Design Principles in Architectural Evolution: a Case Study

Michel Wermelinger Yijun Yu Angela Lozano
Computing Department & Centre for Research in Computing
The Open University, UK

Abstract

We wish to investigate how structural design principles are used in practice, in order to assess the utility and relevance of such principles to the maintenance of large, complex, long-lived, successful systems. In this paper we take Eclipse as the case study and check whether its architecture follows, throughout multiple releases, some principles proposed in the literature.

1. Introduction

Throughout the years, there has been a vast amount of advice on how to design and program software systems, covering a wide range of goals: increasing performance, understandability, or reuse, facilitating testing, etc. Advice also varies in the way those goals are achieved, either through usage of language-specific features [3], paradigm-specific catalogues of generic solutions to recurring problems [1], or general principles [10].

Melton [12] points out that it is largely unknown how object-oriented systems are structured in practice and he advocates for empirical studies that analyse how object-oriented design principles are followed (or not) in practice. Melton argues this would shed light on how effective such principles are to bring about certain system qualities, and it would align academic research better with practical needs. In a separate paper [13], Melton and Tempero check the design principle that dependencies between classes should be acyclic. They measure across a wide range of Java applications the sizes of all cycles present in each application, obtaining a range of different values.

As researchers and educators, we feel it should be our duty to scientifically investigate the validity and relevance of various design principles, so that more specific guidance can be given about which principles are useful in which contexts to achieve which qualities. We therefore wish to contribute to the above research agenda, but we take a different approach. Instead of studying indiscriminately a number of systems, we prefer to select large, complex, long-lived systems developed and used by a large community, as it is for such systems that design principles should be more relevant to sustain the system's continuous maintenance, robustness, and usefulness. By checking which principles such systems follow and which they do not, we hope to get an indication of which principles are most useful.

The systems we are interested in require a clean architecture that captures the major design decisions that influence not only the structure, but also the behavioural interactions, the development, and the business position of the system [11]. One might hence argue that for large and complex systems design principles should be followed particularly at the architectural level, as they will contribute to a clean and understandable software architecture that can adapt to the system's evolution. However, can design principles that have been mainly developed in the context of packages and classes be lifted to the context of architectural components?

To sum up, we wish to investigate the validity, usefulness and generality of design principles, and to do so through empirical studies of their usage in selected systems, for which such principles are relevant at the architectural level. A previous short paper [16] was our first step in that direction: we looked at how Eclipse components and their dependencies evolve over several releases. It was an exploratory study that didn't relate the results to any design principle. This paper takes that work much further, following a systematic approach to the definition and measurement of relevant properties and to the validation of principles.

2. Concepts and Metrics

2.1. Eclipse

Each *release*[1] of the Eclipse SDK (except for release 1.0) provides one or more high-level *features*. Each feature is implemented by a set of *plugins*, Eclipse's components. Each plugin may *depend* for its compilation on Java classes

[1]The Eclipse project uses the term 'release' just for certain kinds of 'builds'. We only analyse stable builds meant for users, hence our preference for the term 'release'.

that belong to other plugins. For example, the implementation of plugin platform (we omit the default org.eclipse prefix) depends in release 3.3.1.1 on eight other plugins, including core.runtime and ui. Each plugin *provides* zero or more *extension points*. These can be *used* 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 runtime 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 releases. There are various types of releases in the Eclipse project. In this paper we analyse *major releases* (e.g. 2.0 or 2.1) and the *maintenance releases* that follow them (2.0.1, 2.0.2, 2.1.1, etc.). In parallel to the maintenance of the current major release, the preparation of the next major release 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) and five other milestones, followed by seven release candidates (3.2RC1, 3.2RC2, etc.) until culminating in major release 3.2.

Figure 1 shows part of the 47 releases 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. Due to page width constraints, we split the timeline in two, at release 3.1. The dotted arrows indicate that some sequences of releases were omitted due to space constraints. We only did that when the chronological and logical orders coincide.

2.2. Modules

To perform our analyses in a systematic way, we define a very simple and generic model that can be used at different levels of abstraction, although in this paper we only apply it at the architectural level.

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 are those between internal elements, while external relationships are those between an internal and an external element. In this way, the description of a module also includes the connections to its context.

This model is generic enough for modules, elements and relationships to be 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, a module represents the whole Eclipse SDK and an element is a plugin, while relationships may denote the static or dynamic dependencies. In particular, we use the following names to refer to modules.

- $Eclipse^s_r$ is the graph for release r of all Eclipse SDK plugins, with arcs denoting static dependencies. The external elements are the third-party plugins on which the SDK depends, like junit.
- $Eclipse^d_r$ is a similar module, but where arcs denote dynamic dependencies between elements. This module has no external elements, because no third-party element provides extension points and hence the Eclipse plugins do not depend dynamically on them.
- $Eclipse_r$ is the union of the previous two modules, i.e. an arc between plugins A and B indicates that A somehow depends on B, whether it is statically, dynamically, or both ways. We explain in Sections 4.2 and 4.3 the need for this module.

We define the following metrics on modules. The *size* of a module is the number of internal elements. The *complexity* is the number of internal relationships. Since it is impossible for a complexity metric to fully capture understandability, our aim was to define complexity as simply and as generally as possible. The *cohesion* is 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. The *coupling* of a module is the sum of the fanin (number of incoming external arcs) and fanout (number of outgoing external arcs). We explain further the rationale for these metrics in Section 5.

2.3. Changes

In this paper, we analyse two release sequences: the 20 major and maintenance releases from $r_1 = 1.0$ to $r_{20} = 3.3.1.1$ over a period of 6 years, and the 27 milestones and release candidates between $r_1 = 3.1$, $r_{17} = 3.2$, and $r_{30} = 3.3$ over a period of 2 years. For the purposes of analysing the architectural evolution, it makes more sense to order the releases by their numbers, rather than by their

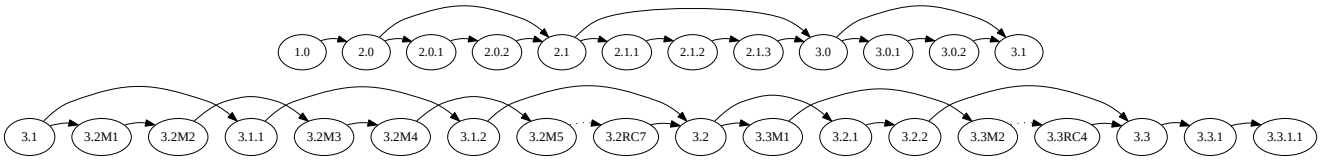

Figure 1. Chronological and logical sequence of analysed releases

dates. For example, whereas the chronological order is 3.1, 3.2M1, 3.2M2, 3.1.1, 3.2M3, 3.2M4, 3.1.2 (see Figure 1), we either follow the maintenance branch sequence 3.1, 3.1.1, 3.1.2, 3.2 or the milestone branch sequence 3.1, 3.2M1, 3.2M2, 3.2M3, ..., 3.2.

To analyse the evolution of a module's elements, we adopt and adapt some of van Belle's terminology and metrics [15]. Given a *change sequence* $C = \{c_1, c_2, \ldots\}$, in which each *change* c_i is the set of elements modified, the changes that affect a certain element e are $C(e) = \{c \in C | e \in c\}$. For our purposes, change c_i is the set of plugins modified between consecutive releases r_i and r_{i+1}, in a given release sequence. Based only on the information contained in a module description, we consider an element to have changed if its fanout set has changed. Hence, modifications to the implementation of an element e from r_i to r_{i+1} are accounted for, i.e. $e \in c_i$, only if the modification resulted in a new architectural dependency.

Note that van Belle uses a change set instead of a change sequence because the order of changes is irrelevant for his metrics. However, for most software evolution studies the order of changes is of interest in order to detect any trends. In our case, we want to observe how a module's size, complexity, cohesion and coupling evolve throughout a release sequence.

Van Belle defines the *likelihood* of a given element being changed as the percentage of changes that affected the element, i.e. $likelihood(e, C) = |C(e)|/|C|$. We slightly modify this definition by putting in the denominator the maximum number of changes that could have affected the element, which can also be defined as the age of the element (the number of releases that include the element since its creation) minus one. For example, if a plugin is introduced in r_{17} and removed in r_{20} then its age is three, because the plugin existed for three releases, while the maximum number of changes that may have modified it is two (c_{17} and c_{18}), because the next change removed the plugin. The reason for this modification to the likelihood formula is to avoid having a low likelihood for elements created late or removed early in the release sequence, even though they were frequently changed during their lifetime.

Van Belle defined the *impact* of an element's changes as the average number of elements that change, whenever e changes, i.e. $impact(e, C) = (\sum_{c \in C(e)} |c|)/|C(e)|$. If e never changed, i.e. its change likelihood is zero, then its impact is zero. Given these two metrics, the *acuteness* of changes to a given element is the impact divided by likelihood. A high acuteness means that the element changes infrequently, but when it does, it has a big impact. As an example, van Belle mentions that interfaces have high acuteness while method bodies should have low acuteness.

2.4. Principles

In 1996/7, Robert Martin wrote a series of articles on object-oriented design principles. Some of them aim at providing guidance on how to design the dependencies between the various parts of a software system in a way that facilitates the maintenance of the system. We will investigate in this paper two of those principles.

The first one is the *Acyclic Dependency Principle* (ADP), which states that the dependencies should form a directed acyclic graph [9]. The rationale is that mutual dependencies increase change propagation and make release management and allocation of work to developers more difficult.

The second one is based on the notion of *stability*, meaning resistance to change. The *Stable Dependencies Principle* (SDP) states that dependencies should be in the direction of stability [10], i.e. if A depends on B, then A should be less stable than B. The reason is that changes to B may trigger changes to A, and therefore A shouldn't be more resistant to change than B, as that will make change propagation harder.

Martin measures the *instability*, i.e. the complement of stability, of each element in the dependency graph as the element's fanout divided by its coupling, i.e. the sum of the element's fanin and fanout. The measure ranges from zero (when the fanout is zero) to one (when the fanin is zero). If the fanin is zero, the element is said to be *irresponsible*, as it provides nothing to other elements. It is easy to change irresponsible elements, because they don't trigger any further changes. Hence, irresponsible elements have the highest instability. If the fanout is zero, the element is said to be *independent*, as it requires nothing from other elements. Hence, there are no internal drivers to change the element. As Martin says, an element that is independent and responsible "has no reason to change and reasons not to change" and hence has the lowest instability value. However, he does not com-

ment on elements that are irresponsible and independent, and therefore have undefined instability.

3. Research Questions

Having all concepts and definitions in place, we can now formulate concrete research questions that fit the overall aims stated in Section 1.

The first group of questions relate to the module metrics and aim to establish whether their evolution follows any pattern observed or prescribed in the literature.

1. Does the evolution of size indicate continuous growth (Lehman's 6th law of software evolution) and follow any of the patterns observed for other open source systems [7]?
2. Does complexity increase (Lehman's 2nd law [7])? Was there any effort to maintain or reduce it?
3. Does coupling decrease and cohesion increase?

The next question probes an observation van Belle made in [15, Chapter 3] to relate his metrics with Martin's concepts: responsible elements have high impact of change, independent elements have low likelihood of changing, and hence stable elements have high change acuteness. Van Belle argues that his approach, which is based on *correlational linkage* between elements obtained from observing co-changes, goes further than Martin's approach, which assumes *causal linkage* between elements to be explicitly given, because it analyses the closure of change propagation and not just an element's immediate neighbours. It is therefore relevant to ask:

4. Is there a relationship between van Belle's and Martin's concepts to assess the changeability of elements? Can static causal linkage predict historic correlational linkage between elements?

The next question directly probes whether Martin's principles are applied (consciously or not).

5. Does Eclipse's architecture follow the ADP and SDP?

Finally, we must consider the case that some of the above questions do not make sense for our case study, because some of those concepts and principles were developed for low-level design abstractions, like classes, and not for high-level architectural components, like plugins.

6. Is it meaningful to try to validate the above principles and guidelines at the architectural level?

To address these questions we have extended the scripts we wrote for our exploratory study [16]. They first extract plugin dependency information from each considered Eclipse release, and then compute the necessary metrics and output them into spreadsheets. In this paper we just recap the main points and advantages of our data processing approach.

Eclipse keeps information about its architectural elements and relations in XML and text metadata files, saving us from having to delve into source code. We wrote bash, awk and XSLT scripts that read those metadata files, extract the relevant information, and produce text files which encode the relations of Section 1 in generalised Rigi Standard Format (RSF) [17]. Next, we use the relational calculator Crocopat [2] to compute derived relations. For example, from the `uses` and `provides` 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 cycles. We compute some metrics directly with Crocopat, while for others we use it to automatically generate spreadsheets in OpenOffice's XML format; we then use OpenOffice to further analyse and visualise the data. The next section provides the results we obtained.

Our data processing approach has two main characteristics. First, the input is just the set of metadata files of each Eclipse release to be analysed. Second, our scripts form a pipeline that reads and writes text files in XML and RSF format, using commonly available free tools. Due to the first characteristic, our approach is independent of any configuration management tool like CVS or Subversion, and it is light-weight and efficient, as it does not involve static source code analysis. Due to the second characteristic, it should be relatively easy to integrate our scripts within existing tool chains, like FETCH [4], and to modify the 'back-end' to handle other systems besides Eclipse.

4. Results and analysis

In this section we present and analyse the results obtained by running our scripts. The presentation follows the order of the questions in the previous section.

4.1. Measuring modules

Except for cohesion, the module metrics (Section 2.2) are simply the number of elements or relationships in a module. We can thus easily visualise the evolution of those metrics with bar charts: the height of each bar represents the value of the metric in a given release (Figure 2). Since the total value is the sum of the elements or relationships kept and added w.r.t. the previous release, we can use stacked bars: the lighter section (labelled A) represents what has been added, the darker one (labelled K) what has been kept. We further subdivide the latter section, showing in the darkest tone what has been kept since r_1 (label K1). By def-

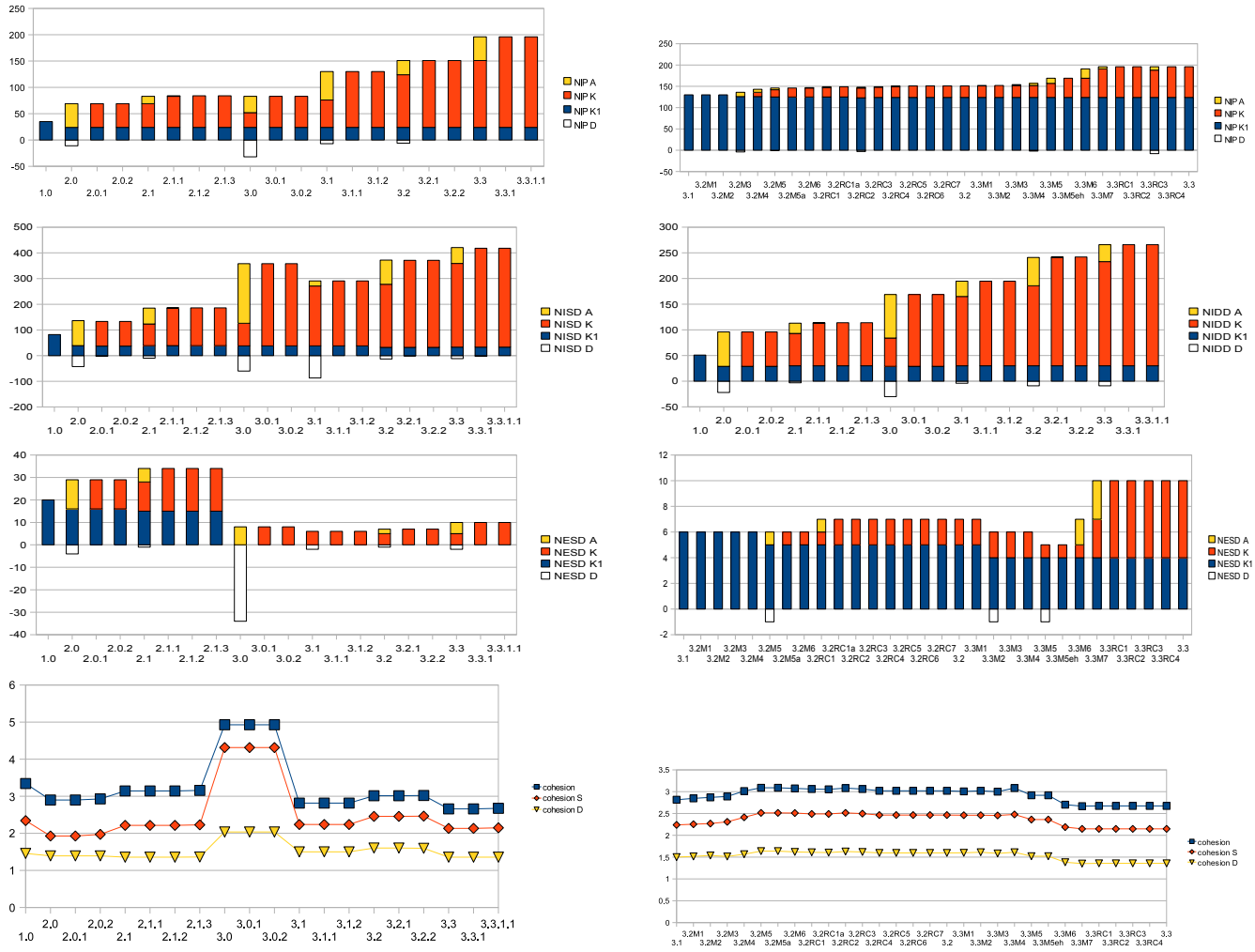

Figure 2. The evolution of module metrics along the two release sequences

inition, the bar for r_1 is completely drawn in the darkest tone. Furthermore, if the number of elements or relationships kept by a release is smaller than the total number of the previous release, then some must have been deleted. To make this more explicit, we extend each bar below the zero axis by as many elements or relationships as deleted (label D). Figure 2 shows the following metrics:

- number of internal plugins: NIP = size($Eclipse_r$)
- number of internal static dependencies: NISD = complexity($Eclipse^s_r$)
- number of internal dynamic dependencies: NIDD = complexity($Eclipse^d_r$)
- number of external static dependencies: NESD = coupling($Eclipse^s_r$)

Note that the numbers of plugins and dependencies are slightly different from those we reported previously [16] because in this paper we make a distinction between internal (Eclipse) and external (third-party) plugins. Also note that there is no chart for the number of external dynamic dependencies as it is always zero (see Section 2.2).

We can observe that, over all releases, the size of the architecture increases more than fourfold. The evolution follows a segmented growth pattern, in which different segments have different growth rates. In particular, the rate is usually zero during maintenance and candidate releases and positive during milestones. In other words, the development of Eclipse follows a systematic process in which the architecture is mainly changed during the milestones of the next major release.

Segmented growth patterns have been observed for other open source systems, as surveyed in [7]. 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 over all 47 releases follows a pattern observed for other systems [18]: 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.

To sum up, we can answer question 1 of Section 3 as follows. Overall, the Eclipse architecture is always growing and as such follows Lehman's 6th law of evolution. A closer look shows that such growth follows a known segmented pattern of alternating equilibrium and punctuation periods. However, contrary to what is known about open source code, we could not observe superlinear growth, and conjecture this will be the case for most software architectures.

As for complexity, we observe it follows the same segmented growth pattern as size. However, the number of static dependencies decreased by 19% in release 3.1, although the number of plugins increased by 57%. This indicates a major restructuring effort in order to improve the architecture for future system evolution. We can therefore say that although complexity increases as Lehman's 2nd law postulates, there has been some effort to counteract its growth, as the various deletions in the NISD and NIDD charts of Figure 2 show.

The evolution of coupling also follows a segmented growth pattern, but with a substantial negative growth in release 3.0, which replaced all external relationships (see the NESD chart in Figure 2). 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 internal plugins which in turn depend on the external plugins. In other words, release 3.0 introduced 'proxy' plugins for the external plugins, and this reduced coupling between Eclipse and third-party components. Additionally, one of the external plugins of release 2.1.3, org.apache.xerces, was removed.

Release 3.1 further reduced the dependency on external plugins, although it grew again in later releases. Although it seems that reducing coupling was a major concern during the milestones of release 3.3, the variations in this metric coincide with changes in the size and complexity metrics in those milestones. We don't include the corresponding charts in Figure 2 due to space constraints. We can hence say that in general Eclipse's architecture follows the 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 to reduce coupling, although it is growing in the latest releases.

However, contrary to the usual advice of increasing cohesion, we observed that it is continuously decreasing. Given that we should not expect a well designed architecture to evolve towards a complete graph, we also measured cohesion as the ratio between complexity and size, instead of the square of size, and show the result in Figure 2. With that definition, we observed that cohesion remained more or less constant throughout the 47 releases over the 6 years, i.e. the number of plugins grows at about the same rate as the number of dependencies.

Release 3.0 is an exception to this trend. A major restructuring increased cohesion by adding many dependencies while keeping the same number of plugins. Since maintenance releases don't change the architecture, this higher level of cohesion was kept until release 3.1. To sum up, the evolution of the Eclipse architecture seems to disregard the widely spread advice of aiming for increased cohesion. We return to this issue when addressing the last question, about the applicability of principles.

Before we move to the next question, let us revisit our observation [16] that, as the dark bars (K1) of Figure 2 indicate, Eclipse exhibits a non-negligible and stable architectural core of plugins and dependencies that are present both in releases 1.0 and 3.3.1.1. Depending on whether we count dependencies or plugins, the core accounts for 41% to 67% of the original architecture and 9% to 11% of the final architecture. The architectural core, which we didn't show in [16], is given in Figure 3. Note that there are only dynamic dependencies between documentation plugins. They do not contain any source code and hence have no static dependencies, but they use the extension point mechanism to document Eclipse in an incremental way. We can also see that some plugins, like pde (Plugin Development Environment), were retained throughout Eclipse's development, but none of their original dependencies remain.

4.2. Measuring changes

To check for any relationship between Martin's stability and van Belle's acuteness, we choose a period of releases and then compare the stability of plugins in the first release of that period with the acuteness of the same plugins over all releases in the period. The rationale is that, if there is a relation between stability and acuteness, then a plugin's resistance to change measured at some point in time should be reflected in the actual subsequent changes.

To make a meaningful experiment, we have to choose a long period that starts with a release that contains a good amount of plugins. It has to be long because most releases change the architecture very little or not at all, and it has to start with a relatively large release because instability is only measured for the initial release of the chosen period.

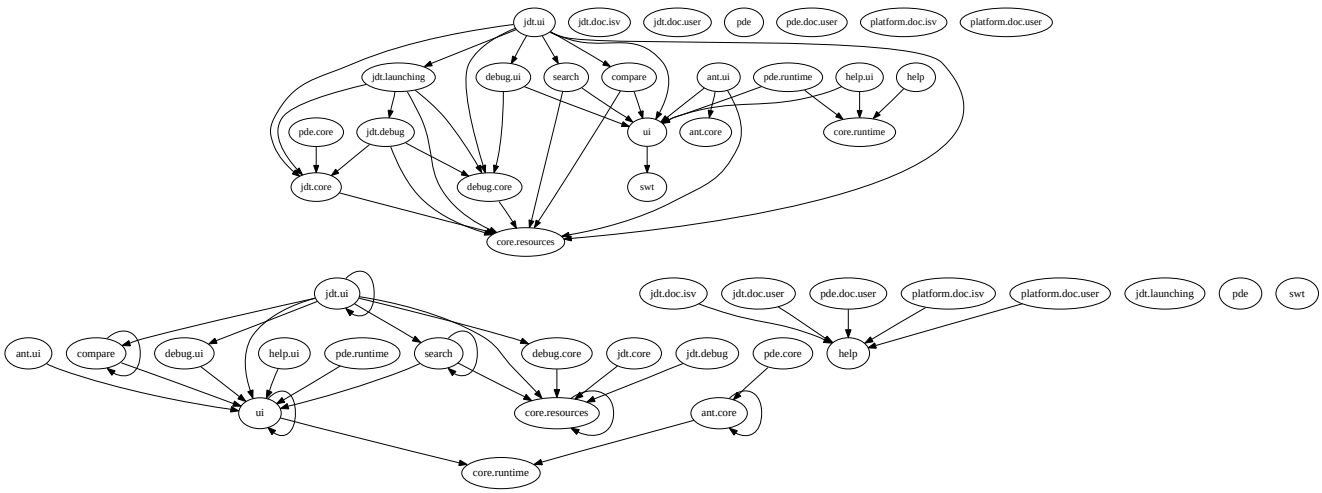

Figure 3. Plugins and static (top) and dynamic (bottom) dependencies kept since 1.0

Due to the growth of Eclipse, the earlier we set the initial release, the longer the analysed period will be, but the less plugins we have for analysis. To strike a balance, we chose the period comprising all releases from 2.0 on, which has 69 plugins.

Van Belle's argument for relating acuteness to stability is based on relating independence to likelihood and responsibility to impact (see Section 3). Hence, we also analysed those relationships. Although likelihood is a percentage, impact and therefore acuteness are not. We thus had to normalise those measures, dividing by the maximal values we obtained for each.

The top of Figure 4 plots for each considered plugin its dependency fanin (DFI) in release 2.0 against its change impact over the 45 succeeding releases. The plugins in the x-axis are ordered by increasing fanin. According to van Belle, responsible plugins (high DFI) should have high impact. We can observe this is not quite the case, although most irresponsible plugins do have indeed low or no impact.

The middle graph in the figure plots the dependency fanout (DFO) against likelihood, ordering the plugins by increasing fanout. According to van Belle, independent plugins (low DFO) should have low likelihood of change. We can observe that most plugins have a likelihood of change that coincides or is very close to their fanout.

The bottom graph plots acuteness against instability, ordering the plugins by increasing instability. According to van Belle, unstable plugins should have low change acuteness. Because of the good correlation between likelihood and independence, but the weak correlation between impact and responsibility, this graph shows a mixed picture. While there is some trend for decreasing acuteness with increasing instability, as desired, there are also many outliers.

Note that while we analysed all 69 plugins for the first two graphs, the third one only includes those 52 plugins that have a defined instability, i.e. where the denominator given by the coupling is non-zero. We calculated DFI, DFO and instability over $Eclipse_{2.0}$, i.e. including both static and dynamic dependencies, in order to reduce the number of plugins with undefined instability. We can however report that only one of those discarded plugins changed, i.e. modified the set of plugins depended upon (Section 2.3), during their lifetime. In other words, all 'stand-alone' (i.e. irresponsible and independent) plugins remain so, except org.eclipse.platform, which started to depend on other plugins in 3.0 and then changed in three further releases.

To sum up, we can answer question 4 of Section 3 by saying that there seems to be indeed some relation between stability and acuteness, although not strong enough for the static causal linkage given by the former to be used as predictor for the historic correlational linkage given by the latter. One should note that stability, being defined upon structural dependencies, only measures internal drives and obstacles to change. Although external reasons for change (e.g. new stakeholder requests) cannot be captured by any internal metric, we can nevertheless see that van Belle's and Martin's concepts capture some relationship between a system's structure and its modifications. The relation is stronger between independence and likelihood: components for which there is no internal drive to change, usually do not seem to have any external drive either, which may be somewhat surprising.

4.3. Principles

To check the Acyclic Dependency Principle (ADP), we consider module $Eclipse_r$ throughout all releases. The reason for taking the union of the two dependencies is that we

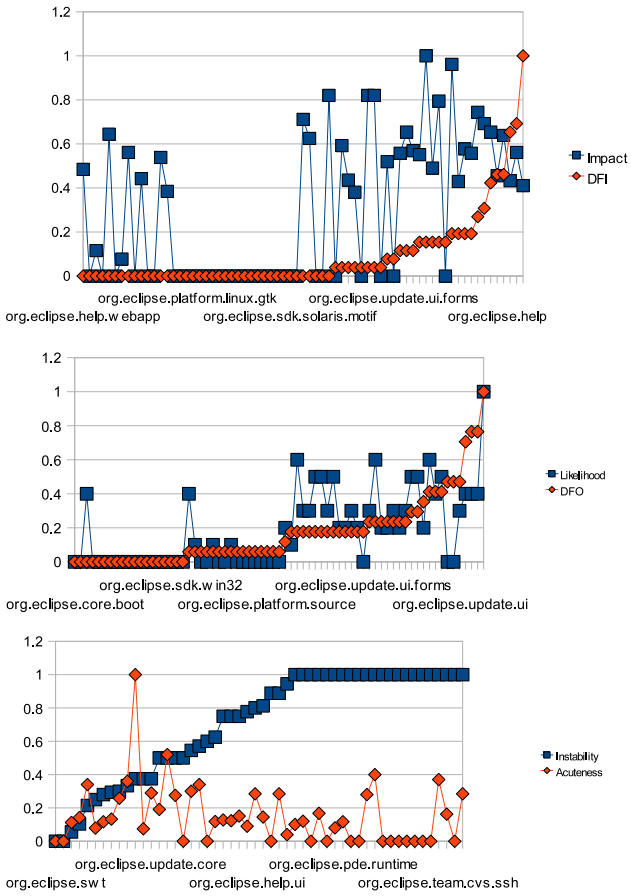

Figure 4. Martin's vs. van Belle's metrics

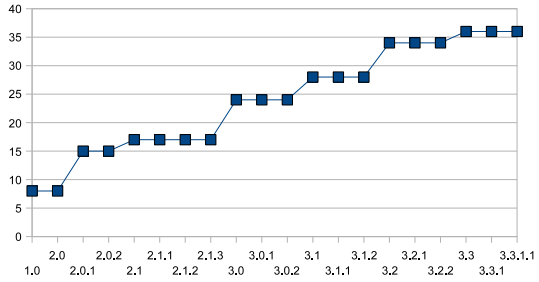

Figure 5. The evolution of self-cycles

want to detect any cycles where plugin A (in)directly depends statically on B and B (in)directly depends dynamically on A. We consider 'self-cycles', i.e. A depends on itself, to be harmless, as they indicate plugins using their own extension points.

As Figure 5 shows, our scripts report a growth of self-cycles that follows the same segmented pattern as plugins and dependencies, with alternating big and small increments. We also checked that between releases 3.1 and 3.3, increases only occurred during milestones, which is coherent with our observations in Section 4.1. More interestingly, the only cycle we found with length over 1 involved just plugins ui, ui.editors, and ui.workbench.editor. It appeared in release 2.1 and disappeared in release 3.0, although none of the plugins was deleted. In fact, one of the plugins was moved to a different feature. Thus, we can say that the Eclipse architecture follows the ADP and that the only existing cycle was removed during a major change to the architecture.

We checked the Stability Dependency Principle (SDP) by computing the instability of each internal plugin separately for $Eclipse_r^s$, $Eclipse_r^d$, and $Eclipse_r$ for all releases, but we show results only for major and maintenance releases due to space constraints. Figure 6 shows the number and ratio of dependencies that violate the SDP. As one can see, static dependencies (SD label) introduce far fewer violations than dynamic dependencies (DD label). The ratio of violations over all releases is 1-5% for static dependencies, and 9-17% for dynamic dependencies. There is no continuous growth in violations, with several reductions in the absolute number and a decreasing trend in the violation ratio of dynamic dependencies. We can therefore state that the Eclipse architecture follows the SDP to a very large degree and that restructuring efforts aim (consciously or not) to keep the violations within a small percentage range.

4.4. Validation

As we have seen, most of the principles and guidelines apply to the architectural evolution of Eclipse, with the prominent exception of cohesion, which deviates more and more from a complete graph. We feel this indicates that increased cohesion is indeed not a valid architectural principle for a system like Eclipse. As Eclipse can be deployed in a variety of combinations of subsystems, e.g. with or without the Plugin Development Environment, a tight cohesion among its components would reduce this flexibility.

The complete absence of static dependency cycles among plugins, except for a single temporary exception, may point to a possible threat to the validity of our results, because Melton and Tempero report thousands of cycles among Eclipse's classes, some of these cycles involving hundreds of classes [13]. One possibility is that the metadata does not capture all dependencies between plugins. Another possibility is that the granularity of plugins is such that cycles remain within the same plugin. Only future research can say which is the case.

5. Related Work

Ramil et al. [7] survey several studies on the evolution of open source systems and report how they relate to

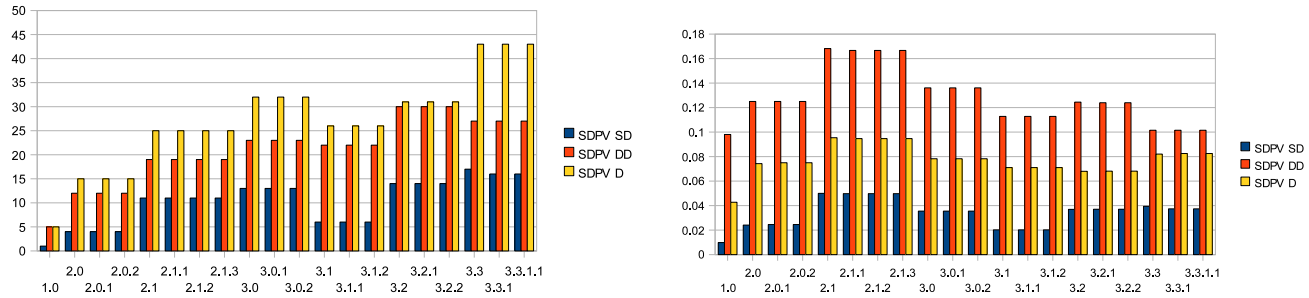

Figure 6. Absolute (left) and relative (right) number of dependencies violating the SDP

Lehman's software evolution laws and to studies of proprietary systems. Most of the surveyed studies analyse the source code, reporting for example the evolution of LOCs or of the McCabe complexity. Some systems have a uniform growth pattern, e.g. superlinear growth throughout their life, while other systems exhibit segments with different growth patterns, including no growth at all. As we have seen in Section 4.1, Eclipse's architectural evolution follows such a segmented growth pattern.

Wu et al. [18] put forward the hypothesis that software architecture controls the transitions between equilibrium and punctuation periods in the evolution. In equilibrium periods, changes are relatively minor and usually do not violate architectural constraints, while punctuation periods, which are relatively short, focus mostly on architectural changes in order to achieve stability in the long run. Wu et al. studied three open source systems written in C and analysed the monthly evolution of static dependencies between files as a way to approximate architectural changes.

In comparison, our study uses more reliable and higher-level (i.e. truly architectural) sources of information. Instead of using files as the basis of our study, which would largely correspond to Java classes, we use architectural components. Instead of 'reverse engineering' from the monthly evolution which releases correspond to which changes, we start from given releases. Moreover, we sample different kinds of releases so that we can check whether they correspond to particular periods. Indeed, maintenance releases and most release candidates represent equilibrium periods, and milestones are the punctuation periods.

Champaign et al. [6] have tried to correlate Martin's stability with likelihood, although they were unaware of van Belle's work. They used the Linux kernel as a case study. The elements considered were source code directories. Dependencies were obtained through an instrumented version of `make`, and a directory was considered to have changed if files were added or removed from it or if an existing file changed its size. The authors compared the likelihood computed over 499 releases with the stability computed over the last release analysed, and found no correlation at all.

We can think of two explanations for their result: they measured the stability at the end of the history instead of at the beginning, and they measured stability and likelihood at different levels: even deleting a single character in a comment counts as a change. In comparison, our study measures changes at the same architectural level as stability: we only consider modifications to the dependencies, which in turn define stability. Moreover, we measure stability of an element before its evolution, to check whether elements that are supposedly hard to change will indeed not change.

Our definition of module and its metrics is inspired by the work of Briand et al. [5], who defined a generic graph-based model for systems, modules and elements, and then propose several properties that measures of size, complexity, cohesion and coupling should follow, comparing their axiomatic framework with existing ones. We did not follow their definition of system and module as it assumes non-overlapping modules and leads to slightly convoluted properties, depending on whether the system is being measured through its elements or through its modules. However, our module metrics, if applied to the system model defined in [5], satisfy the stated properties.

Mens et al. [14] also analyse the evolution of Eclipse, but they rely on different data, mainly source and compiled code of major releases only, and have different aims, concentrating on an in-depth verification of Lehman's first, second and sixth law. Hou [8] looked at source code and release notes to investigate how the design of the Eclipse Java editor evolved. The main aim was to see how design evolves and accommodates additional features. One of the results was that having a stable model-view-controller architectural pattern was beneficial for evolutionary design. This reinforces (at a lower level of design) our finding of a non-negligible stable architectural core.

Eclipse is a complex system and there are many different ways of analysing and measuring it. Our study and [14, 8] thus offer complementary perspectives and we intend in future work to bring together such approaches.

6. Concluding remarks

Although many rational arguments have been given for a plethora of design principles and guidelines put forward throughout the years, there is not always clear empirical evidence about the usefulness of such advice. This paper presents a contribution to improve such state of affairs. We investigate design principles that have been argued to impact, directly or indirectly, on maintenance. We use a range of metrics to check the adherence or not to structural design principles. We use a case study representative of the large, complex, and long-lived systems for which such design principles are relevant to facilitate maintenance. Moreover, the case study was also required to have explicit architectural information, in order to ensure better accuracy of the results.

Although a single case study cannot prove or disprove the validity of a principle, there must be some value at the architectural level for the chosen principles, because Eclipse's history exhibits evidence of corrections to violations. Whether this is conscious or whether these principles are deeper manifestations of implicit good programming and low-level design practice, is to be seen. However, that is for the moment irrelevant for our purposes, because both cases reinforce the adherence to design principles.

A different question is whether such principles are useful for maintenance or not. This paper is just a contribution towards an answer that can only be established using more case studies. Moreover, to assess whether structural design principles have indeed an impact on the system's quality, other measures have to be taken (e.g. about bug reports). Both investigation paths are on our future work plans.

The observations that the ADP is violated by classes but followed by plugins, and that architectural cohesion does not increase, are further signs of the relevance of our question about the "right" level at which principles should be followed. We also could demonstrate some consistency between a causal and a correlational approach to measuring changeability, in spite of the many exogenous factors that cannot be captured by such metrics.

Eclipse is a successful system that is continuously kept useful to thousands of users and tool builders by incrementally accommodating new features, while keeping a flexible and extensible architecture. We believe the principles (ADP, SDP, etc.) and patterns (architectural core stability, segmented growth) that we have found in Eclipse's evolution contribute in part to its success. Hence, our findings could be useful to the practice of software development and maintenance, in particular for systems that require a loosely cohesive architecture with a stable yet extensible core.

A further contribution of this paper is the approach. The study is based on general concepts (like change sequence and module) associated with generic metrics (like size and likelihood), supported by a relational data representation in RSF. In this way, we and others can in the future study the evolution of systems at various levels of granularity, consistently compare measurement results, and exchange data.

References

[1] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

[2] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2):137–149, 2005.

[3] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[4] B. D. Bois, B. V. Rompaey, K. Meijfroidt, and E. Suijs. Supporting reengineering scenarios with FETCH: an experience report. *Electronic Communications of the EASST*, 8, 2008. Selected papers from the 2007 ERCIM Symp. on Software Evolution.

[5] L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Trans. Software Eng.*, 22(1):68–86, 1996.

[6] J. Champaign, A. Malton, and X. Dong. Stability and volatility in the Linux kernel. In *Proc. 6th Intl. Workshop on Principles of Software Evolution*, pages 95–102. IEEE Computer Society, 2003.

[7] J. Fernández-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical studies of open source evolution. In *Software Evolution*, chapter 11, pages 263–288. Springer Verlag, 2008.

[8] D. Hou. Studying the evolution of the eclipse java editor. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange*, pages 65–69. ACM, 2007.

[9] R. C. Martin. Granularity. *C++ Report*, 8(10):57–62, Nov.-Dec. 1996.

[10] R. C. Martin. Large-scale stability. *C++ Report*, 9(2):54–60, Feb. 1997.

[11] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, Jan. 2007.

[12] H. Melton. On the usage and usefulness of OO design principles. In *Companion to the 21st OOPSLA*, pages 770–771. ACM, 2006.

[13] H. Melton and E. Tempero. An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[14] T. Mens, J. Fernández-Ramil, and S. Degrandsart. The evolution of eclipse. In *Proc. 24th Int'l Conf. on Software Maintenance*, 2008. To appear.

[15] T. van Belle. *Modularity and the Evolution of Software Evolvability*. PhD thesis, University of New Mexico, 2004.

[16] M. Wermelinger and Y. Yu. Analyzing the evolution of Eclipse plugins. In *Proc. 5th Working Conf. on Mining Software Repositories*, pages 133–136. ACM, 2008.

[17] K. Wong. *The Rigi User's Manual, Version 5.4.4*, June 1998.

[18] J. Wu, C. Spitzer, A. Hassan, and R. Holt. Evolution spectrographs: visualizing punctuated change in software evolution. In *Proc. 7th Intl. Workshop on Principles of Software Evolution*, pages 57–66, 2004.