Chapter 9

Software Evolution
Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about aging systems
Why software change after delivery is inevitable...

- New requirements emerge when the software is used;
- The business environment changes;
- Discovered errors must be repaired;
- New computers and equipment are added to the system;
- The performance or reliability of the system may need to be improved.
Why effective change management is critical...

- The ability to implement and manage change is critical since...
  - Organizations have huge investments in their software systems - they are critical business assets.
  - To maintain the value of these assets, they must be changed and updated as necessary.
- In fact, most of the software budget in large organizations is devoted to evolving existing software rather than developing new software.
A spiral model of development and evolution
Life cycle phases: evolution vs. servicing vs. phase-out

(cont’d)
Life cycle phases: evolution vs. servicing vs. phase-out (cont’d)

- **Evolution**: The system is in operational use and is evolving as new requirements are proposed and implemented.

- **Servicing**: The system remains useful, but the only changes made are those required to keep it operational (i.e., bug fixes and changes to reflect changes in the software’s environment). No new functionality is added.

- **Phase-out**: The software may still be used but no further changes are made to it.
Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about aging systems
Evolution processes

- Software evolution processes depend on
  - The *type of software* being maintained;
  - The *development processes* used;
  - The *skills and experience* of the people involved.

- *Proposals for change* ("change requests") are the drivers of system evolution…
The software evolution process

Scope, feasibility, cost?
The software evolution process

Scope, feasibility, cost?
Change implementation

(cont’d)
Implicit activity, starting with “impact analysis” : PROGRAM analysis
Change implementation (cont’d)

- Can be thought of as an iteration of the development process. However, a critical difference is that the first stage may involve program understanding, especially if the original system developers are not responsible for the change implementation.

- During this stage (and during “impact analysis”) you have to understand:
  - how the program is structured,
  - how it delivers functionality, and
  - how the proposed change might affect the program.
Handling urgent change requests

- **Urgent changes** may have to be implemented without going through all stages of the evolution process *if*…
  - a *serious system fault* has to be repaired to allow normal operation to continue;
  - changes to the system’s environment (e.g., an OS upgrade) have *unexpected effects*;
  - there are business changes that require a *very rapid response* (e.g., the release of a competing product).
The emergency repair process

1. Change requests
2. Analyze source code
3. Modify source code
4. Deliver modified system
The emergency repair process

So what does this leave out of the *nominal* evolution process?
The software evolution process

Scope, feasibility, cost?
Agile methods and evolution

- Agile methods are based on incremental development so the transition from development to evolution *should* be relatively seamless.
- Automated regression testing is particularly valuable when changes are made to a system.
- Changes may be expressed as additional (or modified) user stories and can be prioritized via customer involvement.

(cont’d)
Agile methods and evolution (cont’d)

- However, problems may arise when there is a handover to a separate “evolution team” if...
  - the evolution team is unfamiliar with agile methods and prefers a plan-based approach (no detailed documentation available), or
  - a plan-based approach has been used for development but the evolution team prefers to use agile methods (no automated tests, code may not have been refactored and simplified).
Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about aging systems
Program evolution dynamics

- The study of the *processes of system change*.
- After several *empirical* studies, Belady and Lehman proposed a number of “laws” concerning system change.
- They are actually just *sensible observations* rather than “laws.”
### “Lehman’s laws”

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Continuing change</strong></td>
<td>A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment.</td>
</tr>
<tr>
<td><strong>Increasing complexity</strong></td>
<td>As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.</td>
</tr>
<tr>
<td><strong>Increasing complexity (entropy)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Large program evolution</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Organizational stability</strong></td>
<td>Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development. (the “saturated state” idea)</td>
</tr>
</tbody>
</table>

(cont’d)


<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conservation of familiarity</td>
<td>Over the lifetime of a system, the incremental change in each release is approximately constant.</td>
</tr>
<tr>
<td>Continuing growth</td>
<td>The functionality offered by systems has to continually increase to maintain user satisfaction.</td>
</tr>
<tr>
<td>Declining quality</td>
<td>The quality of systems will decline unless they are modified to reflect changes in their operational environment.</td>
</tr>
<tr>
<td>Feedback system</td>
<td>Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.</td>
</tr>
</tbody>
</table>
Applicability of “Lehman’s laws”? 

- “Lehman’s laws” seem to apply to large systems, developed by large organizations, as they evolve to meet changing business needs.

- It is not clear how they should be modified for
  - systems that incorporate a significant number of COTS components, and
  - small development organizations.
Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about aging systems
Software “maintenance”

- Modifying a program *after* it has been put into use to *effect* software change.
- The term “maintenance” is mostly used for changing *custom software*. *Generic software* products are said to “*evolve*” to create new versions.
- Does not *normally* involve major changes to the system’s underlying architecture.
- Instead, changes are usually implemented by modifying existing components and/or adding new components to the system.
Types of maintenance

- Maintenance to **repair software faults**
- Maintenance to **adapt software to a different operating environment**
- Maintenance to **add to or modify the system’s functionality**
Types of maintenance

- Maintenance to **repair software faults**
- Maintenance to **adapt software to a different operating environment**
- Maintenance to **add to or modify the system’s functionality**

How is maintenance effort typically divided among these three activities?
Distribution of maintenance effort

- Fault repair (17%)
- Environmental adaptation (18%)
- Functionality addition or modification (65%)
Maintenance costs

- Usually greater than development costs (2 to 100 times greater depending on the application).
- Affected by both technical and several non-technical factors...
- Ageing (“legacy”) software can also have high support costs due to old languages/compilers, etc.
- May be offset somewhat by “designing for change” and/or “preventive maintenance”...
Maintenance costs

- Usually greater than development costs (2 to 100 times greater depending on the application).
- Affected by both *technical* and several *non-technical* factors...
- Ageing (“legacy”) software can also have high support costs due to old languages/compilers, etc.
- May be offset somewhat by “designing for change” and/or “preventive maintenance”...
Factors affecting maintenance costs

- **Team stability**: Maintenance costs are reduced if the same staff are involved with the system for some time.

- **Contractual responsibility**: The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.

(cont’d)
Factors affecting maintenance costs (cont’d)

- **Staff skills**: Maintenance staff are often inexperienced and have limited domain knowledge.

- **Program age and structure**: As programs age, their structure is degraded and they become harder to understand and change.
Maintenance costs

- Usually greater than development costs (2 to 100 times greater depending on the application).
- Affected by both technical and several non-technical factors...
- Ageing ("legacy") software can also have high support costs due to old languages/compilers, etc.
- May be offset somewhat by "designing for change" and/or "preventive maintenance"...
Maintenance costs

- Usually greater than development costs (2 to 100 times greater depending on the application).
- Affected by both technical and several non-technical factors...
- Ageing (“legacy”) software can also have high support costs due to old languages/compilers, etc.
- May be offset somewhat by “designing for change” and/or “preventive maintenance”...
Development vs. maintenance costs

reducing lifetime costs by spending more to increase maintainability during development

![Diagram showing development vs. maintenance costs for System 1 and System 2]
Maintenance prediction

- Concerned with assessing (1) which parts of the system may be affected by change and (2) which may have high maintenance costs.
- This involves predicting:
  - the “number of changes” required over time;
  - the system parts that will be impacted; and
  - the maintainability of different system parts.
- The goal is to predict both the long- and short-term costs of maintenance.
Maintenance prediction (cont’d)

- Predicting maintainability
- Predicting system changes
- Predicting maintenance costs

What will be the lifetime maintenance costs of this system?

What parts of the system will be the most expensive to maintain?

What parts of the system are most likely to be affected by change requests?

What will be the costs of maintaining this system over the next year?

How many change requests can be expected?
Maintenance prediction (cont’d)

Predicting maintainability
Predicting system changes
Predicting maintenance costs

What parts of the system are most likely to be affected by change requests?
What will be the lifetime maintenance costs of this system?
What parts of the system will be the most expensive to maintain?
What will be the costs of maintaining this system over the next year?
How many change requests can be expected?
Change prediction

- **Predicting the “number of changes”** requires an understanding of the relationships between a system and its environment. (*Tightly coupled systems require changes whenever the environment is changed.*)

- Factors influencing this relationship are:
  - the number and complexity of system interfaces;
  - the number of inherently volatile system requirements; and
  - the business processes in which the system is used.
Maintenance prediction (cont’d)

Predicting maintainability

Predicting system changes

Predicting maintenance costs

What will be the lifetime maintenance costs of this system?

What parts of the system will be the most expensive to maintain?

What will be the costs of maintaining this system over the next year?

What parts of the system are most likely to be affected by change requests?

How many change requests can be expected?
Product complexity metrics

- Studies have shown that most maintenance effort is spent on a relatively small number of system components.

- Predictions of maintainability can be made by assessing the complexity of system components. E.g.,
  - complexity of control structures;
  - complexity of data structures; and
  - object, method (procedure) and module size.

Process metrics

- Process measurements taken over time may also be used to assess maintainability:
  - number of requests for corrective maintenance;
  - average time required for impact analysis;
  - average time taken to implement a change request; and
  - number of unresolved change requests.

- If any of these are increasing, it may indicate a decline in maintainability.
Maintenance prediction (cont’d)

- Predicting maintainability
- Predicting system changes
- Predicting maintenance costs

- What will be the lifetime maintenance costs of this system?
- What parts of the system will be the most expensive to maintain?
- What will be the costs of maintaining this system over the next year?
- How many change requests can be expected?
- What parts of the system are most likely to be affected by change requests?
Cost prediction

- Use predicted change request and maintainability info together with intuition and experience to predict costs.

- Cost estimation models (e.g., Boehm’s COCOMO 2) utilize estimates of program understandability and new code development effort to predict costs. (See Chap. 24.)
Reengineering and Refactoring
System reengineering

- Involves restructuring/re-writing part or all of a legacy system to improve its structure and maintainability.
- Older systems may have been optimized for performance or space at the expense of understandability, and/or their structure may have become corrupted.
- May involve: re-documenting, refactoring, translating programs to a modern programming language, and modifying/updating data structures.
Advantages of reengineering

- **Reduced risk**: There is high risk in replacing business-critical software. There may be development problems, staffing problems, specification problems, etc.

- **Reduced cost**: The cost of reengineering can be significantly less than the costs of developing new software.
Reengineering process activities

- **Source code translation**: Convert code to a new language.
- **Reverse engineering**: Analyze the program to understand it.
- **Program structure improvement**: Restructure automatically for understandability†.
- **Program modularization**: Reorganize the program structure.
- **Data reengineering**: Clean-up and restructure system data.

† E.g., using a tool such as IBM’s COBOL Restructuring Facility.
The reengineering process

also an activity
Reengineering approaches

- Automated program restructuring
- Program and data restructuring

- Automated source code conversion
- Automated restructuring with manual changes
- Restructuring plus architectural changes

Increased cost
Reengineering cost factors

- The quality of the software to be reengineered
- The tool support available for reengineering
- The extent to which data conversion is required
- The availability of expert staff for reengineering
  - This can be problematic when systems are based on old technology that is no longer widely used.
Preventative maintenance by refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as "preventative maintenance" that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity, or make it easier to understand.
Refactoring vs. reengineering

- **Reengineering** takes place after a system has been maintained for some time because maintenance costs are increasing. You use automated tools (in part) to reengineer a legacy system to create a new system that is more maintainable.

- **Refactoring** is a continuous, largely manual 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.
This program really stinks!
Fowler’s “bad smells” in program code...

- **Duplicate code:** The same or very similar code is included at different places in a program. Remove and implement as a single method or function.

- **Long (large) methods:** Methods that are too long should be redesigned as a number of shorter (smaller) methods.

- **Switch (case) statements:** Often involve duplication where the switch depends on the type of a value. In object-oriented languages, polymorphism can often be used to simplify program structure.

(cont’d)
Fowler’s “bad smells” in program code… (cont’d)

- **Data clumping**: Occurs when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. Can often be replaced with an object that encapsulates all of the data in one place.

- **Speculative generality**: Occurs when overly general programs are used in case this is needed in the future. These can often be simplified.
Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about aging systems
Legacy system management

- Organizations that rely on legacy systems must choose a strategy for evolving these systems...
  - Continue maintaining the system as is;
  - Transform the system through reengineering to improve its maintainability;
  - Replace the system with a new system; or
  - Scrap the system completely and modify business processes so that it is no longer required.

- The strategy chosen should depend on the **system quality** and its **business value**.
Example of a legacy system assessment
Legacy system categories

- Low quality, low business value: These systems should be scrapped.
- Low-quality, high-business value: These make an important business contribution but are expensive to maintain. Should be reengineered or replaced if a suitable system is available.
- High-quality, low-business value: Maintain if expensive changes are not required; otherwise scrap.
- High-quality, high business value: Continue in operation using normal system maintenance.
Business value assessment

- Assessment should take different viewpoints into account:
  - System end-users;
  - Business customers;
  - Line managers;
  - IT managers; and
  - Senior managers.

- Interview different stakeholders and collate results.
Considerations in business value assessment

- **The use of the system**: If systems are only used occasionally or by a small number of people, they *may* have a low business value.

- **The business processes that are supported**: A system *may* have a low business value if it forces the use of inefficient business processes.

- **System dependability**: If a system is not dependable and the problems directly affect business customers, the system has (or will eventually have) a low business value.

- **The system outputs**: If the business depends on system outputs, then the system has a high business value.
System quality assessment

- **Environment assessment**: hardware and associated support software that are required to maintain the system

- **Application assessment**: quality factors affecting dependability, maintainability, documentation, etc.
## Factors used in system environment assessment

<table>
<thead>
<tr>
<th>Factor</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supplier stability</td>
<td>Is the supplier <strong>still in existence</strong>? Is the supplier <strong>financially stable</strong> and likely to <strong>continue in existence</strong>? If the supplier is no longer in business, does someone else maintain the systems?</td>
</tr>
<tr>
<td>Failure rate</td>
<td>Does the <strong>hardware</strong> have a high rate of reported failures? Does the <strong>support software</strong> crash and force system restarts?</td>
</tr>
<tr>
<td>Age</td>
<td>How old is the hardware and software? The older the hardware and support software, the more <strong>obsolete</strong> it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.</td>
</tr>
<tr>
<td>Performance</td>
<td>Is the performance of the system <strong>adequate</strong>? Do performance problems have a significant <strong>effect on system users</strong>?</td>
</tr>
</tbody>
</table>

(cont’d)
### Factors used in system environment assessment (cont’d)

<table>
<thead>
<tr>
<th>Factor</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support requirements</td>
<td>What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.</td>
</tr>
<tr>
<td>Maintenance costs</td>
<td>What are the costs of hardware maintenance and software support licences? Older hardware may have higher maintenance costs than modern systems. Software support may have high annual licensing costs.</td>
</tr>
<tr>
<td>Interoperability</td>
<td>Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?</td>
</tr>
</tbody>
</table>
## Factors used in application quality assessment

<table>
<thead>
<tr>
<th>Factor</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understandability</td>
<td>How difficult is it to <em>understand</em> the source code of the current system? How complex are the control structures that are used? Do <em>variables</em> have meaningful names that reflect their function?</td>
</tr>
<tr>
<td>Documentation</td>
<td>What system documentation is <em>available</em>? Is the documentation complete, consistent, and current?</td>
</tr>
<tr>
<td>Data</td>
<td>Is there an <em>explicit data model</em> for the system? To what extent is data duplicated across <em>files</em>? Is the data used by the system <em>up to date</em> and <em>consistent</em>?</td>
</tr>
<tr>
<td>Performance</td>
<td>Is the performance of the application <em>adequate</em>? Do performance problems have a significant <em>effect</em> on system users?</td>
</tr>
</tbody>
</table>

(cont’d)
## Factors used in application quality assessment (cont’d)

<table>
<thead>
<tr>
<th>Factor</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming language</td>
<td>Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?</td>
</tr>
<tr>
<td>Configuration management</td>
<td>Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?</td>
</tr>
<tr>
<td>Test data</td>
<td>Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?</td>
</tr>
<tr>
<td>Personnel skills</td>
<td>Are there people available who have the skills to maintain the application? Are there people available who have experience with the system? (Or is this a “system environment” issue?)</td>
</tr>
</tbody>
</table>
Application system measurement

- You can also collect quantitative data to make an assessment of the quality of the application system. E.g.,
  - The number of system change requests *(corrupts structure)*
  - The number of different user interfaces used by the system *(complexity, potential inconsistencies/redundancies)*
  - The volume of data used by the system *(complexity, more potential inconsistencies)*
Key points

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.

- For most systems, the costs of software maintenance usually far exceed software development costs.

- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation.

(cont’d)
Key points (cont’d)

- Lehman’s “laws,” such as the notion that change is continuous, describe a number of insights derived from several studies of large system evolution.

- Three types of software maintenance were considered: bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.

- Software **reengineering** is concerned with restructuring and re-documenting software to make it easier to understand and change.

(cont’d)
Key points (cont’d)

- *Refactoring* is a form of *preventative maintenance*.

- The **business value** of a legacy system and the quality of the application and its environment should be assessed to help decide if a system should be replaced, transformed, or maintained.
Chapter 9

Software Evolution