Software Evolution

Chapter 9
(abridged)

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

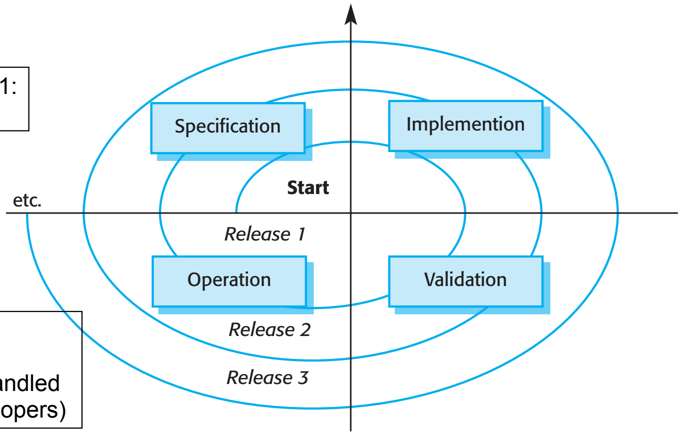
Software change

- 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 software engineering problem: managing change to existing software systems

Importance of evolution

- Software systems are 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



Before Release 1: Development

After Release 1: Maintenance (may NOT be handled by original developers)

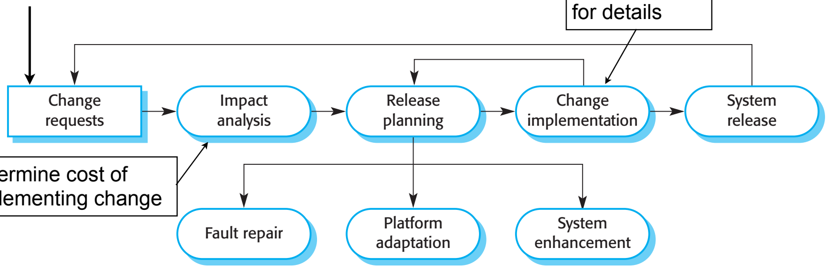
Bottom Line: All software processes become iterative development.

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.
  - requests for new features
  - bug reports
  - ideas for improvements

The software evolution process: overview

See next slide for details



Determine cost of implementing change

Bugs

Change in environment

New features

If evolution is handled by a team other than original development team: program understanding is an additional part of change implementation.

Change implementation:

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

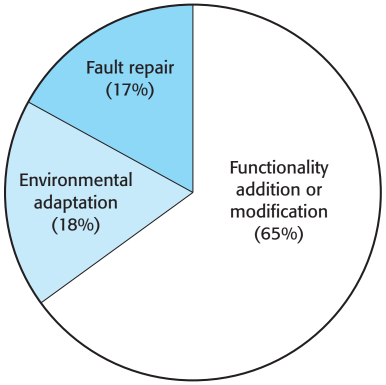
9.3 Software maintenance

- 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.
  - Otherwise it’s just iterative development
- Modifications may be simple or extensive
  - But NOT normally involving major changes to the system’s architecture.

Types of maintenance

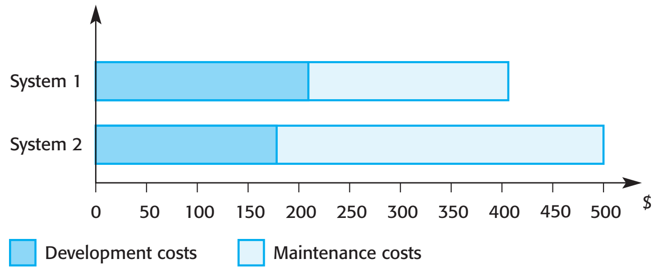
- Repairing software faults
  - Changing a system to correct coding, design, or requirements errors.
- Adapting software to a different operating environment
  - Changing a system so that it operates with a modified external system (e.g. new OS, or other software).
- Adding to or modifying the system’s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

Maintenance effort distribution



Development and maintenance costs

"A stitch in time saves nine"



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 even 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

- Estimating the overall maintenance costs for a system in a given time period (for planning purposes)
- 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, to determine complexity
  - Lines of code, program size, number of objects, methods, etc.
  - cyclomatic complexity: number of execution paths through code

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
- Reverse engineering
  - Analyzing source code to determine its design/structure
  - This does not change the code, produces documentation.
- Program restructuring
  - Reorganize control structures and functions for understandability
- Data reengineering
  - Clean-up and restructure system data.

9.3.3 Preventative maintenance by refactoring

- Refactoring is: 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 running 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 various 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 appropriate subclass.
- Data clumping
  - The same group of items occur in several places in a program.
  - Replace with an object that encapsulates all of the data (struct/obj)
- Speculative generality
  - Unused parameters, classes, etc, included “just in case”.
  - These can often simply be removed

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;
   }
}
```