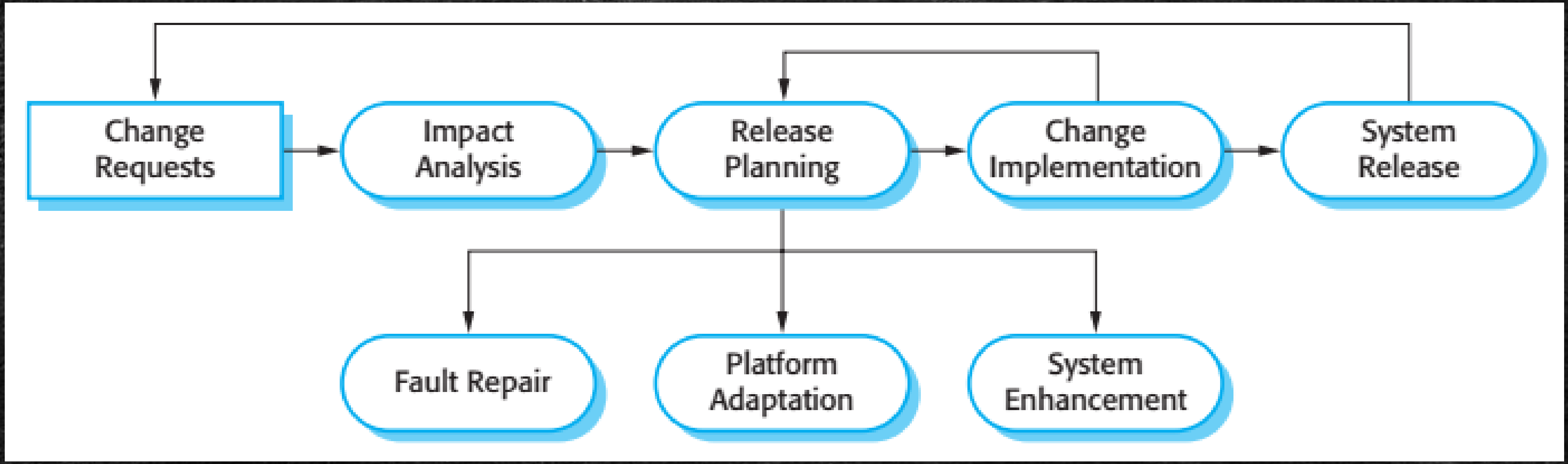
- Rajlich and Bennett (2000) proposed this model
- **Evolution** is the phase in which significant changes to the software architecture and functionality may be made
- During **servicing**, the only changes that are made are relatively small, essential changes

Change identification and evolution processes



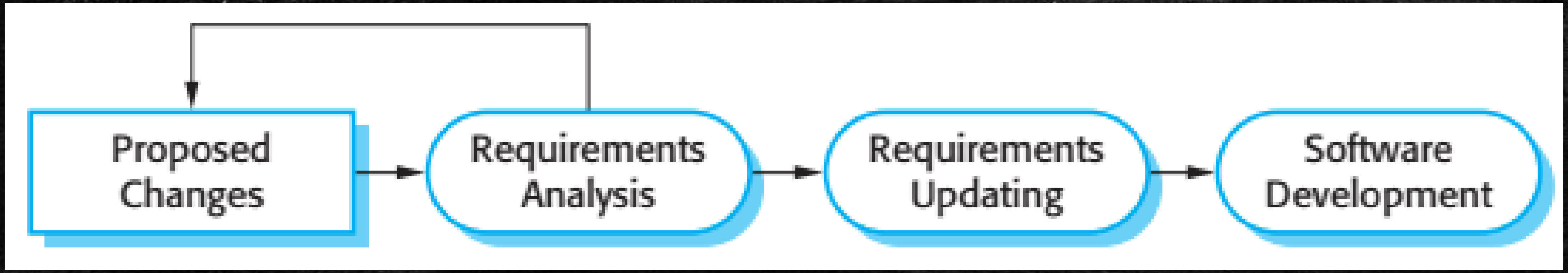
- In some organizations, evolution may be an **informal process** where change requests mostly come from conversations between the system users and developers
- In other companies, it is a **formalized process** with structured documentation produced at each stage in the process
- The processes of change identification and system evolution are cyclic and continue throughout the lifetime of a system

The Software Evolution process



- adapted from Arthur (1988)
- The process includes the fundamental activities of **change analysis**, **release planning**, **system implementation**, and **releasing a system** to customers
- The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change
- If the proposed changes are accepted, a new release of the system is planned

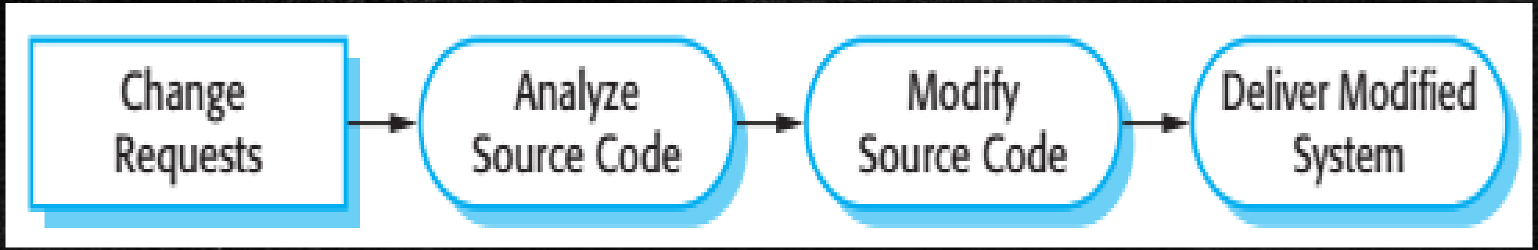
- During the evolution process, the requirements are analyzed in detail and implications of the changes emerge that were not obvious in the earlier change analysis process
  - This means that the proposed changes may be modified and further customer discussions may be required before they are implemented
- Change requests sometimes relate to system problems that have to be tackled urgently



Change implementation should modify the system specification, design, and implementation to reflect the changes to the system

Three reasons why changes arise

1. If a serious system fault occurs that has to be repaired to allow normal operation to continue
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system



- Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem
- The danger is that the requirements, the software design, and the code become inconsistent

Lehman’s laws (1)

- Lehman’s laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution
- The first law states that **system maintenance is an inevitable process**
  - As the system’s environment changes, new requirements emerge and the system must be modified
  - When the modified system is reintroduced to the environment, this promotes more environmental changes, so the evolution process starts again

Lehman's laws (2)

- The second law states that, **as a system is changed, its structure is degraded**
  - The only way to avoid this happening is to invest in preventative maintenance
  - You spend time improving the software structure without adding to its functionality.
  - This means additional costs, over and above those of implementing required system changes
- The third law suggests that **large systems have a dynamic of their own** that is established at an early stage in the development process.
  - This determines the gross trends of the system maintenance process and limits the number of possible system changes.

Lehman’s laws (3)

- The fourth law suggests that **most large programming projects work in a ‘saturated’ state**
  - That is, a change to resources or staffing has imperceptible effects on the long-term evolution of the system
  - This is consistent with the third law, which suggests that program evolution is largely independent of management decisions
  - This law confirms that large software development teams are often unproductive because communication overheads dominate the work of the team

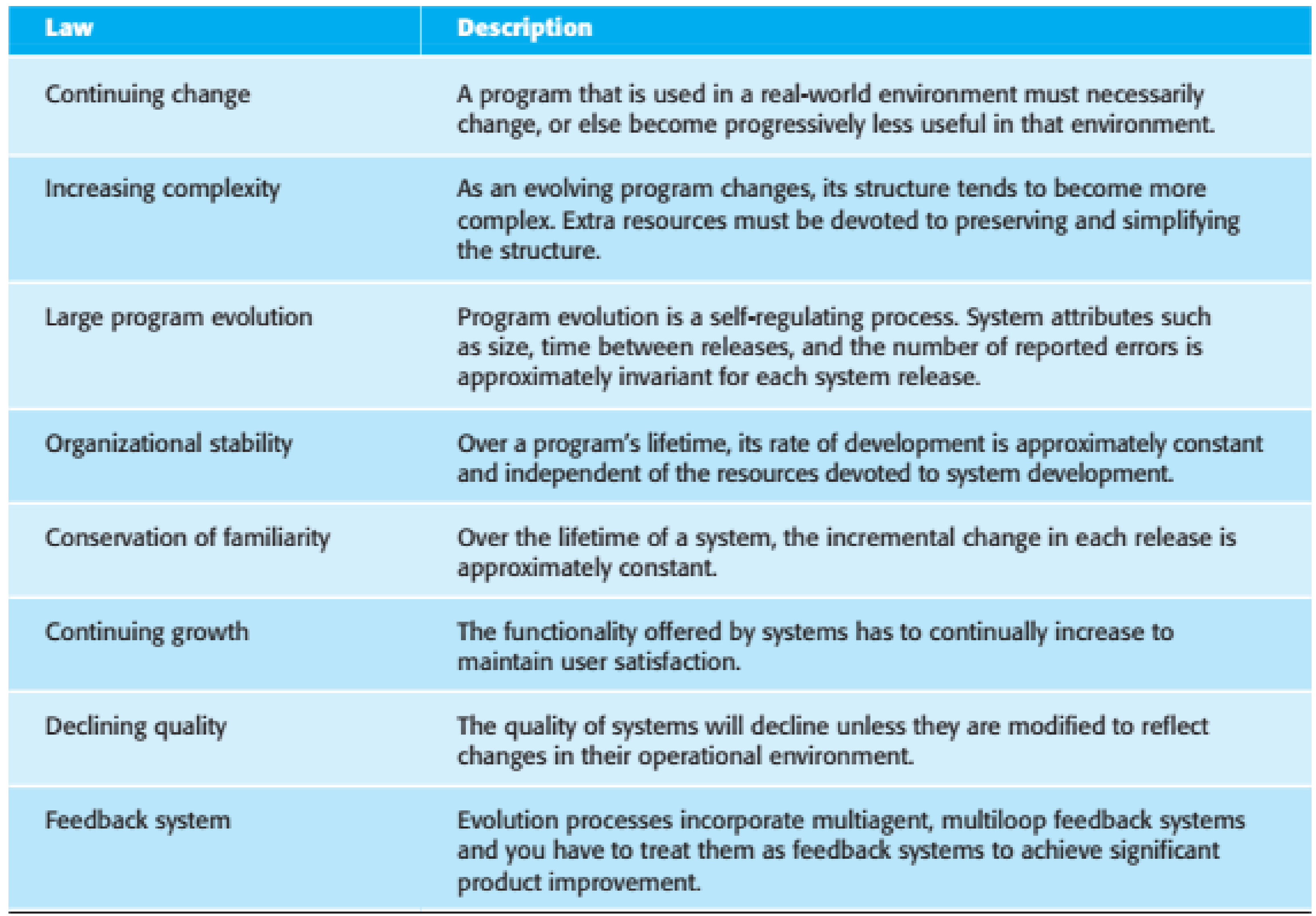
Lehman's laws (4)

- Lehman's fifth law is concerned with the **change increments in each system release**
  - Adding new functionality to a system inevitably introduces new system faults
  - The more functionality added in each release, the more faults there will be
  - Therefore, a large increment in functionality in one system release means that this will have to be followed by a further release in which the new system faults are repaired
  - Relatively little new functionality should be included in this release
  - This law suggests that you should not budget for large functionality increments in each release without taking into account the need for fault repair.

Lehman’s laws (5)

- The sixth and seventh laws are similar and essentially say that **users of software will become increasingly unhappy with it unless it is maintained and new functionality is added to it**
- The final law reflects the most recent work on **feedback processes**, although it is not yet clear how this can be applied in practical software development

Lehman’s Laws



Law	Description
Continuing change	A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.
Feedback system	Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Software Maintenance

- The general process of changing a system after it has been delivered
- The term is usually applied to custom software in which separate development groups are involved before and after delivery
- The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or accommodate new requirements
- Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system
- There are three types of software maintenance, namely bug fixing, modifying the software to work in a new environment, and implementing new or changed requirements

Three types of software maintenance (1)

Fault repairs

- Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components
- Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.

Three types of software maintenance (2)

Environmental adaptation

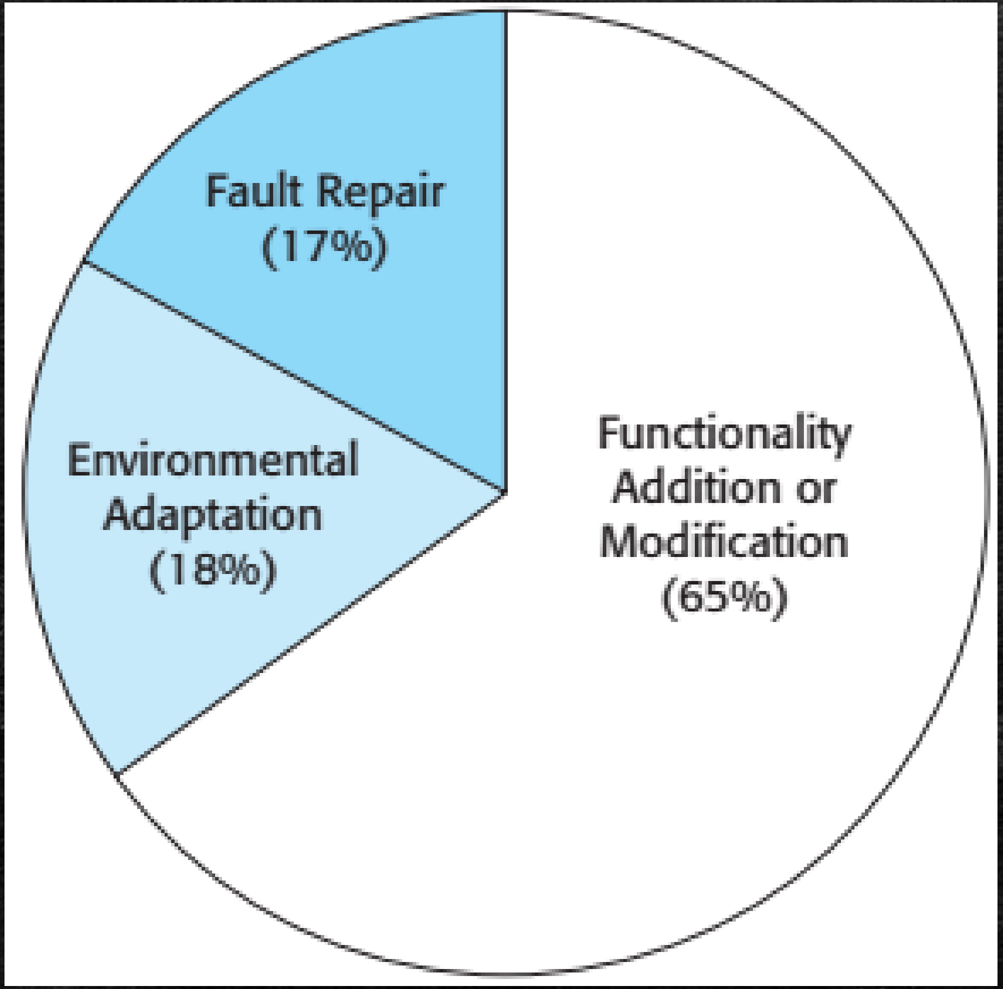
- This type of maintenance is required when some aspect of the system’s environment such as the hardware, the platform operating system, or other support software changes
- The application system must be modified to adapt it to cope with these environmental changes

Three types of software maintenance (3)

Functionality addition

- This type of maintenance is necessary when the system requirements change in response to organizational or business change
- The scale of the changes required to the software is often much greater than for the other types of maintenance

Maintenance Effort Distribution



An approximate distribution of maintenance costs

- The specific percentages will vary from one organization to another but, universally, repairing system faults is not the most expensive maintenance activity
- Evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort

Legacy Systems (1)

- Legacy systems are old systems that are still useful and are sometimes critical to business operation
- They may be implemented using outdated languages and technology or may use other systems that are expensive to maintain
- Often their structure has been degraded by change and documentation is missing or out of date
- Nevertheless, it may not be cost effective to replace these systems
- They may only be used at certain times of the year or it may be too risky to replace them because the specification has been lost

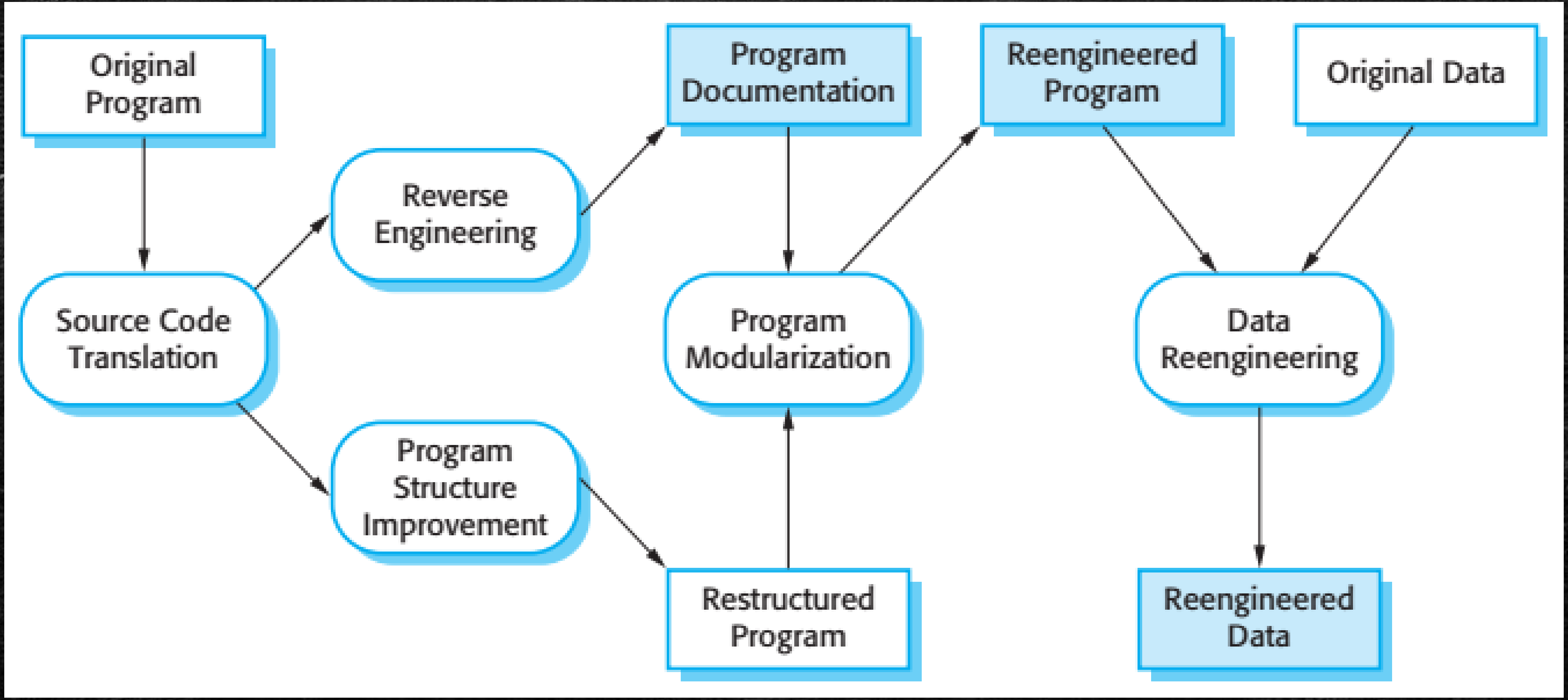
Legacy Systems (2)

- The business value of a legacy system and the quality of the application software and its environment should be assessed to determine whether the system should be:
  - Replaced
  - Transformed, or
  - Maintained

Software Reengineering

- Concerned with restructuring and redocumenting software to make it easier to understand and change
- To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability
- Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, and modifying and updating the structure and values of the system’s data
- The functionality of the software is not changed and, normally, you should try to avoid making major changes to the system architecture

The Reengineering Process (1)



The Reengineering Process (2)

Source code translation

- Using a translation tool, the program is converted from an old programming language to a more modern version of the same language or to a different language

Reverse engineering

- The program is analyzed and information extracted from it
- This helps to document its organization and functionality
- This process is usually completely automated.

The Reengineering Process (3)

Program structure improvement

- The control structure of the program is analyzed and modified to make it easier to read and understand
- This can be partially automated but some manual intervention is usually required

The Reengineering Process (4)

Program modularization

- Related parts of the program are grouped together and, where appropriate, redundancy is removed
- In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository)
- This is a manual process

The Reengineering Process (5)

Data reengineering

- The data processed by the program is changed to reflect program changes
- This may mean redefining database schemas and converting existing databases to the new structure
- You should usually also clean up the data
- This involves finding and correcting mistakes, removing duplicate records, etc.
- Tools are available to support data reengineering

Benefits of Reengineering (1)

Two important benefits from reengineering rather than replacement:

Reduced risk

- There is a high risk in redeveloping business-critical software
- Errors may be made in the system specification or there may be development problems
- Delays in introducing the new software may mean that business is lost and extra costs are incurred

Benefits of Reengineering (2)

Reduced cost

- The cost of reengineering may be significantly less than the cost of developing new software
  - Ulrich (1990) quotes an example of a commercial system for which the reimplementation costs were estimated at $50 million
  - The system was successfully reengineered for $12 million

Refactoring

- Refactoring, making small program changes that preserve functionality, can be thought of as preventative maintenance that reduces the problems of future change
- The process of making improvements to a program to slow down degradation through change (Opdyke and Johnson, 1990)
- It means modifying a program to improve its structure, to reduce its complexity, or to make it easier to understand
- When you refactor a program, you should not add functionality but should concentrate on program improvement

Comparison between Reengineering and Refactoring

Reengineering

- takes place after a system has been maintained for some time and maintenance costs are increasing
- Makes use automated tools to process and reengineer a legacy system to create a new system that is more maintainable

Refactoring

- a continuous process of improvement throughout the development and evolution process
- It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system

Examples of “bad smells” that can be improved through refactoring (1)

Fowler et al. (1999)

Duplicate code

- The same of very similar code may be included at different places in a program
- This can be removed and implemented as a single method or function that is called as required

Long methods

- If a method is too long, it should be redesigned as a number of shorter methods

Examples of “bad smells” that can be improved through refactoring (2)

Fowler et al. (1999)

Switch (case) statements

- These often involve duplication, where the switch depends on the type of some value
- The switch statements may be scattered around a program
- In object-oriented languages, you can often use polymorphism to achieve the same thing

Examples of “bad smells” that can be improved through refactoring (3)

Fowler et al. (1999)

Data clumping

- Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccur in several places in a program
- These can often be replaced with an object encapsulating all of the data

Speculative generality

- This occurs when developers include generality in a program in case it is required in future
- This can often simply be removed

END OF PRESENTATION