**Chapter 9 – Software Evolution**

Lecture 1



**Topics covered**

1. Software changes always happen
2. Software is so vital - must be updated and changed
3. Spiral model of versions leads to evolution, servicing and phase-out
4. Key activity is 'change implementation'
5. Urgent changes - need to be done quickly
6. Agile methods well suited to evolution
7. Problems with handover of software between different teams



**Software change**

✧ Software change will always happen:

- New requirements emerge when the software is used;
- The business environment changes;
- Errors must be repaired;
- New computers and equipment is added to the system;
- The performance or reliability of the system may need improving.

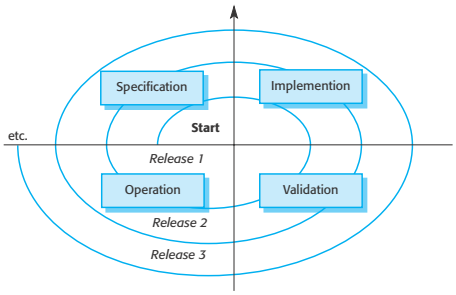
✧ **Question: How do businesses deal with these changes?**

## Importance of evolution

- ✧ Organizations have huge investments in their software systems - they are vital business assets.
- ✧ To maintain the value of these assets to the business, they must be changed and updated.
- ✧ The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.



## A spiral model of development and evolution

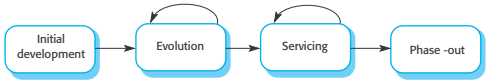





## Evolution and servicing

- ✧ Evolution
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- ✧ Servicing
  - At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and small changes. **No new functionality is added**.
- ✧ Phase-out
  - The software may still be used but no further changes are made to it.

## Evolution and servicing



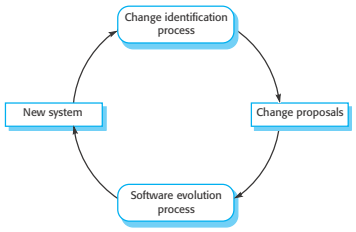



## Evolution processes

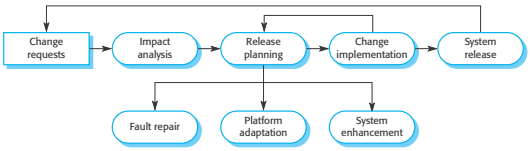
- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Ideas for change are the driver for system evolution.
  - Linked with components that are affected by the change, allows the cost and impact of the change to be estimated.
- Proposing changes and evolution **continues** throughout the system lifetime.



## Change identification and evolution processes



## The software evolution process






## Change implementation




## Change implementation

- Iteration of the development process where the changes to the system are designed, implemented and tested.
- An important difference is that the first stage of change implementation may involve **program understanding**, especially if the original programmers are not doing the changes.
- They need to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

## Urgent change requests

- Urgent changes may have to be implemented without going through all stages of the software engineering process
  - If a serious system fault has to be repaired to allow normal operation to continue;
  - If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
  - If there are business changes that require a very rapid response (e.g. the release of a competing product).
  - This is called **Emergency Repair**



## The emergency repair process




## Agile methods and evolution

- Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
- Automated regression testing is particularly valuable when changes are made to a system.
- Changes may be expressed as additional user stories.

**Two types of handover problems**

1. The development team have used an **agile** approach but the evolution team is unfamiliar with agile methods and prefer a **plan-based** approach.
   - The evolution team may expect detailed documentation to support evolution and this is **not** produced in agile processes.
2. Where a **plan-based** approach has been used for development but the evolution team prefer to use **agile** methods.
   - The evolution team may have to start from scratch developing **automated tests** and the code in the system may not have been **refactored** and simplified as is expected in agile development.



**Key points**

1. Software changes always happen
2. Software is so vital - must be updated and changed
3. Spiral model of versions leads to evolution, servicing and phase-out
4. Key activity is 'change implementation'
5. Urgent changes - need to be done quickly
6. Agile methods well suited to evolution
7. Problems with handover of software between different teams



**Chapter 9 – Software Evolution**

Lecture 2 - Maintenance

## Software maintenance

- Software maintenance - 3 levels of changes
- Levels of maintenance costs - depend on age, changes, staff etc
- Try to predict the changes may be required
- System re-engineering for older legacy systems
- Prevent maintenance by refactoring

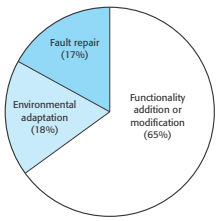


## Software maintenance

- Modifying a program after it has been put into use.
- The term is mostly used for changing custom software.
- Maintenance does not normally involve major changes to the system's architecture.
- Example changes include:
  - Correct coding errors (small change)
  - Correct design areas or new operating system (medium change)
  - New requirements (large change)



## Figure 9.8 Maintenance effort distribution

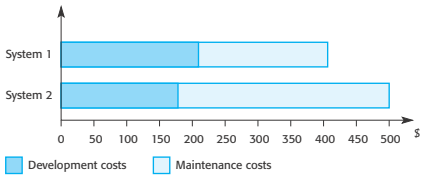


## Maintenance costs

- ✧ Usually greater than development costs (2* to 100* depending on the application).
- ✧ Affected by both technical and non-technical factors.
- ✧ Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- ✧ Ageing software can have high support costs (e.g. old languages, compilers etc.).



## Figure 9.9 Development and maintenance costs




So why is it more expensive to add functionality after a system is in operation?



## Maintenance cost factors

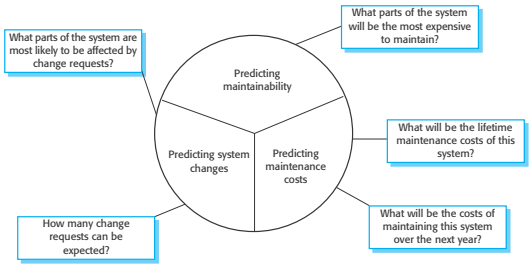
- ✧ Team stability
  - Maintenance costs are reduced if the same staff are involved with them for some time.
- ✧ Contractual responsibility
  - The original developers aren't paid to do maintenance, so there is no incentive to design for future change.
- ✧ Staff skills
  - Maintenance staff are often 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 prediction

- ✧ Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
- ✧ Managers hate surprises, especially if these result in unexpectedly high costs..
- ✧ Try to estimate the overall maintenance costs for a system in a given time period.



## Maintenance prediction






## Change prediction

- ✧ Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- ✧ Systems that rely other systems, require changes whenever the environment is changed.
- ✧ Factors influencing this relationship are
  - Number and complexity of system interfaces;
  - Number of system requirements that frequently change;
  - The business processes where the system is used.

## System re-engineering

- A **legacy system** is an older version of software that have had many changes
- Re-engineering aims to re-structure or re-write part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.



## Advantages of reengineering

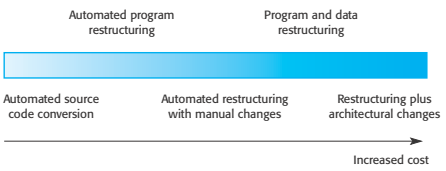
- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.



## Reengineering process activities

- Source code translation
  - Convert code to a new modern language.
- Reverse engineering
  - Analyse the program to understand it;
- Program structure improvement
  - Restructure to make it more understandable;
- Data re-engineering
  - Clean-up and restructure system data.

## Figure 9.12 Reengineering approaches






## Reengineering cost factors

- ✧ The quality of the software to be reengineered.
- ✧ The tool support available for reengineering.
- ✧ The extent of the data conversion which is required.
- ✧ The availability of expert staff for reengineering.
  - This can be a problem with old systems based on 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.
- ✧ When you refactor a program, you should not add functionality but rather concentrate on program improvement.

## Refactoring examples

- Duplicate code
  - The same or 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.
- Switch (case) statements
  - These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.



## Key points

- Software maintenance - 3 levels of changes
- Levels of maintenance costs - depend on age, changes, development team etc
- Try to predict the changes may be required
- System re-engineering for older legacy systems
- Prevent maintenance by refactoring