

Basic Principles of Software Evolution: Summary

Anthony Rouneau

May 24, 2016

1 Technical aspects of Software Evolution

Here are some technical aspects of the Software Evolution.

- Version management
- Software quality measurement and improvement
- Legacy systems and migration
- Reverse Engineering and program comprehension
- Model-driven evolution
- Change propagation and program comprehension
- Traceability
- Co-evolution
- Consistency maintenance
- Regression testing
- Design for change
- Visualisation and statistical analysis of evolution histories
- Software product lines and product families

Definition 1. *Reverse engineering* – Process of analysing a project to fully understand it and to extract the key issues. Mostly, the system is a large legacy system with a poor documentation and in a bad shape. Visualisation techniques can help.

Definition 2. *Re-engineering* – Horseshoe process (cf. figure 1) that targets to rethink a project in order to address one or multiple problems.



Figure 1: Re-engineering

1.1 Visualisation techniques

Visualisation techniques are useful to **understand** the way a software works, but also to understand the **complexity** of a software. There is five types of visualisation.

Code duplication The duplication of the code can be visualised as a dotplot, showing the similarity between multiple files. *CCFinderX*, *Duploc*

Dependencies The dependencies between the class or with external libraries can be shown of graphs, binding two dependant classes. *Stan*, *IntelliJ IDEA*, ...

Metrics The quality metrics can be visualised on multiple sort of views, 2D, 3D, ... *3D Treemap*, *Sunburst*, ...

Change The change of the software over time can be visualised, comparing the different versions. *Chronia*.

Quality The metrics can be summarized into one quality metrics that can be visualised. The managerial evolution can be observed over time. The complexity can be viewed to ease reverse engineering. *Code Crawler, X-Ray, ...*

- **Coarse-Grained visualisation** – Multiple metrics visible in one view. Can be used to reverse engineer.
- **Fine-Grained visualisation** – Class Blueprint views can be used to understand classes and class hierarchies, using a pattern language.
- **Evolution Matrix** – Using a pattern language, one can analyse the system evolution and/or the classes evolution.

2 Managerial aspects

- Traditional process model
- Evolutionary process model
 - Staged life-cycle
 - Iterative and incremental process
 - Agile methods
- Software configuration management
 - Change management
 - Version management
- Estimation techniques
 - Change impact analysis
 - Effort estimation
 - Cost estimation
 - Change metrics

2.1 Evolution process

The evolution of a software follows steps, defined in the chosen process model. The **urgent changes** can bypass such model to be available faster.

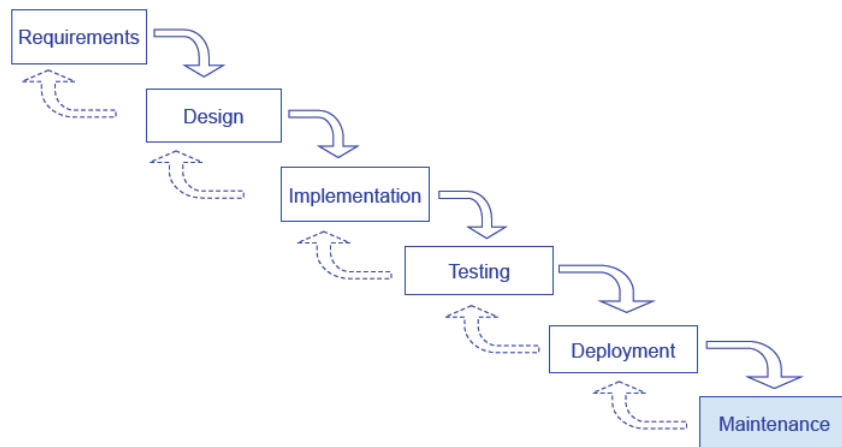


Figure 2: Traditional waterfall process model

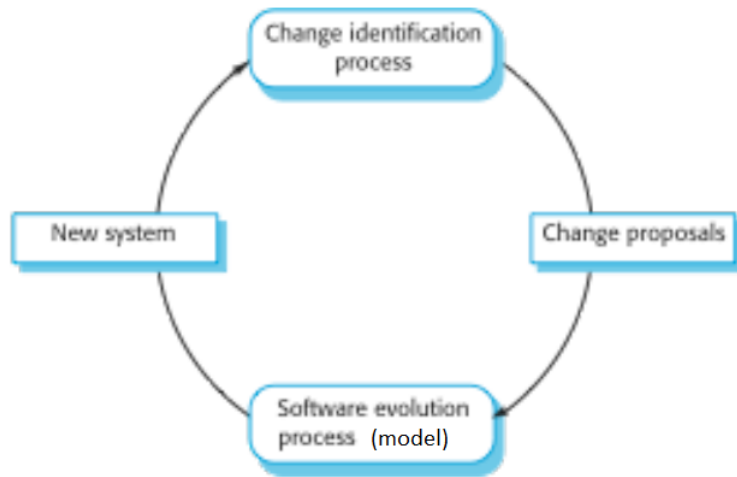


Figure 3: Evolutionary process

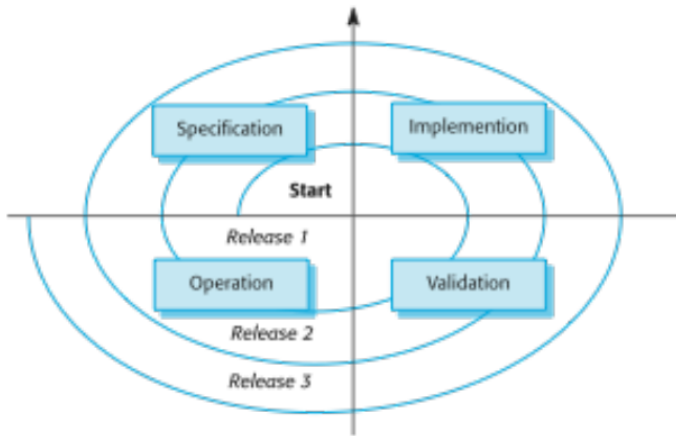


Figure 4: Evolution spiral model

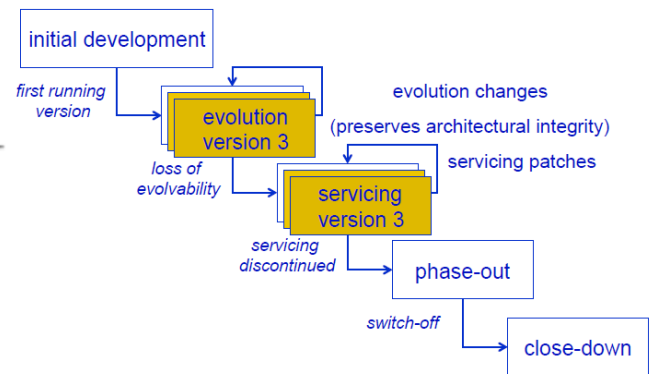


Figure 5: Evolution staged life model

2.2 Software configuration management

Changing a part of the software may affect other parts. Software configuration management consists of predicting the total number of changes required by an evolution and making the software the less subject to change by reducing coupling and dependences.

Definition 3. *Ripple effect* – The phenomenon where a change in one piece of a software system affects at least one other area of the same software system (either directly or indirectly).

Definition 4. *Change propagation* – Occurs when making a change to one part of a software system requires other system parts that depend on it to be changed as well. These dependent system parts can on their turn require changes in other system parts. In this way, a single change to one system part may lead to a propagation of changes to be made throughout the entire software system.

When a component is changed, visit every module that includes the component and check if it still fits. If a change is required, the same process is applied, taking the module as component.

3 Software Maintenance

The software maintenance is used to avoid **large** software to become legacy systems. In fact, adding new functionalities without considering code quality gives rise to **technical debt**.

Definition 5. *Technical debt* – *Lack of quality in the code due to modifications done in a hurry. The project is unclear while the debt has not been paid back through re-engineering. Measured in days; an estimation of time that would take to accomplish the pending tasks.*

Definition 6. *Software maintenance (1)* – *The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*

Definition 7. *Software maintenance (2)* – *The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity*

The software change is :

- **Unpredictable** – All the changes and bugs can not be anticipated in the design.
- **Expensive** – You generally don't get paid to solve bugs but to develop as fast as possible.
- **Difficult** – The errors are hard to find and the source code can be messy.

3.1 Legacy system

Characteristics :

- Original developers no longer available
- Outdated development methods used
- Extensive patches and modifications have been made
- Missing or outdated documentation

3.2 Parnas' ageing software

Symptoms :

- Lack of knowledge
 - Insufficient, inconsistent or obsolete documentation
 - Departure of original developers
 - Disappearance of inside knowledge about the system
 - Missing tests
- Code symptoms
 - Duplicated code, code smells, lack of modularity
 - Lack of overall structure or architecture
- Process symptoms
 - Constant need for bug fixes Too long time to fix bugs or to add new functionality
 - Difficult to separate functionalities

3.3 Necessity of software change

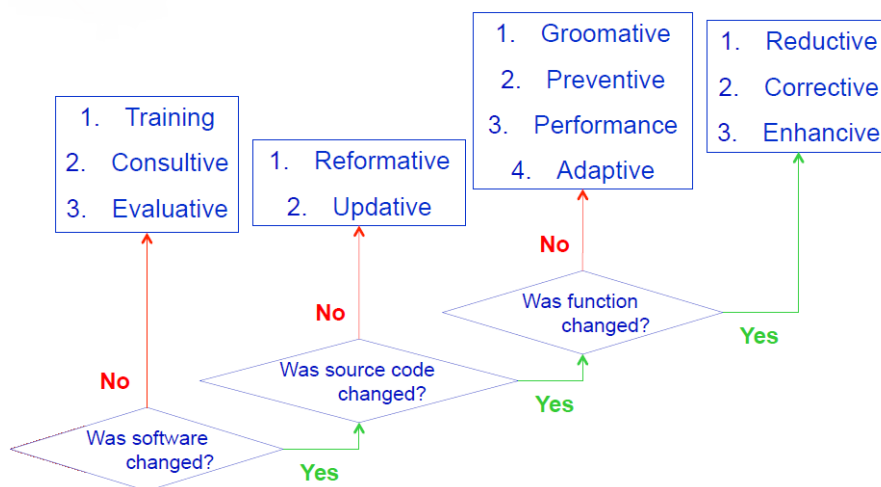
Software change is inevitable because:

- New requirements emerge when the software is used or developed.
- The business environment changes.
 - New customers
 - New demands
 - Organisational changes
 - Competitors
- Errors must be repaired
 - Bug fix routine
 - Emergency fix
- Hardware changed
- Improvements needed in efficiency
- New technologies have to be used
- Changes in data formats
 - New standards
 - New currency, ...

3.4 Types of software maintenance

- **Adaptive** – Adapt a software after delivery to support new technologies
- **Corrective** – Repair errors discovered after delivery
- **Perfective** – Add new functionalities to the software or improve it after delivery
 \implies *main reason*
- **Preventive** – Correct latent faults before they become effective ones.

<i>When? / Why?</i>	Correction	Enhancement
Proactive (before it happens)	Preventive	Perfective
Reactive (after it happened)	Corrective	Adaptive



3.5 Seven deadly sins

According to SonarQube, there are seven deadly technical debt that should be avoid as much as possible.

- Bugs and Potential Bugs
- Coding Standards Breach
- Duplications
- Lack of Unit Tests
- Bad Distribution of Complexity
- Spaghetti Design
 - Cyclic dependencies
 - High coupling
 - Low cohesion
- Not Enough or Too Many Comments

3.6 Clones

There are three types of clones

- **Type 1** – Same code, except for the line breaks, etc...
- **Type 2** – Same code, except for the variable/method names.
- **Type 3** – Same code modulo some changes/additions.