Some Issues in the 'Archaeology' of Software Evolution

Michel Wermelinger and Yijun Yu

Computing Department & Centre for Research in Computing
The Open University, UK

Abstract. During a software project's lifetime, the software goes through many changes, as components are added, removed and modified to fix bugs and add new features. This paper is intended as a lightweight introduction to some of the issues arising from an 'archaeological' investigation of software evolution. We use our own work to look at some of the challenges faced, techniques used, findings obtained, and lessons learnt when measuring and visualising the historical changes that happen during the evolution of software.

1 Introduction

Wikipedia defines archaeology as "the science that studies human cultures through the recovery, documentation, analysis, and interpretation of material culture and environmental data." Software archaeology is similar: it studies the human activities during the lifetime of a software project through the artefacts it generates, such as code and design documents. As important are those intermediate items that often get lost after software delivery, in the form of emails, memos, tickets, drafts, comments or logs. Being the materials for archaeological study, these artefacts form important trails for others to reconstruct the history of development [7, 15]. Their history is the fundamental memory for developers to maintain the software. They are also a valuable source for similar projects, showing good practice to be reused and pitfalls to be avoided.

All these artefacts are, ideally, kept in a readily interpretable and persistent form with the help of the 'digitalised memory' of the project life: version controlled repositories, archived emails, reported bug records, etc. However, just sieving through these data already imposes great challenges to software archaeologists, not to mention the additional difficulties in interpreting them. There are at least three types of transformations involved.

– 'Horizontal' or evolutionary transformations are conceptual units of work that lead from one version of the software to the next. These high-level transformations are often 'mediated' or triggered by other artefacts (e.g. bug reports) but the archaeological evidence of these transformations is often just fine-grained changes (e.g. lines of code added). As software processes are also programs [20], evolutionary transformations become amenable to analysis.

– ‘Vertical’ or generative transformations are responsible for producing a lower-level artefact from a higher-level one. An example is the transformation from requirements to code via design documents. Code generation [8], meta-programming [10], and model-driven development [23], among others, are approaches that allow to specify and automate some of these transformations.
– Archaeological transformations involve the extraction and processing of data from available artefacts in order to turn such artefacts into models, measurements, visualizations and documentation that can help the archaeologist interpret the project’s history.

Due to the different kinds of transformations involved, some common challenges faced when recovering and reconstructing the past are:

1. **Abstraction.** The sheer amount of data can easily overload a person [25]. The key is to use abstractions that serve the purpose at hand. One has to be careful though, as transforming the original data into its abstract form (e.g. a model or a metric) might lose subtle but important details.
2. **Lost artefacts.** Some artefacts are hardly documented, e.g. the original goals and motivations, assumptions, tacit knowledge, design rationale and principles [32]. However, they are what the archaeologists would like to infer from other related artefacts, at the risk of deriving wrong or biased information.
3. **Automation.** The archaeologist needs an assistant to perform the mundane work of data collection and transformation, otherwise they may not be able to understand the overall software system, because the history of artefacts is complex and expensive to find out manually. The assistant is ideally a robot that can perform these transformation tasks automatically. However, fully automated recovery is not always achievable depending on the source and target of the transformations [14].
4. **Evaluation.** Any analysis performed by one archaeologist should make sense to another archaeologist, but it may be more valuable if the evaluation is performed by the ‘witnesses’ (the original developers or users) or by alternative sources. Automated tools for measuring and visualising the software artefacts can be of a great help to make the study repeatable. The use of standard formats such as Rigi [29] to record the results would also help reduce the barrier for others to evaluate them.

To reconstruct a vivid history of software development, we aim to understand not only the deliverable artefacts such as the software itself, but also the tacit knowledge reflected by the communications and the coordination amongst stakeholders of the software. Recovering higher-level transformations from finer-grained changes would help the archaeologists reconstruct a model to sufficiently represent the evolution. Generative and archaeological transformations are not always in the form of round-trip to maintain the equivalence between the source and the target. Certain properties in the source can be preserved in the target, often with additional information at a lower level of abstraction. The reality is, however, that the information to recover transformations is not always available.

When transformations are applied in archaeology, one must be prepared for the loss of information, sometimes important one. As it is often difficult to obtain the original sources when studying the target of these transformations, one must be careful not to put too much trust into the data sets. Only when reliable information is hard to obtain, one should rely solely on the data rather than the people. Open source projects [12] often make the life of archaeologists much easier as they make available not only the targets but also the sources and mediators of evolutionary and generative transformations, including the code, the email archives and the bug reports, etc.

In the next three sections we present a part of our own archaeological work on analysing architectural evolution and discuss in Section 5 some general issues and lessons arising from it. While our previous papers [26, 27] focussed on the technical results (i.e. the outcome of the archaeological process), this one emphasizes the means to obtain them (i.e. the archaeological process itself) and the research path decisions taken. As such, Sections 1, 5 and part of 2 are new, and Sections 3.2 and 3.3 are the result of extensively rewriting, updating and expanding material that was fragmented across several papers [26, 27, 33], adding many more details about the data model and infrastructure used to assess the past history. Nevertheless, we also updated the results (Section 4), adding new data about the more recent releases of the chosen case study, and introducing a distinction between forced and unforced changes, which in turn led to several changes in the data visualization approach.

The paper, like the summer school that originated it, is aimed mainly at postgraduate students. By narrating our experience, and not just the end results, we aim to give those wishing to enter this research area a glimpse of the 'backstage' events of software archaeology. Interested readers are encouraged to afterwards consult more detailed treatments of this subject [19, 17, 24, 7, 15].

2 Motivation

Our research on architectural evolution started in a rather opportunistic way, when we came across the call for papers for the challenge track of the 5th Working Conference on Mining Software Repositories[1]. The challenge was to mine the Eclipse project, an open source integrated development environment (IDE) with respect to 1) bug analysis, 2) change analysis, 3) architecture and design, 4) process analysis or 5) team structure. The call for papers provided several CVS repositories with subsets of the Eclipse project, but authors could choose any other data source.

Given our past interest in software evolution and software architecture, and knowing that Eclipse had a strong IBM lead, we decided to attempt an analysis of the architectural change process, thereby addressing topics 2), 3) and 4) of

[1] http://msr.uwaterloo.ca/msr2008/challenge/

the five proposed[2]. To be more precise, the research questions that we had in mind were:

1. Is there any systematic architectural change process, or is the architecture being continually modified in every release?
2. Does the architectural evolution follow any of Lehman's software evolution laws, like continuous growth and increased complexity?
3. Is there any evidence of restructuring work aimed at reducing growth and complexity?
4. Is there any stable (i.e. unchanged) architectural core around which the system grew?

Once we saw the Eclipse project contained data to enable the archaeological investigation of those questions [26], the next step was to widen the scope of the research questions, looking at whether Eclipse's architecture was following design guidelines that have been proposed to ease changes, like absence of cyclic dependencies, low coupling and Martin's stable dependency principle [18]. The detailed motivation, research questions and results of those investigations were presented in [27]. Here, we only revisit one of the questions:

5. Does cohesion increase and coupling decrease over time?

The main point to keep in mind is that, while the research was *triggered* by a given case study, it was *led* by general research questions about architectural process and design principles. The overarching motivation was to *invalidate* such principles, in the spirit of falsifiability of scientific hypotheses [21]: if a highly successful and continuously evolving infrastructure project like Eclipse, on which many third-party components and applications have been built, does *not* follow commonly recommended guidelines, the usefulness (or at least the importance) of such guidelines could be questioned.

3 Data Collection

After having explained our motivation and particular architectural research angle, we can look more closely at the case study, which data was extracted, and how.

3.1 The Case Study

The case study consists of multiple *builds*, i.e. snapshots, of the Eclipse Software Development Kit (SDK) source code. Each build is implemented by a set of *plugins*, Eclipse's components. Each plugin may *depend* for its compilation on Java classes that belong to other plugins. For example, the implementation of plugin platform (we omit the default org.eclipse prefix) in 3.3.1.1 depends on eight

[2] We later also looked briefly into the team structure of Eclipse and how it changed over time [28], but will not go further into it for this paper.

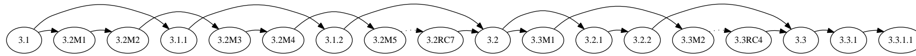

Fig. 1. Chronological and logical sequences of some of the analysed builds

other plugins, including core.runtime and ui. Each plugin *provides* zero or more *extension points*. These can be *required* at run-time by other plugins in order to extend the functionality of Eclipse. A typical example are the extension points provided by the ui plugin: they allow other plugins to add at run-time new GUI elements (menu bars, buttons, etc.). It is also possible for a plugin to use the extension points provided by itself. Again, the ui plugin is an example thereof: it uses its own extension points to add the default menus and buttons to Eclipse's GUI.

In the remaining of the paper, we say that plugin X *statically depends* on plugin Y if the compilation of X requires Y, and we say that X *dynamically depends* on Y if X uses at run-time an extension point that Y provides. Note that the dynamic dependencies are at the architectural level; they do not capture run-time calls between objects.

For our purposes, the architectural evolution of Eclipse corresponds to the creation and deletion of plugins and their dependencies over several builds. There are various types of builds in the Eclipse project. We analysed *major and minor releases* (e.g. 2.0 or 2.1) and the *service releases* that follow them (e.g. 2.0.1). In parallel to the maintenance of the current release, the preparation of the next one starts. The preparation consists of some *milestones*, followed by some *release candidates*. For example, release 3.1 was followed by milestone 1 of release 3.2 (named 3.2M1), further five other milestones, and seven release candidates (3.2RC1, 3.2RC2, etc.), culminating in minor release 3.2.

Figure 1 shows part of the builds we analysed, and their chronological and logical order. The logical order is indicated by solid arrows: each release may have multiple logical successors. The chronological order is represented by positioning the nodes from left to right: each release has a single chronological successor. The dotted arrows indicate that some builds, in which the chronological and logical orders coincide, were omitted due to page width constraints.

For our purposes, it makes more sense to order the builds by their numbers rather than by their dates, i.e. to follow a logical rather than a chronological order. The latter is useful when analysing the amount of changes per fixed time frame, which is for example necessary if one wishes to compare the evolution of different systems [12]. In our case, due to research question 1 (Section 2), we wish to check whether architectural changes are associated to particular builds. Hence, we compare changes between builds in logical order. For example, instead of analysing the chronological sequence 3.1, 3.2M1, 3.2M2, 3.1.1, 3.2M3, 3.2M4, 3.1.2 (see Figure 1), we either follow the main sequence 3.1, 3.1.1, 3.1.2 or the milestone sequence 3.1, 3.2M1, 3.2M2, ..., 3.2. For this paper we analysed two build sequences: the 26 major, minor and service releases from 1.0 to 3.5.1 over a period of almost 8 years (from November 2001 to September 2009), and the

27 milestones and release candidates between 3.1, 3.2, and 3.3 over a period of 2 years (from June 2005 to June 2007).

3.2 The Data Model

To perform our analyses in a systematic way and to be able to reapply them to other case studies, we define a very simple structural model and associated metrics. We were inspired by an existing axiomatic metrics framework [5], in which a generic structural model serves to impose constraints to characterize different kinds of metrics (size metrics, cohesion metrics, etc.). Our structural model is simpler and our metrics largely follow the constraints proposed in [5].

We represent a *module* (to use a relatively neutral term) by a directed graph, where nodes represent elements and arcs represent a binary relation between elements. Each element is classified as being either *internal* or *external* to the module. Likewise, internal relationships *IR* are those between internal elements *IE*, while external relationships *ER* are those between an internal and an external element *EE*. In this way, the description of a module also includes the connections to its context. Formally, a module is a graph $G = (IE \cup EE, IR \cup ER)$, such that $IE \cap EE = \emptyset$, $IR \subseteq IE \times IE$, and $G' = (IE \cup EE, ER)$ is a bipartite graph.

We define the following metrics on modules.

- The *size* of a module is the number of internal elements: $\text{size}(G) = |IE|$.
- The *complexity* is the number of internal relationships: $\text{complexity}(G) = |IR|$. Since it is impossible for a single metric to fully capture complexity, our aim was to define it as simply and as generally as possible.
- The *cohesion* could be defined as the ratio between the complexity and the square of the size. The reason for this definition is for cohesion to be normalised and to reach its maximal value for complete graphs. Given that we should not expect a well designed architecture to evolve towards a complete graph, we define the metric instead as to be a simple relationships to elements ratio: $\text{cohesion}(G) = \text{complexity}(G)/\text{size}(G)$.
- The *coupling* of a module is the number of (incoming and outgoing) external dependencies: $\text{coupling}(G) = |ER|$.

The graph-based model is generic enough for modules, elements and relationships to represent almost anything. For example, modules and elements can represent Java packages and classes, respectively, with arcs representing the inheritance relation. A module may also correspond to a class, with elements representing methods and arcs representing the call relation.

For our purposes, we wish to apply the model to Eclipse and other plugin-based architectures. Therefore, we take a module to be the whole architecture of a sub-system (the Eclipse SDK in this case study) and an element to be a plugin, while relationships may denote the static or dynamic dependencies. Because of the latter, we also need to include in the model the extension points provided and required by each plugin.

We use a relational representation instead of a graph-based one, for practical reasons. Operationally, the first step consists of defining the following relations from the repository's data:

- $IP(p)$ or $EP(p)$ holds if p is an internal or external plugin
- $Prov(p, e)$ or $Req(p, e)$ holds if plugin p provides or requires extension point e
- $SD(p, p')$ holds if plugin p statically depends on plugin p'

From these, the following relations can be computed:

- internal static dependencies $ISD(p, p') \equiv SD(p, p') \wedge IP(p) \wedge IP(p')$
- external static dependencies $ESD(p, p') \equiv SD(p, p') \wedge \neg ISD(p, p')$
- dynamic dependencies $DD(p, p') \equiv \exists e : Prov(p', e) \wedge Req(p, e)$
- internal dynamic dependencies $IDD(p, p') \equiv DD(p, p') \wedge IP(p) \wedge IP(p')$
- external dynamic dependencies $EDD(p, p') \equiv DD(p, p') \wedge \neg IDD(p, p')$
- internal dependencies $ID(p, p') \equiv ISD(p, p') \vee IDD(p, p')$
- external dependencies $ED(p, p') \equiv ESD(p, p') \vee EDD(p, p')$

Given the above relations, computing the metrics is just a matter of computing the cardinality (i.e. the number of tuples) in the appropriate relation. For example the size is $|IP|$ and the complexity is $|ISD|$ or $|IDD|$ or $|ID|$, depending on which dependencies we take as the arcs. Note that in general $|ID| \leq |ISD| + |IDD|$.

The relational model further allows to compute missing (i.e. required but not provided) and unused (i.e. provided but not required) plugins and extension points. For example, given all plugins $P(p) \equiv IP(p) \vee EP(p)$ and all dependencies $D(p, p') \equiv SD(p, p') \vee DD(p, p')$ we have

- missing plugins $MP(p) \equiv \exists p' : D(p', p) \wedge \neg P(p)$
- unused extension points $UEP(e) \equiv \exists p : Prov(p, e) \wedge \neg \exists p' : Req(p', e)$

Missing artefacts indicate potential compile-time or run-time errors, or an ill-defined module boundary, or some problem with the data mining process. Unused artefacts tell us how open and extensible the module is. Too many unused elements such as unused extension points provided by the internal plugins, might be an indication of premature generality. A completely self-contained and closed module would have no missing nor unused elements.

To allow a historical analysis, the model has to be enriched with the notion of a snapshot, which is a module at some point in time. For our case study, a snapshot is one of the Eclipse builds mentioned in Section 3.1. All the above relations must have an additional argument stating the snapshot in which they hold. For example, $P(p, s)$ holds if plugin p exists at snapshot s and $SD(p, p', s)$ holds if p statically depends on p' in snapshot s. To allow flexibility in the choice of the snapshot sequences to analyse, we allow the researcher to define the relation $Next(s, s')$, which states that snapshot s' comes immediately after snapshot s. The relation is considered ill-defined if a snapshot succeeds itself, has more than one successor, or if more than one snapshot has no predecessor.

The unique snapshot without predecessor is considered the first release of the sequence: $First(s') \equiv \exists s'' : Next(s', s'') \wedge \neg \exists s : Next(s, s')$.

Once a sequence is defined, it is possible to compute how each module snapshot has been obtained from the previous one. In particular, we compute:

- added plugins $AP(p, s') \equiv P(p, s') \wedge Next(s, s') \wedge \neg P(p, s)$
- kept plugins $KP(p, s') \equiv P(p, s') \wedge First(s) \wedge P(p, s)$
- deleted plugins $DP(p, s') \equiv Next(s, s') \wedge P(p, s) \wedge \neg P(p, s')$
- previous plugins $PP(p, s') \equiv P(p, s') \wedge \neg AP(p, s') \wedge \neg KP(p, s')$

and similarly for static and dynamic dependencies. This of course assumes that elements and relations maintain a unique name throughout the module's history, which means that a renaming will be counted as a simultaneous deletion and addition. The aim of computing the kept (i.e. unchanged) elements and relationships of the module is to address Question 4 in Section 2.

3.3 The tool infrastructure

We developed a suite of small tools that first extract the data, then compute the metrics, and finally visualise the results. However, we took care to make the suite relatively independent of our particular needs, in order to be useful in a variety of contexts. Therefore, instead of developing a standalone application or an extension for a particular IDE, we have put together a simple pipeline architecture of scripts that manipulate text files. This makes it easier to interface with other tools and to replace part of the pipeline, e.g. for a different case study.

A partial architecture of our tool suite[3] is shown in Figure 2 as a set of processes that convert input data files on the left into the output data files on the right. Among the processes, *fact extractors* obtain factual relations from artefacts of a single release of the software system and store the relations in Rigi Standard Format (RSF) files. RSF is a simple and widely used text format in which each line represents a tuple, with the relation name being followed by each tuple element, separated by spaces [29]. We next used the relational calculator Crocopat [3] to implement a *fact merger* that combines facts about selected individual snapshots into a single fact base by adding the snapshot id to every relation tuple. *Metric calculators* compute from the fact base a number of metrics, such as size and complexity. The *reporters* present the metrics and the architecture in a number of ways, including various visualisations. In the remaining of this section we detail parts of the mining process.

For each Eclipse plugin there is an XML file, called `plugin.xml`, that lists the extension points provided and used by that plugin, and the other plugins it depends on for compilation. Since release 3.0, the static dependency is in another file, `MANIFEST.MF`, which is not in XML format. These metadata files are hence a straightforward source of dependency information between plugins, saving us from having to delve into their source code. We fully agree with Alex Wolf's argument in his WICSA'09 keynote, that configuration files are an underexplored

[3] The complete suite also includes the mining of Bugzilla repositories [28].

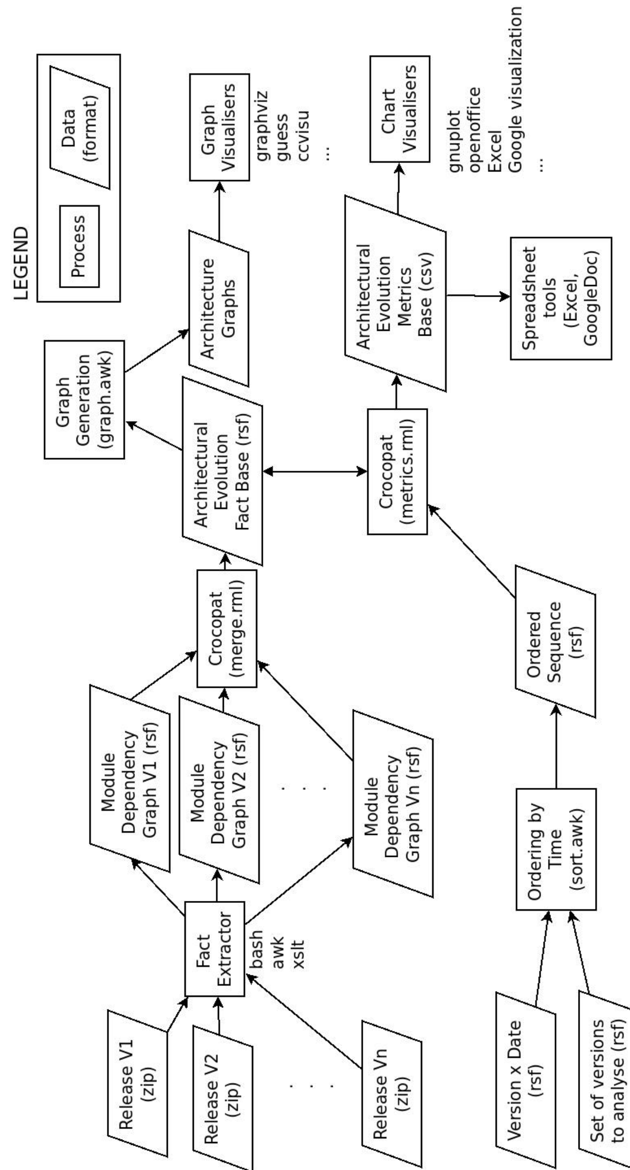

Fig. 2. Overview of our toolset

source of architectural information, which has so far been mainly extracted, in a potentially not very reliable way, from source code.

We first considered extracting the metadata files for each build directly from the CVS repository, for example by checking out all files with tag `R_3_1` (CVS tags cannot include periods) in order to obtain the information about release

3.1. However, after a while we found out that there is no direct correspondence between CVS tags and builds. In other words, comparing the set of metadata files obtained from the CVS repository with the set of those included in the actual builds, we found that often the two sets didn't coincide. We also tried to check out the files according to the known date of the build, but again there was a mismatch. We realized the Eclipse project uses for each build a complicated file that indicates which source files are included.

The input to our analysis is therefore not a CVS repository, but a set of compilable source code archives, one per build we wish to analyse. How each source code archive was obtained is not of concern to our tool, making it independent of any configuration management system. In our case, for each of the builds we analysed, we downloaded the source code of the whole SDK from http://archive.eclipse.org or its mirrors. In previous work [26, 27] we only analysed builds up to 3.3.1.1. When starting to download more recent ones for this paper, we were dismayed to find out that the Eclipse project no longer keeps older milestones and release candidates in their archive, in order to save storage and bandwidth. We therefore were only able to add releases (6 of them) for this paper. Fortunately, we kept a copy of the previously downloaded milestones and release candidates, enabling us to do further mining on them.

The repository is first processed by some shell, AWK and XSLT scripts that extract the information about the existing architectural elements from the `plugin.xml` files (and `MANIFEST.MF` files, depending on the build). The result of this processing is a RSF file with the basic relations (IP, EP, $Prov$, Req and SD) presented in Section 3.2. Whereas in previous work we defined as an internal plugin any component for which a plugin.xml file existed, in this paper we only take the subset of those where the name starts with org.eclipse but does not end in source. A source plugin wraps the source code of some other plugin, so that the code can be accessed for help and debugging purposes in the Eclipse IDE, by providing extensions to the pde.core plugin. Given that source plugins don't add functionality, we decided to ignore them for this study. Moreover, since in recent releases many plugins also have their source counterpart, this would greatly inflate the metrics, in particular the size metric.

Once we have the basic relations for each snapshot, we use Crocopat first to merge all RSF files into a single one (top left of Figure 2) as mentioned before, and second to compute any derived relations and metrics (Section 3.2 and centre of Figure 2), given the snapshot sequence. For example, from the $Prov$ and Req relations between plugins and extension points, a Crocopat script computes the dynamic dependency relation among plugins. Crocopat is also used to compute transitive closures over dependencies, in order to detect dependency cycles. The Crocopat script also computes added, deleted and kept plugins and dependencies, distinguishing between unforced and forced additions and deletions. We will explain those concepts in Section 4.

Finally, for the 'front end' of the chain, we use Crocopat and AWK to automatically translate the relevant relations in the RSF files (e.g. SD) into files for

input to graphviz[4], GUESS [1] and CCVisu [2]. This allows to display or animate the architectural structure in various ways. As for showing the evolution of metrics along build sequences, we simply use bar and line charts. In previous work we used Crocopat to generate spreadsheets in OpenOffice's XML format and used OpenOffice or Excel to create the charts. For this paper we took another path: Crocopat generates comma separated value files (one for each sequence) which we upload to Google Spreadsheets. We then wrote Javascript code that calls the Google Visualization API[5] in order to get the data from the spreadsheets, generate charts and embed them into a web page.

Using Google tools has several advantages over the previous approach. First, the data is made public to other researchers and in various formats (HTML, OpenOffice, Excel) without any additional effort on our part. Second, the bar and line charts are large and interactive, allowing the reader to click on the data points to see the exact values, instead of just perceiving generic trends from a small, static, and grey chart in a paper. Third, the Google Visualization API includes an expressive data query language that allows some calculations to be performed on the fly, like computing the ratio of the values in two columns. This means that some additional metrics can be presented without having to change the Crocopat script, run it again and upload the new spreadsheet.

Overall, our tool infrastructure has been designed and developed over time with the aim of being flexible, light-weight and interoperable. Flexibility and interoperability are achieved by an open and easy to modify pipe-and-filter architecture in which the pipes are text files in standard formats (XML, RSF) and the filters are scripts executed by widely used, freely available, and generic data processing and visualization tools (AWK, XSLT, Crocopat, graphviz, etc.). Due to this, it should not be too difficult to integrate our scripts within existing tool chains, like FETCH [4], and to modify the 'back-end' to handle other systems besides Eclipse.

The approach is light-weight because it is independent of any particular configuration management tool like CVS or Subversion, because it just relies on metadata files and not on static code analysis, and because the relations are kept in text files. Given the small size of the database (87,397 tuples for the 53 Eclipse builds analysed), our approach remains very efficient.

4 The results

After presenting the data model and how the data is mined and processed, we are in a position to show the results. The charts presented in this section (and others) can be interacted with at a web page[6] that also links to the spreadsheets with all measurements.

To show the evolution of the metrics over the two snapshot sequences, we use mostly stacked bar charts, with each bar segment showing a particular subset of

[4] http://www.graphviz.org/
[5] http://code.google.com/apis/visualization
[6] http://michel.wermelinger.ws/chezmichel/2009/10/the-architectural-evolution-of-eclipse

the total number of items (plugins or dependencies). The segments are stacked, from bottom to top, as follows: unforced deletions, forced deletions, kept items (i.e. since the first snapshot in the sequence), previous items, forced additions, unforced additions. In general, a change is considered unforced if it is by choice, and forced if it is due to another change, e.g the unforced deletion of a plugin forces the deletion of all its extension points and dependencies.

We use the same colour for unforced additions and deletions, and the same colour for forced deletions and additions. Since deletions are represented by negative numbers and additions by positive ones, there is no possible confusion. We also use a darker colour to distinguish kept from previous items. On the PDF version of this paper you can see we use warmer colours (red and orange) for changed and cooler colours (blue tones) for unchanged items. The aim of these choices was to have a reduced colour palette that translated well to grey scale values in the printed version, while using position and hue to quickly draw the reader's attention to the unforced changes at the extremities of each bar.

4.1 Size

Figure 3 shows the evolution of Eclipse's size, along the two snapshot sequences. Note that the number of kept plugins is with regard to the first release in the sequence, i.e. 1.0 or 3.1. We consider all plugin additions and deletions as unforced, because they are architectural choices.

We can observe that, over all releases, the size of the architecture increases more than sevenfold, from 35 to 271. The evolution follows a segmented growth pattern, in which different segments have different growth rates. In particular, the rate is zero during service and positive during major and minor releases. A look at the interim builds reveals that most of those changes occur in milestones, although some also occur in the later release candidates.

Segmented growth patterns have been observed for other open source systems, as surveyed in [12]. Those studies also observed superlinear growth, i.e. growth with increasing rates, which is not the case here. Our hypothesis is that while those studies focused on source code, we focus on the architecture, which, to remain useful and understandable to stakeholders, has to be kept within a reasonable size. In fact, the evolution of the size follows a pattern observed for other systems [30]: long *equilibrium periods*, in which changes can be accommodated within the existing architecture, alternate with relatively short *punctuation periods*, in which changes require architectural revisions.

4.2 Complexity

Figure 4 plots the changes to overall complexity, i.e. to relation ID (Section 3.2). The web page indicated earlier provides additional charts for static and dynamic internal dependencies and for milestones and release candidates. A forced addition or deletion of a dependency is associated to the creation or removal of

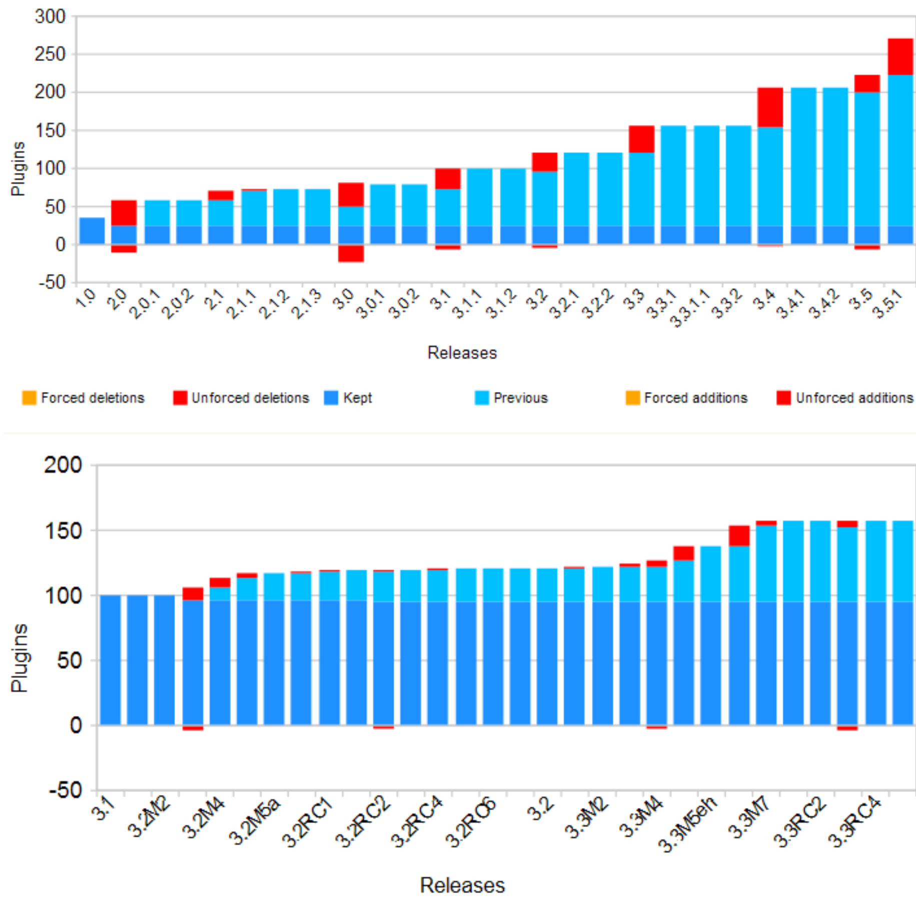

Fig. 3. Evolution of the size

at least one of the involved plugins, i.e. the addition (resp. deletion) of a dependency between two plugins is called unforced if both plugins already existed (resp. still remain).

Again, dependencies change mostly during milestones and remain the same during service releases, except for a few deletions in 3.3.1. However, contrary to continuous increase of size, there has been a decrease of complexity in release 3.1, i.e. there was some effort to counteract the system's growth.

Moreover, the chart shows that most additions are forced, i.e. new dependencies are due to new plugins, while most deletions are unforced, i.e. due to changes in the plugins' implementations in order to reduce dependencies.

The new releases analysed for this paper continue to keep the same plugins and dependencies since release 1.0, as seen by the continuous dark blue segments in Figures 3 and 4. The architectural core is hence the same as presented in [27].

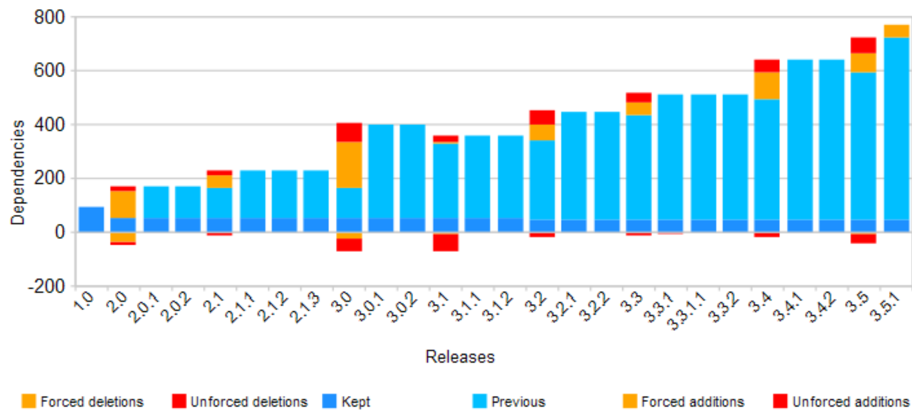

Fig. 4. Evolution of the overall complexity

4.3 Cohesion

The previous charts show that size and complexity grow 'in sync', following the same punctuation and equilibrium pattern. There are however two exceptions: release 3.0 substantially increased the complexity while only slightly increasing the size, and release 3.1 decreased the complexity while increasing the size.

Hence, computing the cohesion, we note it is remarkably almost constant (Figure 5) except for the increase at 3.0, which was kept until 3.1 because service releases didn't change the architecture. After 8 years, the cohesion levels of release 3.5.1 (1.40 internal dynamic dependencies and 2.17 static ones per plugin) are very similar to those of the much smaller release 1.0. The chart also shows that there are many more static dependencies than dynamic ones, as can be checked with the additional complexity charts on the web site mentioned earlier.

Interestingly, when we showed the previous version of this chart [27] to Eclipse developers at IBM Zurich, we were told there was no explicit aim to keep the cohesion constant. Nevertheless, we conjecture this might be an indirect consequence of possibly wishing to keep the various Eclipse SDK sub-systems (the Plugin Development Environment, the Java Development Toolkit, etc.) loosely cohesive to facilitate the configuration of the IDE to individual needs.

4.4 Coupling

The evolution of coupling also follows a segmented growth pattern, but with a substantial decrease in release 3.0, which replaced all external dependencies (Figure 6). Release 3.1 further reduced the dependency on external plugins, although it grew again in later releases.

We looked into the actual dependencies and plugins involved, and realized that plugins that depended on external plugins in 2.1.3, depend in 3.0 on new

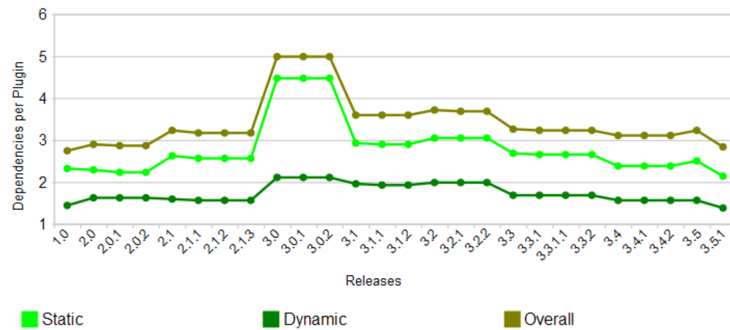

Fig. 5. Evolution of the cohesion

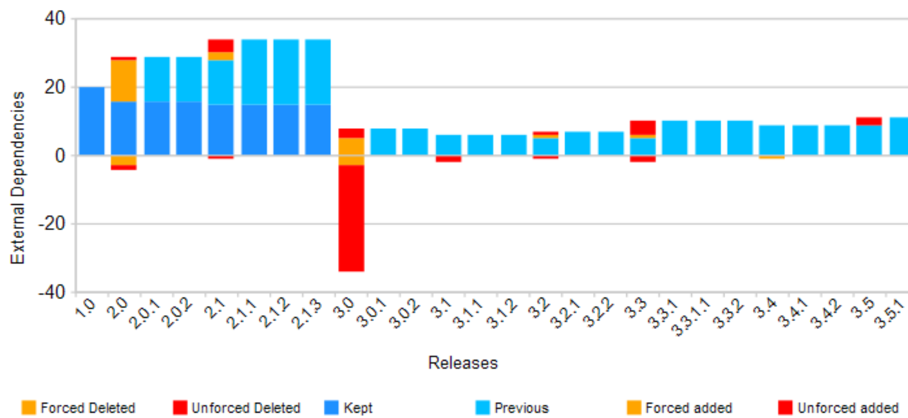

Fig. 6. Evolution of the coupling

internal plugins which in turn depend on the external plugins. In other words, release 3.0 introduced internal 'proxy' plugins for the external plugins, and this reduced coupling between Eclipse and third-party components. Additionally, one of the external plugins used by release 2.1.3, org.apache.xerces, was removed. Figure 6 sums up all these modifications as unforced changes (the rewiring) and forced changes (due to the removed plugin and new proxies). Overall, the chart shows most changes to the coupling are unforced, i.e. by choice rather than due to the addition or removal of plugins.

4.5 Summary

We can now return to the initial questions (Section 2) and summarize what the archaeological investigation has shown.

1. The development of Eclipse follows a systematic process in which the architecture is mainly changed during the milestones of the next major or minor

release. Some release candidates may still introduce some small changes, but the architecture is frozen for the last few builds before the release. Service releases almost never introduce any architectural changes.
2. Overall, the Eclipse architecture is always growing and as such follows Lehman's 6th law of evolution. Due to the systematic change process, such growth follows a known segmented pattern of alternating long equilibrium and shorter punctuation periods, the latter mostly during milestones. Complexity (as measured by the dependencies among plugins) also increases, as Lehman's 2nd law postulates, and does so following the same segmented growth pattern as size.
3. There has been some effort to reduce the system's growth, but overall deletions are far fewer than additions, possibly to avoid breaking the many existing third-party Eclipse plugins. The major reduction efforts have been in releases 3.0 (small size growth, reduced coupling) and 3.1 (reduced complexity and coupling).
4. The new releases analysed for this paper continue to use the layered architectural core we presented before [27].
5. The Eclipse architecture is kept loosely cohesive during its evolution, contrary to our initial expectations. However, Eclipse developers follow the usual advice of minimising coupling: the number of external static dependencies is very small compared to the number of internal ones and there have been explicit efforts (i.e. unforced changes) to reduce coupling.

To sum up, we were not able to find any empirical evidence to falsify the investigated design guidelines and evolution laws, with the possible exception of increased cohesion. From the above observations (systematic process, segmented growth, punctual but extensive restructurings, and avoidance of deletions), we feel that Eclipse can be used as a pedagogical case study of best practice to achieve sustainable architectural evolution of software frameworks.

5 Discussion

Reflecting on our work, we can pass on several lessons and issues to be aware of when embarking on an archaeological investigation of software evolution.

For brevity and clarity, most research papers only present the results, i.e. the 'after the fact' picture of the research process, in which all pieces of the puzzle fit into a perfectly logical conceptual building. The dead ends and twists and turns of the path that led to the results remain often unreported. In reality, the process is not as linear as the papers, written on hindsight, seem to imply. In particular, software archaeology is iterative and incremental, with a constant interplay between research questions, which provide the overall guidance, and the available data, which constrains what can be done. Both parts mutually influence each other and together shape the overall data mining and analysis. For example, while we presented the research questions (Section 2), the data model (Section 3.2) and its extraction from the Eclipse repository (Section 3.3) in a sequential fashion, each phase apparently determining the next one, in reality

a preliminary analysis of the repository was needed to assess what data could be extracted, i.e. what kinds of builds and architectural information was available, which in turn helped shape the research questions, the abstracted data model and the tool set. Software archaeologists must therefore be prepared to 'follow' the data, especially if faced with lost artefacts, as mentioned in the introduction. Like in real archaeology, software-related artefacts may be lost because they were not recorded in a persistent way, or because they were later 'destroyed' by accident or on purpose. Keeping your own copy of datasets might be a good idea, as we found out (Section 3.3).

However, the research cannot be completely data-driven. Software repositories are simply too rich and big for an ad-hoc exploration to guarantee interesting results with little effort. Research questions are hence fundamental to frame and guide an efficient mining process. Moreover, questions must be explicit and relevant in order to avoid the dreaded 'so what?' question by critics. Relevancy can be pedagogical, practical or theoretical, often being a mix of the three, as in our case: while the main aim was theoretical, seeking empirical evidence of design guidelines and evolution laws, the results can be used for teaching purposes, using Eclipse as a good practice exemplar of architectural evolution.

The mining infrastructure should also be a reusable asset. Tool development takes considerable effort; return on investment is obtained by using the tools over several research iterations. Moreover, one should strive to build upon third-party infrastructure. Examples of reusable tools that build upon other existing tools are MoDisco[7], an Eclipse plugin, and the batch-oriented tool chain FETCH [4]. Both approaches have advantages. Tools within IDEs become part of the developers' workflow: the archaeological process is tightly integrated with the development process, each one feeding into the other. IDE-independent tools like FETCH can be more general and flexible, because wiring together existing generic data processing and visualization tools allows adaptation to a variety of research scenarios and data sources. On the other hand, tools like MoDisco aim to achieve such flexibility by providing a generic model transformation infrastructure that is able to generate metrics, visualizations and documents from models, allowing users to tailor the models and transformations to their particular needs. While our approach is also driven by a model (of the system's structure), it is ad-hoc in the sense that the model and metrics, albeit generic, are fixed, whereas a truly model driven approach like MoDisco is much more customizable, systematic, expressive and reusable. However, such characteristics come at a price: model-driven approaches require heavy-weight infrastructure and considerable investment from the user to learn and customize it, even for simple models and measurements like those in this paper.

Once the data has been mined and processed, it has to be presented. Simple quantitative displays (e.g. line diagrams) are a good indication of the change rate, but visualising the actual transformations (e.g. the before and after architecture) still poses a challenge, even if using animations. As the charts in Section 4 indicate, even at the highest level of design abstraction, any realistic system

[7] http://www.eclipse.org/gmt/modisco

comprises hundreds of artefacts. Presenting them in an understandable way on a big screen is challenging, let alone on paper. We have experimented with graphviz and GUESS, but results were unsatisfactory. Only graphs with relatively few nodes and arcs, like the architectural core, can be easily depicted.

Contrary to Physics and other subjects, there is not yet a culture in Computing that leads authors to fully publicise the data on which their conclusions are based, so that other researchers can build on it and independently verify it. Publishers do not yet provide the means for such data to be stored and accessed as easily as the papers that report on the data. Fortunately, due to the Web 2.0 it is becoming easier for authors to publish their data and visualizations, and we described one way to do so in Section 3.3.

Tracking changes in artefacts is a long-standing research strand. As mentioned in the introduction of this paper, one of the issues is abstraction, in particular how to abstract fine-grained changes into meaningful transformations. One possible heuristic is to attempt to minimize the number of transformations that encompass all observed changes. Two approaches that follow such a strategy for source code changes are [6, 13]. Those proposals appeal to the language engineering community [16, 9] where the primary artefacts are text-based.

On the other hand, when the artefacts are structured as models, one may leverage more semantic information (e.g. from UML model elements and their relationships) to detect structural changes [31, 22]. Such approaches appeal to the model-driven engineering community because the basic changes detected can suggest more complex adaptive framework changes [11] at the modeling level.

Whereas text or model comparison reconstructs the actual changes, measuring changes is a good way to spot overall trends. One particularly relevant trend for evolution is zero changes, i.e. what does *not* change. In our work, it corresponds to the architectural core, an important design feature.

However, metrics don't tell the whole story: they don't capture all the 'what' and 'how' of evolution and certainly not the 'why'. Hence, measurements should be complemented by an inspection of the actual artefacts and, if possible, by other information sources, e.g. bug reports or the system's developers. For example, metrics and the distinction between forced and unforced changes can tell that Eclipse was restructured in release 3.0, but only looking at plugins can one understand it was in part due to the adoption of the OSGi run-time infrastructure. Also, without asking the developers one might assume that the constant cohesion is a deliberate design aim.

In spite of all sources of information one can consult, researchers and their audience must accept that in software archaeology there will always be some space for subjective interpretation, first because, contrary to apples falling on scientists' heads, software projects don't follow any natural laws, and second because threats to the validity of the conclusions can hardly be completely eliminated. There might be errors in the mining infrastructure, the statistical method employed might be inappropriate for the data at hand, etc. As in the natural sciences, any abstraction/model can only provide a partial view on the studied subject, and software development is a complex socio-technical endeavour with

many potential confounding factors. In our case, the simple size, complexity and cohesion metrics only provide a very partial view of software architecture.

Researchers often strive to justify they adequately handled the threats to validity, but it is probably sometimes better to just point to them as opportunities for further improvement. After all, research is a community practice, not an individual pursuit.

6 Conclusions

This paper is a twofold tutorial. On the one hand, in a similar spirit to the case studies presented in business management literature, the research provides empirical evidence for using Eclipse as a tutorial case study on sustainable good practice for architectural evolution. On the other hand, the research serves as a tutorial-by-example on some of the issues faced when doing archaeological investigations into software evolution. For that, we provide more details on the research process than in our previous papers, make the measurements publicly available, and reflect on our experience.

Software archaeology, not just for evolution, is blooming due to the increased availability of rich software project repositories. We hope this paper helps the next generation of researchers in this exciting area.

References

1. Adar, E.: GUESS: a language and interface for graph exploration. In: Proc. SIGCHI Conf. on Human Factors in Computing Systems. pp. 791–800. ACM (2006)
2. Beyer, D.: CCVisu: automatic visual software decomposition. In: Proc. of Int'l Conf. on Software Engineering, companion volume. pp. 967–968. ACM (2008)
3. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. IEEE Trans. Software Eng. 31(2), 137–149 (2005)
4. Bois, B.D., Rompaey, B.V., Meijfroidt, K., Suijs, E.: Supporting reengineering scenarios with FETCH: an experience report. Electronic Communications of the EASST 8 (2008)
5. Briand, L.C., Morasca, S., Basili, V.R.: Property-based software engineering measurement. IEEE Trans. Software Eng. 22(1), 68–86 (1996)
6. Canfora, G., Cerulo, L., Di Penta, M.: Tracking your changes: A language-independent approach. IEEE Softw. 26(1), 50–57 (2009)
7. Canfora, G., Penta, M.D.: New frontiers of reverse engineering. In: Future of Software Engineering. pp. 326–341. IEEE (2007)
8. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. 61(3), 190–210 (2006)
9. Cordy, J.R.: The txl source transformation language. Sci. Comput. Program. 61(3), 190–210 (2006)
10. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (June 2000)
11. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: Proc. 30th Int'l Conf. on Software Engineering. pp. 481–490. ACM (2008)

12. Fernández-Ramil, J., Lozano, A., Wermelinger, M., Capiluppi, A.: Empirical studies of open source evolution. In: Software Evolution, chap. 11, pp. 263–288. Springer Verlag (2008)
13. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling:tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering 33, 725–743 (2007)
14. Jackson, M.: Automated software engineering: supporting understanding. Autom. Softw. Eng. 15(3-4), 275–281 (2008)
15. Kagdi, H.H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance 19(2), 77–131 (2007)
16. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. 14(3), 331–380 (2005)
17. Madhavji, N.H., Fernandez-Ramil, J., Perry, D.E.: Software Evolution and Feedback: Theory and Practice. Wiley (2006)
18. Martin, R.C.: Large-scale stability. C++ Report 9(2), 54–60 (Feb 1997)
19. Mens, T., Demeyer, S. (eds.): Software Evolution. Springer (2008)
20. Osterweil, L.: Software processes are software too. In: Proc. 9th Int'l Conf. on Software Engineering. pp. 2–13. IEEE (1987)
21. Popper, K.R.: The Logic of Scientific Discovery. Hutchinson (1959)
22. Schmidt, M., Gloetzner, T.: Constructing difference tools for models using the sidiff framework. In: Proc. 30th Int'l Conf. on Software Engineering, companion volume. pp. 947–948. ACM (2008)
23. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
24. Shull, F., Singer, J., Sjoberg, D.I. (eds.): Guide to Advanced Empirical Software Engineering. Springer (2008)
25. Stringfellow, C., Amory, C., Potnuri, D., Andrews, A., Georg, M.: Comparison of software architecture reverse engineering methods. Information and Software Technology 48(7), 484 – 497 (2006)
26. Wermelinger, M., Yu, Y.: Analyzing the evolution of Eclipse plugins. In: Proc. 5th Working Conf. on Mining Software Repositories. pp. 133–136. ACM (May 2008)
27. Wermelinger, M., Yu, Y., Lozano, A.: Design principles in architectural evolution: a case study. In: Proc. 24th Int'l Conf. on Software Maintenance. pp. 396–405. IEEE (October 2008)
28. Wermelinger, M., Yu, Y., Strohmaier, M.: Using formal concept analysis to construct and visualise hierarchies of socio-technical relations. In: Proc. 31st Int'l Conf. on Software Eng., companion volume. pp. 327–330. IEEE (May 2009)
29. Wong, K.: The Rigi User's Manual, Version 5.4.4 (June 1998)
30. Wu, J., Spitzer, C., Hassan, A., Holt, R.: Evolution spectrographs: visualizing punctuated change in software evolution. In: Proc. 7th Intl. Workshop on Principles of Software Evolution. pp. 57–66 (2004)
31. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proc. 20th Int'l Conf. on Automated Software Engineering. pp. 54–65. ACM (2005)
32. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.C.S.: Reverse engineering goal models from legacy code. In: Int'l Conf. on Requirements Engineering. pp. 363–372. IEEE (2005)
33. Yu, Y., Wermelinger, M.: Graph-centric tools for understanding the evolution and relationships of software structures. In: Proc. 15th Working Conf. on Reverse Engineering. pp. 329–330. IEEE (October 2008)