# Towards a Software Evolution Benchmark

Serge Demeyer
Universiteit Antwerpen
Universiteitsplein 1
B-2610 WILRIJK
serge.demeyer@uia.ua.ac.be

Tom Mens[*]
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 BRUSSEL
tom.mens@vub.ac.be

Michel Wermelinger
Departamento de Informática
Universidade Nova de Lisboa
P-2829-516 CAPARICA
mw@di.fct.unl.pt

## ABSTRACT

Case-studies are extremely popular in rapidly evolving research disciplines such as software engineering because they allow for a quick but fair assessment of new techniques. Unfortunately, a proper experimental set-up is rarely the case: all too often case-studies are based on a single small toy-example chosen to favour the technique under study. Such lack of scientific rigor prevents fair evaluation and has serious consequences for the credibility of our field. In this paper, we propose to use a representative set of cases as benchmarks for comparing various techniques dealing with software evolution. We hope that this proposal will launch a consensus building process that eventually must lead to a scientifically sound validation method for researchers investigating reverse- and re-engineering techniques.

## Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement.

## General Terms

Measurement, Experimentation, Standardization.

## Keywords

software evolution, empirical survey, case studies, benchmark.

## 1. INTRODUCTION

Software engineering is in a constant state of flux. On top of the rapid escalation of hardware we keep on enlarging our repertoire of techniques to build (and maintain) software. As a consequence, developers that want to improve their software production process must choose from a wide range of techniques and accompanying tools. However, adopting a new technique or tool always involves a cost, hence before launching into "yet another solution" rational decision makers demand evidence for the claimed benefits.

Empirical surveys are the traditional scientific means to provide such evidence [1]. Indeed, if the experiment is continued for a long enough time, reliable claims can be made about the cost/benefit trade-off of a certain technique. Unfortunately, such experiments take a lot of time and effort, hence can only be applied for techniques that have proven to be mature.

Case-studies represent a more lightweight opportunity for providing scientific evidence [2]. Not only does a case-study illustrate applicability on a concrete project, it also provides the opportunity to learn about new techniques and —most importantly— it can be performed in a reasonable amount of time. Therefore, case-studies are ideally suited for investigating novel techniques which yet have to prove their value. However, to serve as a basis for rational decision making, a case-study should satisfy two criteria: (a) be *representative*, i.e., the results obtained from the study can be generalised to other projects in a well-defined category; and (b) be *replicable*, i.e., all artefacts necessary to perform the study are accessible so that the results can be confirmed or falsified by others.

With the appearance of the world-wide web and the trend towards open-source projects, a wide range of realistic cases satisfying both criteria is now available. This has the unpleasant side-effect that the interpretation of case-study results is hard because it is difficult to compare the wide variety of results. One way to circumvent this problem is to agree within a scientific community on a representative set of cases that together may serve as benchmarks for the problems being investigated. Building consensus on such benchmarks is a process which requires several iterations and revisions within the scientific communities however. This paper makes a first step in that direction and as such we hope to launch a consensus building process that eventually must lead to a scientifically sound validation method.

The paper proposes benchmarks for comparing techniques to deal with *software evolution.* We expect such a benchmark to be used to validate three kinds of techniques: (a) predictive analysis (verify whether a technique may predict certain kinds of evolution based on the current state of the system); (b) curative activity (verify whether a technique supports a given software evolution process, e.g. refactoring); (c) retrospective study (verify whether a technique can reconstruct how and why a software system has evolved in the past). To that effect, we specify the design space of evolving software systems by listing the characteristic attributes of these systems (section 3). Afterwards, we present 4 cases as representatives for part of the design space (sections 4-5). Complementary to the cases, we propose a list of attributes to classify the merits of the techniques and tools themselves (section 6). We conclude the paper with an explicit position statement phrased as a number of questions regarding the potential value of the benchmarks (section 7).

[*] Tom Mens is a postoctoral fellow of the Fund for Scientific research – Flanders (Belgium).

## 2. CONTEXT

In January 2001, we started a research network, which has the ultimate goal "to develop a consistent set of formal techniques and accompanying tools that will support software-developers with typical evolution problems of large and complex software systems"[1]. This goal was chosen because numerous studies have shown that more than half of the global budget of software development is spent during maintenance. Hence the need for better methods, techniques and tools.

The network consists of a wide range of international research groups investigating techniques that are somehow related to evolving software. To give an idea of the diversity within the group, here is the list of expertise areas from the various partners: (a) declarative reasoning (mostly logic-based); (b) (graph-) rewriting systems; (c) software metrics; (d) software visualisation techniques; (e) analysis and design methods; (f) migration to component-based and web-based systems; (g) meta modelling approaches; (h) reverse engineering; (i) code generation.

Facing this diversity, the group concluded that there was a definite need for benchmarks that would allow them to compare the various techniques. Since all of the groups were particularly concerned with the object-oriented paradigm, we agreed that this should be the common theme driving the benchmarks.

## 3. EVOLUTION CHARACTERISTICS

In order to specify the design space of evolving software systems, we define a list of characteristic attributes. With such a list, we can later assess whether the cases may serve as representatives for the complete design space.

**Life-cycle characteristics** determine whether the case comprises artefacts that correspond to all phases in the software life-cycle.

- Analysis (requirements specification, domain models, user interviews, mock-ups, CRC cards, use cases, ...)
- Design (architecture, detailed design, formal specifications)
- Implementation (source code)
- Testing (test plans, test code, test results)
- Maintenance (bug reports, feature requests, version control, configuration management)

**Evolution characteristics** assess the evolution process of the case, i.e., the various iterations and increments that one can identify in the development process.

- Number of iterations
- Total time of the evolution process
- Scale (size of each iteration in lines of code, number of classes, number of use cases)
- Type of iteration (refactoring, extension, correction, adaptation)
- Granularity of increments in terms of time (days, weeks, months), size (lines of code, pages of documentation) or components (methods, classes, modules)
- Staff (how many persons were involved, personnel turnover between iterations, level of experience of the developers, ...)

[1] See http://prog.vub.ac.be/FFSE/network.html

**Domain characteristics** qualify the domains of the case.

- Application domain (e.g., telecommunication, e-commerce, desktop systems)
- Problem domain (e.g., graphical user interfaces, distributed systems, web-based systems, embedded systems, real-time systems)
- Solution domain (e.g., library, framework, components, program)

**Tool characteristics** evaluate the kind of tools that are necessary to *replicate* the case. As such, the tool characteristics are not necessary to assess whether a case may serve as a representative.

- Implementation language(s) (C++, Java, Smalltalk, Object Pascal (Delphi), Ada, Eiffel, ...)
- Analysis and design language or notations (UML, OMT, EROOS, Z, VDM, statecharts, ...)
- Operating system (Unix, Linux, Windows, MacOS, ...)
- Integrated development and CASE environments
- Special libraries (CORBA, ...)
- Extra utilities (merge tools, version control tools, ...)

## 4. CANDIDATE CASES

Now that we have defined the characteristic attributes of evolving software systems, we list possible cases that may serve as representative for a certain category of evolving software systems.

Material (source code, documentation) concerning the listed cases is available at http://prog.vub.ac.be/FFSE/cases/.

### 4.1 Toy examples

Toy examples are limited in scope and as such provide the ideal means to make an initial assessment of a technique. Of course they cannot be used as a means to assess the scalability of a technique, the other categories must serve this purpose.

- **LAN Simulation.** Both the Software Composition Group of the University of Berne and the Programming Technology Lab of the Vrije Universiteit Brussel make use of a small simulation of a LAN network to illustrate and teach good object-oriented design. The simulation starts with a simplistic model that gets refactored as requirements are added.

Besides the LAN simulation, there are other cases that may serve as a representative for the small-scale software systems. To ease comparison, we insist that people stick to the LAN simulation. Yet, to encourage research-in-the small, we list some other options as well.

- **TicTacToe.** In his course "P2 : Programmierung 2", Prof. Oscar Nierstrasz guides the students trough a series of steps to incrementally develop a game (TicTacToe) which is later incrementally extended into a framework for board games (incl. distributed playing and GUI). All increments are written in Java. A scaled-down version of the TicTacToe game is also available in Oberon.
- **Conduits Framework.** In yet another introductory course on object-oriented programming (Principles of Object-Oriented Languages), Prof. Theo D'Hondt has developed the Conduits framework, which is a framework to simulate flows of fluids in pipes. It is available in both Java and Smalltalk.

## 4.2 Industrial Systems

In general, it is hard to find good industrial cases as these typically involve non-disclosure agreements, hence go against the criterion of replicability of experiments. Yet, industrial cases are necessary to obtain the necessary credibility.

- **VisualWorks/Smalltalk.** Smalltalk has a long tradition of shipping the source code along with its products. A quite impressive one is the GUI-builder that comes with the VisualWorks programming environment and which is freely available for academic purposes.
- **Swing.** Java as well has its platform independent GUI-builder named Swing. The subsequent releases of the Swing framework mark a smooth evolution process.

## 4.3 Public Domain Software

As an alternative for industrial cases, we consider software created for the public domain.

- **HotDraw (DrawLets).** HotDraw is a two-dimensional graphics framework for structured drawing editors (i.e. from CASE tools to PERT chart editors). It started off as a Smalltalk framework, but was later redone in Java (under the name of Drawlets).
- **JUN.** An impressive framework for implemention of fast 3D graphics and multimedia applications. Available in Smalltalk and Java.

## 4.4 Open-source projects

The current wave of open-source software development provides many evolving software projects. Within the scope of this paper we restrict ourselves to a few examples.

- **Mozilla.** An open-source web browser, designed for standards compliance, performance and portability and partly serving as a basis for the Netscape browser.
- **Squeak.** An open source variant of Smalltalk, based on the Smalltalk-80 virtual machine written entirely in Smalltalk and with superb graphic features.
- **Jikes.** An open source compiler for Java written in C++.

## 5. SELECTED CASES

Table 1 sets out the selected cases against the specified characteristics. This way we can assess which part of the design space is covered by the cases and which part is still left open, hence should be covered by other cases.

From the table we can infer that the candidate cases fall a bit short. The early life-cycle phases are sparsely covered because there is little analysis or design documentation available. Also, the implementations are limited to Java, C++ and Smalltalk systems, thus no Ada nor Eiffel. Finally, some important application domains (embedded and distributed systems) are not yet covered.

Moreover, we lack some quantitative data concerning the various cases (i.e., number of iterations and staff). Also, the replicability of the cases is questionable since for some of them only the last release is publicly accessible. It is feasible to collect the missing data, but before committing to these efforts we want to see whether a better suite of cases can be identified.

## 6. TECHNIQUES AND TOOLS

To complement the characteristics for evolving software systems (section 3) which lead to the selection of the cases (sections 4-5), this section proposes a list of attributes that may be used to classify the merits and applicability of the evolution support techniques and tools themselves.

**Table 1 Overview of the selected cases and their characteristics**

| | **Toy Example** | **Industrial System** | **Public Domain** | **Open-source project** |
|---|---|---|---|---|
| | LAN -Simulation | VisualWorks & Swing | HotDraw & JUN | Mozilla |
| **Life-cycle** | | | | |
| • Analysis | No | No | Yes (CRC Cards) | No |
| • Design | Yes (UML) | Partly | Yes | Yes |
| • Implementation | Smalltalk, Java | Smalltalk, Java | Smalltalk, Java, C++ | C, C++, Java |
| • Testing | No | No | No | Yes |
| • Maintenance | No | Partly | Partly | Yes |
| **Evolution** | | | | |
| • Scale | Tiny (< 20 classes) | Medium | Small | Very large |
| • Type | Refactoring, Extension | All types | All types | All types |
| • Granularity | Fine (refactoring) | Coarse | Coarse | Fine + Large |
| • Staff | 1 | | 2 | |
| **Domain** | | | | |
| • Application domain | networks | tool | tool | network |
| • Problem domain | simulation | GUI | graphics | web-based systems |
| • Solution domain | program | black-box framework | white-box framework | family of programs |

### 6.1 Time of evolution support

**(a) Predictive** (= before evolution). This category covers all techniques and tools that allow maintainers to make decisions concerning the parts of the software that should be improved.

- *Evolution-critical*: identify those parts of the software that need to be evolved due to a lack of quality (e.g., quality metrics)
- *Evolution-prone*: assess which parts are likely to evolve in the future (e.g., by visualising the number of changes)
- *Evolution-sensitive*: distinguish those parts of the software that will suffer from evolution (e.g., impact analysis)

**(b) Curative** (= during evolution). This category concerns techniques and tools that support the actual changes to the software system.

- *Passive*: infrastructure that allows to keep track of the changes (e.g., version control systems)
- *Active*: techniques and tools that allow to apply changes (e.g., refactoring tools, merge tools)

**(c) Retrospective** (= after evolution). This category includes techniques and tools that allow to analyse where, how and why a software system has evolved in the past.

- *state-based*: techniques and tools that compare the intermediate stages (e.g., the UNIX facility `diff`)
- *change-based*: techniques and tools that analyse the changes (e.g., inspecting the change log)

### 6.2 Degree of Automation

**(a) Semi-automatic tools** still require some human assistance. This can be to interpret the results (e.g., software visualisation), or to provide additional information that cannot be derived automatically (e.g. complex refactorings).

**(b) Fully-automated tools** do not require any human intervention (e.g., parsers).

## 7. OPEN QUESTIONS

In this paper, we propose to use benchmarks for evaluating techniques dealing with evolving software. The benchmarks correspondt with a number of cases each representing different kinds of evolving software artefacts; differences being captured in a set of characteristics. To complement the cases, this paper also proposes a list of attributes that may be used to classify the merits of evolution support techniques and tools themselves.

Of course the definition of benchmarks is only a first step towards sound scientific research in the area of software evolution. We invite the workshop participants to help us by answering the following questions

- **Does it makes sense to define benchmarks** ? The purpose of this paper is that several research groups each apply their favourite technique on at least one of the cases. As such we can compare the various techniques and see how they may complement or overlap each other. We feel that such benchmarks are an ideal vehicle for exchanging information and experience concerning evolving software. Nevertheless, to complete specification of such benchmarks still requires a lot of work. *Hence we ask the workshop participants if they would consider to validate their work with such benchmarks?*
- **Are the characteristics complete / minimal** ? The list of characteristics has been designed as an instrument for selecting a set of representative cases. However, care should be taken whether the instrument is accurate. In particular, we should address the question whether the list of characteristics is complete, because if it is not then we risk that the selected cases are not representative. Equally important is the question whether the list of characteristics is minimal, because if it is not then we risk that we must select too many cases to cover all possibilities. Our initial experience with the case selection suggests that the list is not minimal but reasonably complete. *However, we explicitly ask the workshop participants to propose improvements or –even better– point out other attempts concerning software evolution benchmarks.*
- **Are the cases representative** ? The cases are meant to represent the whole design-space of evolving software systems. However, the current cases fall a bit short: they are weak in the early life-cycle phases (little analysis or design documentation is available); they only include object-oriented implementations limited to Java, C++ and Smalltalk systems (no Ada or Eiffel, ...); they cover few application domains (no embedded systems or distributed systems). *Therefore, we explicitly ask the workshop participants to point out other industrial or semi-commercial cases that will provide better coverage of all the characteristics.*
- **Are the cases replicable** ? For the given cases, the source code and documentation is at least available on the web. However, for most of them only the latest release is directly accessible: for earlier releases one should contact the original developers.

  We intend to collect all material (including better quantitative data on the size and the granularity of the changes) and acquire the necessary permissions to use it for experimentation purposes. *However, we first want to ask the workshop participants what kind of information they need to replicate a case study.*

## 8. ACKNOWLEDGEMENTS

This research is performed inside an International Research Network on *Foundations of Software Evolution*, financed by the Fund for Scientific Research - Flanders (Belgium).

## 9. REFERENCES

[1] Fenton, N., S. L. Pfleeger and R.L. Glass "Science and Substance: a Challenge to Software Engineers". IEEE Software 11(4), July 1994.

[2] Fenton, N. and S. L. Pfleeger, *Software Metrics: A Rigourous and Practical Approach*, International Thomson Computer Press, 1997.