Software Evolution

Chapter 9

Software Evolution
in the textbook

- Introduction
- 9.1 Evolution processes
  - Change processes for software systems.
- 9.2 Program evolution dynamics
  - Understanding software evolution
- 9.3 Software maintenance
  - Making changes to operational software systems
- 9.4 Legacy system management
  - Making decisions about aging software

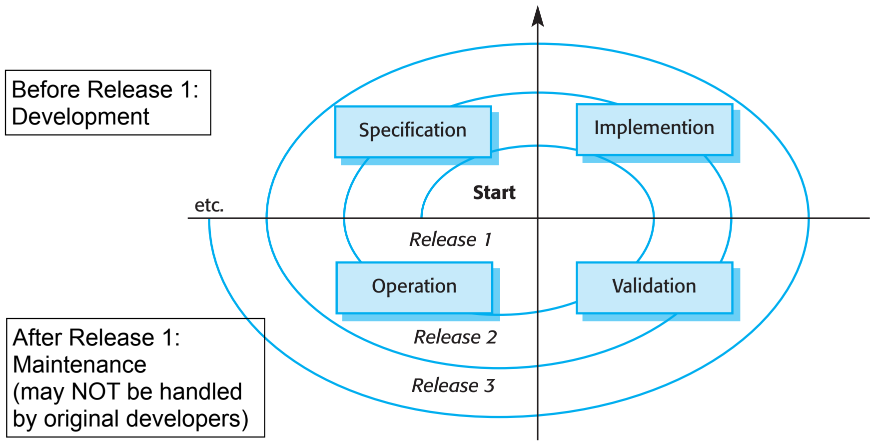
Software change

- Software must change to remain useful
  - The business environment changes
  - Errors must be repaired
  - New computers and equipment are added to the system
  - The performance or reliability of the system may have to be improved.
- Key problem: managing change to existing software systems

Importance of evolution

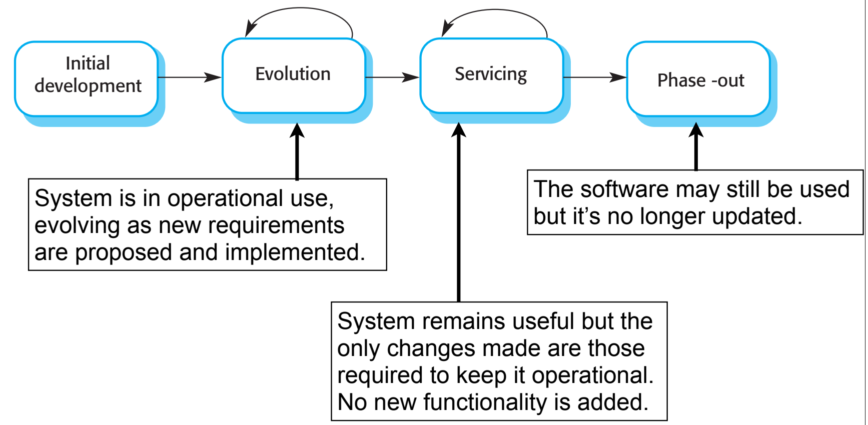
- Software systems: critical and costly business assets.
- Software must be changed/updated to maintain its value
- Goal: use software many years to get return on investment
  - Air traffic control: 30 years
  - Business systems: 10 years
- Large companies spend more on changing existing software than developing new software.

The software evolution process



Bottom Line: All software processes become iterative development.

Evolution vs servicing

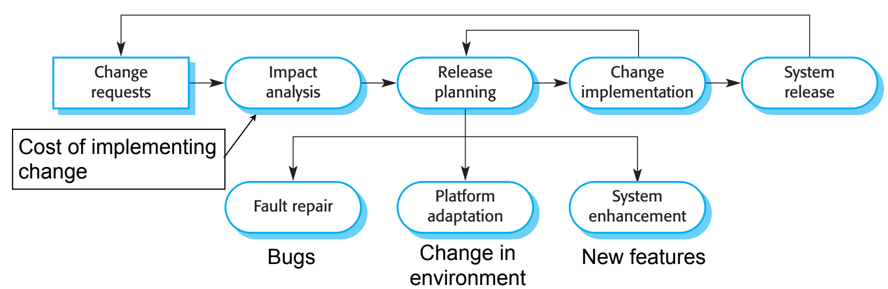


This is an alternative view of the software evolution life cycle

9.1 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.
- Process may be informal or formal
- Proposals for change are the driver for system evolution.

The software evolution process



If evolution is handled by a team other than original development team: program understanding is an additional required step.

Change implementation

- Requirements (follow change process)
  - Analysis
  - Update specifications
  - Validation
- Program understanding, as needed
- Design
  - Update design documents and/or models
- Implementation
  - Modify source code
- Testing

Urgent change requests

- Sources of urgent changes
  - Defect somehow blocking normal operation
  - Changes to the system's environment (e.g. OS upgrade)
  - Business changes requiring rapid response (e.g. the release of a competing product).
- May not be able to follow formal change process
  - Quick and dirty code change
  - Minimal testing
- Problem:
  - Code quality is diminished
  - Specs and code are now inconsistent
- Should: follow formal process later.

Agile methods and evolution

- Transition from development to evolution is seamless.
  - Agile methods and traditional evolution are based on incremental development
- Evolution is equivalent to the later releases.
- No changes to the standard agile methods are necessary.
- Only problem is transitioning to another team.

Handover problems

- Development team used an agile approach but evolution team prefers a plan-based approach.
  - Evolution team may expect detailed documentation to support evolution
- Development team used a plan-based approach but the evolution team prefers agile methods.
  - Automated tests may need to be developed from scratch.
  - Code in the system may need to be refactored.

9.2 Program evolution dynamics

- The study of system change.
- Lehman and Belady (1985): made several major empirical studies of evolving systems.
- Lehman's laws derived from these studies
- Apply to
  - large systems developed by large organizations
  - systems subject to changing business requirements
- Take them into account when planning releases of large systems

Lehman's laws 1-4

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. [Additional cost]
Self-regulation	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 [Invariant work rate]	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development. [More developers don't help]

Lehman's laws 5-7

Law	Description
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant. [features per release]
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The perceived quality of systems will decline unless they are modified to reflect changes in their operational environment.

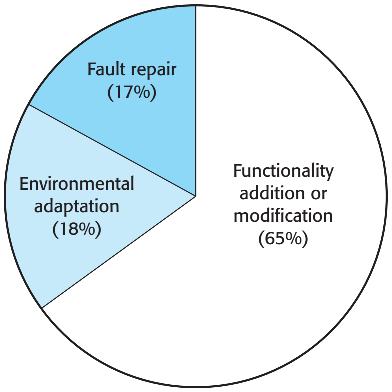
9.3 Software maintenance

- Modifying a program after it has been put into use.
- The term is often applied to cases where a separate development team takes over after delivery.
- Modifications may be simple or extensive
  - But not normally involving major changes to the system's architecture.

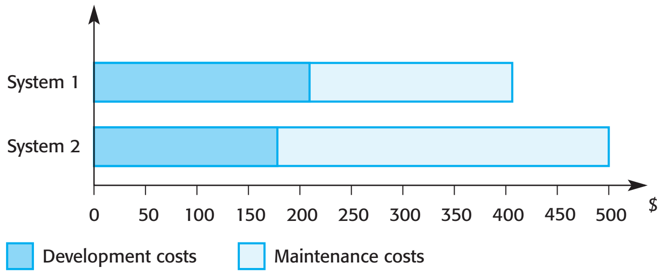
Types of maintenance

- Repairing software faults
  - Changing a system to correct coding, design, or requirements errors.
- Adapting software to a different operating environment
  - Changing a system so that it operates with a modified external system (e.g. new OS, or other software).
- Adding to or modifying the system’s functionality
  - Modifying the system to satisfy new requirements.

Maintenance effort distribution



Development and maintenance costs



In system 1, extra development costs are invested in making the system more maintainable, effectively reducing overall costs.

Maintenance cost factors

why adding new functionality after delivery costs more

- Team stability
  - New team members take time to learn the system.
- Poor development practice
  - The developers of a system may have no incentive to write maintainable software if they won’t be maintaining it.
- Staff skills
  - Maintenance staff are often inexperienced and have limited domain knowledge.
- Program age and structure
  - As programs age, (without refactoring) their structure is degraded--they become harder to understand and change.

9.3.1 Maintenance prediction

Maintenance prediction is concerned with:

- Estimating the overall maintenance costs for a system in a given time period.
- Assessing which parts of the system may cause problems and have high maintenance costs

Complexity metrics

- Studies have shown that
  - Most maintenance effort is spent on a relatively small number of system components.
  - The more complex a component, the more expensive it is to maintain.
- Software metrics
  - Measure of a piece of software
  - Lines of code, program size, number of objects, methods, etc.
  - cyclomatic complexity: number of execution paths through code
  - These metrics are used to determine complexity

9.3.2 Software reengineering

- Problem: Many older systems are difficult to understand and change.
  - May have been optimized for performance or space.
  - Structure may have been corrupted by series of changes
  - May have been poorly designed or commented
- Solution: Reengineering
  - Re-structuring or re-writing part or all of a software system without changing its functionality.
  - The system may be re-structured and re-documented to make it easier to maintain.

Software reengineering: Why not just rewrite from scratch?

- Reengineering takes less time
  - Developing a new system almost always takes longer than expected.
  - Re-developing a system involves duplicating work that has already been done for the existing system.
  - No matter how bad the old system is, it can probably be greatly improved in less time than starting over again from scratch.
- There is no guarantee the new system would be better.
- Joel on Software: Things you should never do

  http://www.joelonsoftware.com/articles/fog0000000069.html

Software reengineering techniques

- Regression Testing
  - To ensure modifications don’t change functionality.
- Source code translation
  - If it needs to be in a new language
  - Can be automated
- Reverse engineering
  - Analyzing source code to determine its design/structure
  - This does not change the code, produces documentation.
  - Can be automated

Software reengineering techniques

- Program restructuring
  - Restructure for understandability
  - Reorganize control structures and functions.
  - Can be automated, probably requires manual intervention
- Data reengineering
  - Clean-up and restructure system data.
  - Automated or manual

9.3.3 Preventative maintenance by refactoring

- Changing a software system: altering its internal structure without changing its external behavior
  - To improve readability.
  - To improve structure.
  - Reduce complexity.
  - Bottom line: easier to modify in the future
- No added functionality
- Preventative maintenance: reduces future maintenance costs

Refactoring versus Reengineering

- Both alter the code without altering functionality, with the purpose of making code more maintainable.
- Reengineering
  - Takes place after system is in use.
  - Applied when maintenance costs are too high.
  - Often involves automated tools on legacy code.
- Refactoring
  - Ongoing process, from start of development
  - Applied on smaller scale
  - Avoids structure degradation from the start

Where to apply refactoring (bad smells)

- Duplicate code
  - Same or very similar code found at different places in a program.
  - Extract method: put similar code into a single method/function
- Long method
  - Long methods are difficult to understand, modify.
  - Redesign as many shorter methods
- Switch (case) statements
  - Multiple switch statements with same cases.
  - Make subclasses, move each case into corresponding subclass.

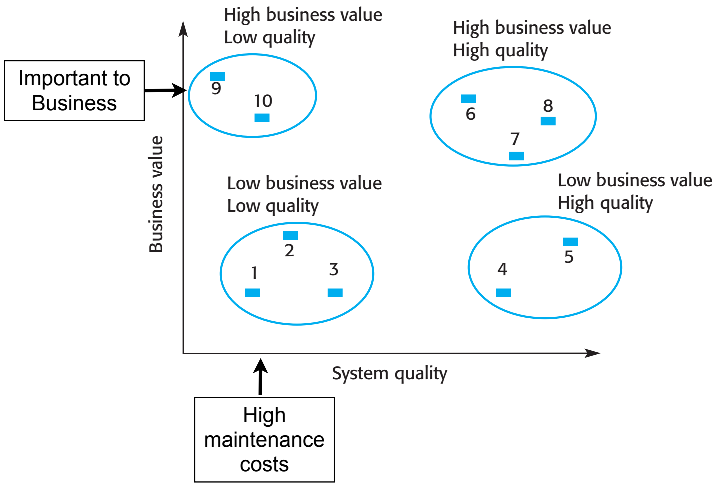
Where to apply refactoring (bad smells)

- Data clumping
  - When the same group of data items (fields in classes, parameters in methods) occur in several places in a program.
  - Replace with an object that encapsulates all of the data.
- Speculative generality
  - When developers include generality in a program in case it is required in the future (unused parameters, classes, unnecessary abstract classes).
  - This can often simply be removed

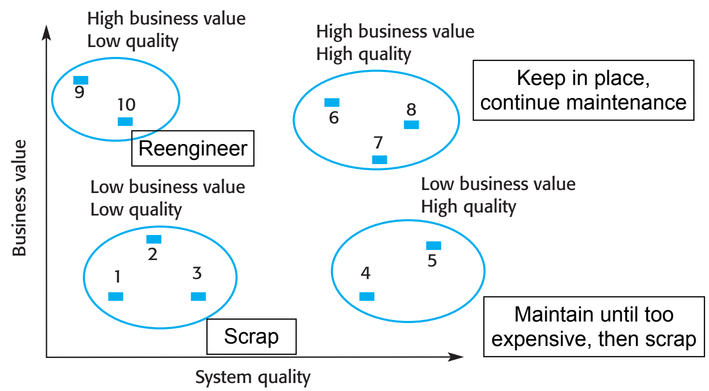
9.4 Legacy system management

- What is a legacy system?
  - System developed using obsolete technology or methods
- Strategies for evolving legacy systems
  - Scrap the system completely
  - Continue maintaining the system
  - Reengineer the system to improve its maintainability
  - Replace the system with a new system
- The strategy chosen should depend on:
  - the **system quality**
  - its **business value**

Example legacy system assessment



Example legacy system assessment



Reevaluate all the systems regularly (each year)

Refactoring example

Note: classes are incomplete: constructors, getters/setters are not shown.

```
class Employee
   double monthlySalary;
   double commission;
   double bonus;
   int getType() { … }
   int payAmount() {
      switch (getType()) {
         case ENGINEER:
            return monthlySalary;
         case SALESMAN:
            return monthlySalary + commission;
         case MANAGER:
            return monthlySalary + bonus;
         default:
            throw new RuntimeException("Incorrect Employee");
      }
   }
```


Refactoring example

Move cases into (new) subclasses

```
class Employee…
   double monthlySalary;
   double commission;
   double bonus;
   int payAmount();
}
class Engineer : Employee
   int payAmount() {
      return monthlySalary;
   }
class Salesman : Employee
   int payAmount() {
      return monthlySalary + commission;
   }
class Manager : Employee
   int payAmount() {
      return monthlySalary + bonus;
   }
```


Refactoring example

Push down field: when a field is used only by some subclasses

```
class Employee… {
   double monthlySalary;
   int payAmount();
}
class Engineer : Employee {
   int payAmount() {
      return monthlySalary;
   } }
class Salesman : Employee  {
   double commission;
   int payAmount() {
      return monthlySalary + commission;
   } }
class Manager : Employee  {
   double bonus;
   int payAmount() {
      return monthlySalary + bonus;
   }
}
```