Basic Principles of Software Evolution: Summary

Anthony Rouneau

May 25, 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.

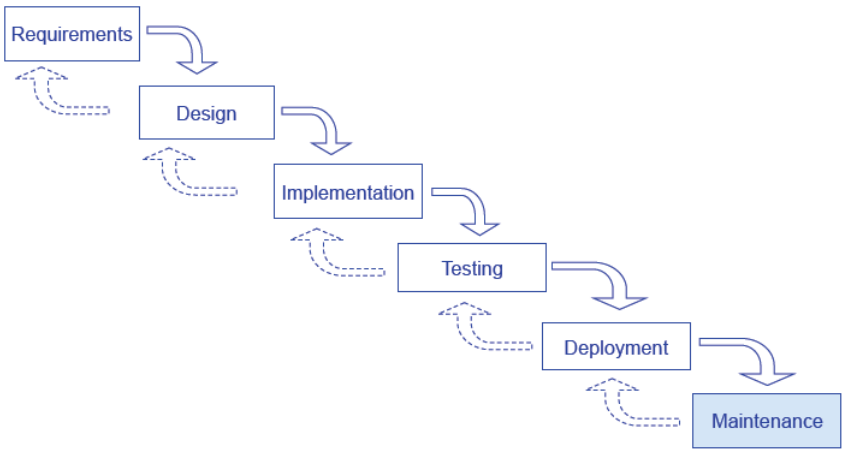


Figure 2: Traditional waterfall process model

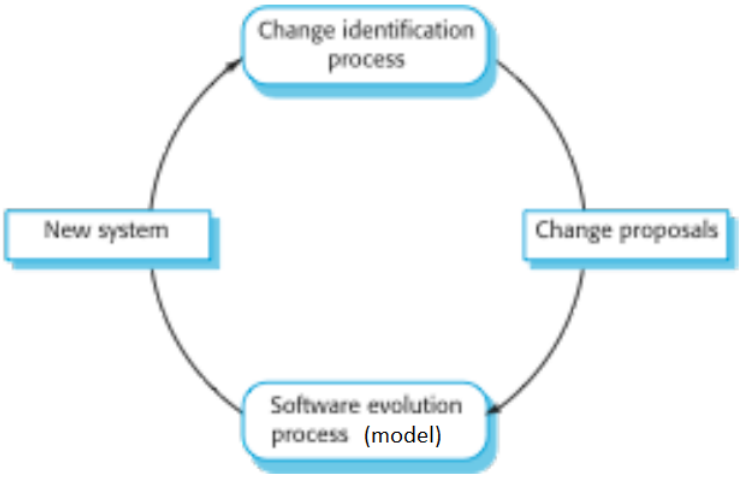


Figure 3: Evolutionary process

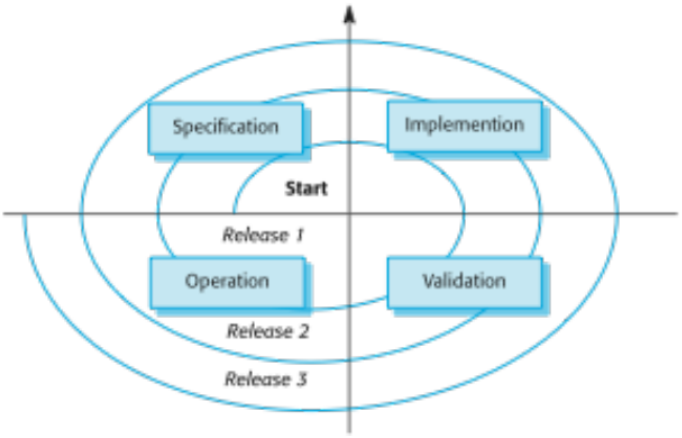


Figure 4: Evolution spiral model

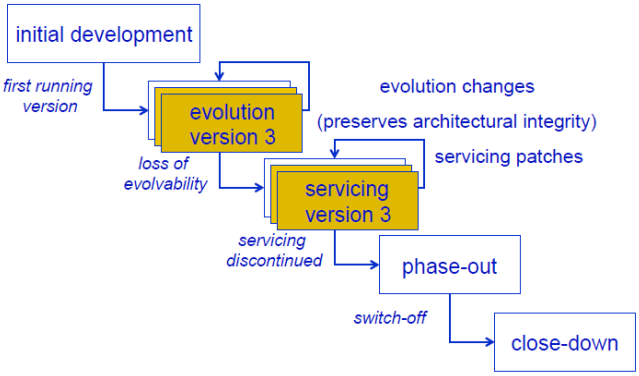


Figure 5: Evolution staged life model

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 unclean 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. Maintenance costs exceed development costs with at least a factor 2*

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

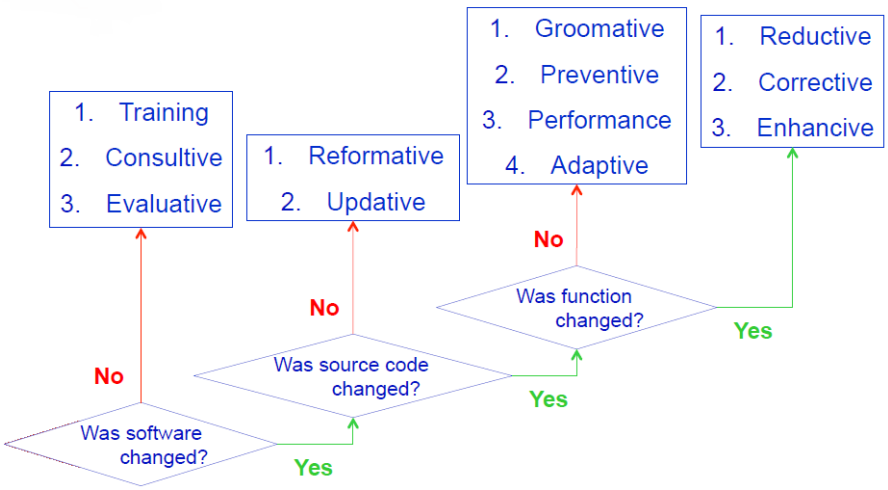
Software change is inevitable because:

- 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
  $\Longrightarrow$ *main reason*
- **Preventive** – Correct latent faults before they become effective ones.

When? / Why?	**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.

3.7 Bad Smells

Definition 8. ***Bad Smells*** *– Structures in the code that suggest (sometimes scream for) the possibility of refactoring.*

Definition 9. ***Anti-Pattern*** *– Identifies common mistakes and how these mistakes can be overcome in refactored solutions.*

Anti-Pattern example **The Blob** or **God Class**.

× General form:

- *Designs where one class monopolizes the processing,*
  and other classes primarily encapsulate data.
- Key problem is: majority of responsibilities are allocated to a single class.
- In general it is a procedural design ⟶ conflicts with OO paradigm.

× Refactored Solution:

- Identify or categorize related attributes and operations.
- Look for 'natural homes' for these collections of functionality.
- Apply OO Design techniques e.g., inheritance, ...

Bad smells examples

- **Long method** – Method that is too long → difficult to understand, change or extend.
- **Large class** – Class that is trying to do too much → too many instance variables/methods.
- **Long parameter list** – Too long method parameter list → difficult to understand.
- **Feature envy** – Method interested more in other class(es) → should moved.
- **Inappropriate intimacy** – Two classes are too tightly coupled with each other.
- **Lazy class** – Class that is not doing enough.
- **Data class** Class that contains data but almost no behaviour.
- **Duplicate code** – Redundant code, i.e., code that appears more than once.
- **Comments** Comments are misused to compensate for poorly structured code.

These bad smells can be detected using metrics:

- **Long method** use a combination of LOC (lines of code) and CC (Cyclomatic Complexity).
- **Long parameter lists** count number of method paramters.
- **Large class** count number of variables and methods.
- **Feature envy** use coupling metrics.
- **Inappropriate intimacy** use coupling metrics.

3.8 Refactoring

Definition 10. ***Refactoring*** *– A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality (simplicity, flexibility, understandability, ...).*

Refactorin methods example

- **Using a loop** instead of declaring all the statements.
- **Encapsulate public field** with getters and/or setters.
- **Extract behaviour** into another method or class.
- **Extract class/method/subclass**.
- **Move method** into another class.

Refactor categories

- **Creating template methods**
  - Cut methods in smaller pieces to separate the common behaviour from specialised one. → Use inheritance mechanism to redefine methods.
  - Can be used to improve reuse, to remove duplicated code, ...
- **Optimising class hierarchies**
  - Insert or remove classes in a hierarchy and redistribute functionality over these classes.
  - Can be used to increase cohesion, simplify interfaces, remove duplicated code.
- **Incorporating composition relationships**
  - Convert inheritance into aggregation when inheritance is overused or incorrectly used.

Size of refactors

- **Small**
  - Short time to proceed.
  - Instant satisfaction.
- **Big**
  - Long time ($\sim$ 1 month).
  - Agreement with the team needed.
  - No instant satisfaction and no visible progress.

Utility of refactoring Apart from making the code simpler and have better quality metrics, refactoring can be used to generate **change metrics.** The structures of the well-known refactors can be detected between two version of a software. Nonetheless, Refactoing can lead to **conflicts** when applied to frameworks used in some little software. Those software must change along with the framework, which is not easy.

Model refactoring Refactors can be applied to diagrams (UML, ...)

Definition 11. ***Model Driven Engineering*** *– Develop application using models only. Everything can be considered as model (source code, syntax, ...). Implies a higher level of abstraction and allows anybody that know the syntax of the model to understand the software.*

Multiple refactors can be applied to UML diagrams:

- **Class diagrams** – Similar to method and class classic refactors, move methods, push up, pull down, ...
- **State machines** – Merge states, split states, create composite state, flatten states, ...
- **Activity diagrams** – Paralleling independent activities, concurrent state serialisation.

4 Product Line Engineering

Definition 12. ***Software Product Line*** *– Set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

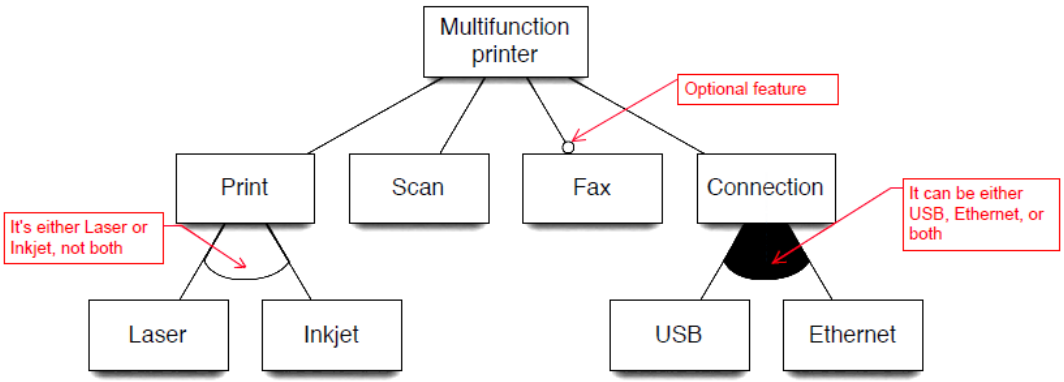


Figure 6: Feature diagram

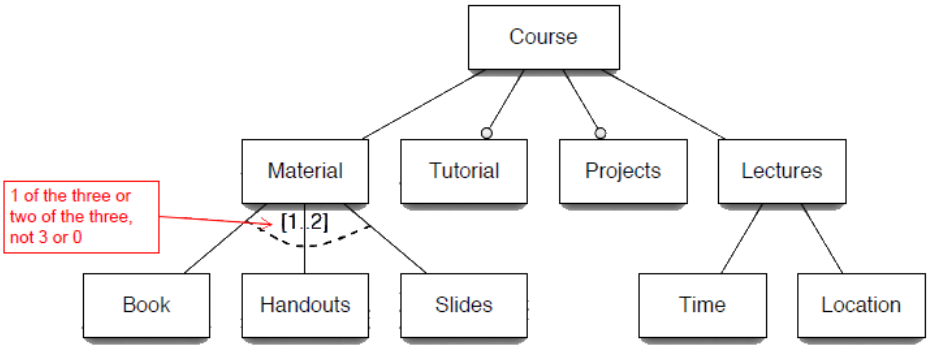


Figure 7: Feature diagram

- **Commonality** – List of the features that appear in all products.
- **Variability** – List of the features that appear not in all products.
- **Dead feature** – Feature present in the diagram that appears in no product.

5 Data extraction

One could want to extract data from numerous database concerning an Open Source project. The main challenges to do so are:

- **Identity Merging** – A single contributor can have his name/nickname spelled different ways for different sites (inverse first and last name, "accent" missing, punctuation, middle names, nickname for a specific project, ...). One solution is to compute the distance between two names (Levenstein distance for Strings) and merge the similar names. Manual post-process is necessary.
- **Workload analysis** – It may be useful to analyse the main topic on which each contributor has worked. The commits on the git repository are analysed after the files of the project have been labelled with their respective topic. The $\widetilde{T}$ procedure allows to identify the main topic and the secondary topics of the project.
- **Community analysis** – If the developers focus on a single task, it is likely that the project (large) is heterogeneous, because a large project requires a lot of different qualification to release (graphical, literal, code dev, ...).