Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Assessing Architectural Evolution: a Case Study

Michel Wermelinger · Yijun Yu · Angela Lozano · Andrea Capiluppi

the date of receipt and acceptance should be inserted later

Abstract This paper proposes to use a historical perspective on generic laws, principles, and guidelines, like Lehman's software evolution laws and Martin's design principles, in order to achieve a multi-faceted process and structural assessment of a system's architectural evolution. We present a simple structural model with associated historical metrics and visualizations that could form part of an architect's dashboard.

We perform such an assessment for the Eclipse SDK, as a case study of a large, complex, and long-lived system for which sustained effective architectural evolution is paramount. The twofold aim of checking generic principles on a well-know system is, on the one hand, to see whether there are certain lessons that could be learned for best practice of architectural evolution, and on the other hand to get more insights about the applicability of such principles. We find that while the Eclipse SDK does follow several of the laws and principles, there are some deviations, and we discuss areas of architectural improvement and limitations of the assessment approach.

1 Motivation

The first law of software evolution (continuous change) states that systems "must be continually adapted else they become progressively less satisfactory", and the sixth law (continuous growth) states that the "functional content must be continually increased to maintain user satisfaction over their lifetime" (Lehman et al 1997). The success of a system—and we mean continuous success, not the equivalent of a musical one hit wonder—hence depends in part on sustaining its evolution for a period of time.

M. Wermelinger, Y. Yu
Computing Department & Centre for Research in Computing, The Open University, UK
E-mail: {m.a.wermelinger, y.yu}@open.ac.uk
A. Lozano
Département d'Ingénierie Informatique, Université catholique de Louvain, Belgium
E-mail: angela.lozano@uclouvain.be
A. Capiluppi
School of Computing, Information Technology and Engineering, University of East London, UK
E-mail: a.capiluppi@uel.ac.uk

That evolution is a key aspect has been realized for a long time: throughout the years, a vast amount of advice, guidelines and techniques has been proposed in the literature to facilitate the eventual changes to a system's requirements, design and implementation. Advice varies in the way future changes are made easier, e.g. through usage of language-specific features (Bloch 2001), paradigm-specific catalogues of generic solutions to recurring problems (Ben-Ari 1982; Gamma et al 1995), or general principles (Martin 1997).

Approaches to support evolution are particularly beneficial for systems that are large and complex, and hence difficult to maintain, and have many users they have to continuously satisfy. Such systems 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 (Medvidovic et al 2007).

Given the importance of evolution and architecture for many systems, this paper focusses on architectural evolution, which can be seen from two perspectives: on the one hand, architectural design should facilitate system evolution; on the other hand, the evolution process must be aware of the system's architecture and support its conceptual role. Our overall goal is to help contribute towards improved understanding and practice of architectural evolution. Towards that goal, the paper has two aims: first to provide an approach to assess the architectural evolution of a project, and second to provide a pedagogical exemplar of good architectural evolution.

The assessment approach to be detailed in the following sections is based on a set of questions together with tool-supported quantitative measurements and visualizations, which allow developers, project managers or researchers to see architectural evolution trends and thereby answer the assessment questions. The questions are largely drawn from design guidelines and principles proposed in the literature to facilitate evolution and maintenance. Although assessment has ultimately to be qualitative, as the specific context of the project has to be taken into account, our approach allows for the assessment to be based on rather objective and quantifiable adherence to independently proposed advice for best practice.

To validate whether our approach is fit for its intended purpose, we apply it to a case study. We did a so called information-oriented selection, in which "cases are selected on the basis of expectations about their information content" in order to "maximize the utility of information from small samples and singles cases" (Flyvbjerg 2006). In particular, we chose the case study to be both *paradigmatic*, i.e. representative of a certain class of systems, and *critical*, i.e. likely to allow certain general inferences to be made (Flyvbjerg 2006). We chose what Flyvbjerg calls a 'most likely' critical case, which for our purposes means a case study that is likely to follow the evolutionary design guidelines and principles we included in the assessment. As Flyvbjerg points out, 'most likely' cases are well suited for the falsification of hypotheses and propositions, in the sense of Popper (1959). If it turns out that the case study follows some assessment guidelines and principles but not others, one can argue the latter may not be as determinant for achieving good architectural evolution as the former, at least for the class of systems the case study represents.

Besides helping refute hypotheses, case studies are also useful as rich narratives of concrete and context-dependent knowledge, which Flyvbjerg (2006) argues is more valuable than the vain search for non-existent predictive theories and universals. Eysenck (1976) wrote that "sometimes we simply have to keep our eyes open and look carefully at individual cases—-not in the hope of proving anything, but rather in the hope of learning something", and Kuhn (1987) argues that a discipline without systematic production of exemplars is an ineffective one. Melton (2006) 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. In later work, Melton and

Tempero (2007) took a single principle and checked it across many different systems. Another possibility is to analyse various, possibly related, principles on the same system, as we do in order to assess its architectural evolution.

The application of our assessment approach to a paradigmatic and 'most likely' critical case hence addresses both aims of this paper: it helps validate the approach and it provides an in-depth account of an exemplar from which lessons can be learned, as done in other disciplines, e.g. business management, which is heavily based on case studies of individual organizations and products.

In this section we motivated and outlined the research scope and approach of this paper. The next section presents the principles and guidelines chosen for assessment and reformulates our aims as research questions, Section 3 describes which data was collected and how, Section 4 reports the results observed, Section 5 analyses the results, discusses threats to validity and reflects on lessons learned, Section 6 compares this to other work, and Section 7 provides concluding remarks.

2 Questions

There are three separate but related issues involved in our work. First, there is the actual task of assessing architectural evolution. This will be guided by so called assessment questions, presented in the next subsection. Second, we must question whether the design principles, guidelines and measurements used in the assessment are fit for purpose. This will be formulated as a research question (Section 2.2) or discussed as threats to validity (Section 5.3), and covers the first aim of proposing a suitable approach for assessing architectural evolution. Third, the application of the assessment approach to a carefully chosen case study should yield valuable lessons for architectural evolution. This addresses the paper's second aim and is also formulated as a research question.

2.1 Assessment questions

In the following we present a baker's dozen of questions to guide the architectural evolution assessment. The questions were selected to satisfy the following requirements, but we do not claim the list of questions to be comprehensive.

- The questions should be related to evolution, e.g. by looking at historical trends. Questions that can be asked of a single snapshot of the system should be about an architectural issue that affects evolution.
- The questions can be asked about any system, independently of domain, technology, architectural languages, styles and patterns.
- The questions allow their answers to draw on automated quantitative measurements of the system's architecture-related data in the development repository.
- The questions should be diverse (e.g. not just about Lehman's evolution laws) in order to get a multi-faceted picture of a system's architectural evolution. In particular, questions should cover the two perspectives mentioned in the previous section: how architectural design can facilitate evolution (e.g. questions 9–13 below) and how the evolution process can support the architectural design (e.g. questions 2 and 6).
- The questions should preferably be based on guidelines and principles already proposed in the literature, to avoid our own bias.

Note that only the last requirement is optional, all others are mandatory for each question. The rationale for the second and third requirements is to allow for tool-supported continuous assessment across a range of projects, e.g. via an architectural evolution 'dashboard' embedded within the development environment. The general quantitative approach of this work is then complemented by questions that are system-specific, qualitative or require non-architectural data, like 'what is the impact of using off-the-shelf component X on the architectural evolution of system Y?', 'is quality X of system Y kept by architectural changes?' or 'what is the impact of architectural change on maintenance cost?'.

We start the list of assessment questions with one about the architectural style and its changes:

1. Does the architecture follow any style, like pipes and filters? If so, does the style keep the same or does it change over time?

As important is what does *not* change:

2. Is there any stable (i.e. unchanged) architectural core around which the system grew?

To look at particular trends in the changes, we start with some evolution principles as formulated in Lehman et al (1997), and already studied for other systems (Fernández-Ramil et al 2008).

3. Does the architecture's size follow Lehman's 6th law of software evolution (continuous growth)? Does growth follow any of the patterns observed for other systems?
4. Does the architecture follow Lehman's 2nd law (increased complexity)?
5. Is there any evidence of restructuring work aimed at reducing growth and complexity?

Besides looking at trends of *how* the architecture changes, it may also be illuminating to see *when* it changes.

6. Is there any systematic architectural change process?

Two of the key principles of structured software design is low coupling and high cohesion, whose purpose is to achieve design stability (Stevens et al 1979). Following a historical perspective, we look at their trends:

7. Does the architecture's cohesion increase?
8. Does the architecture's coupling decrease?

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 structure components and their dependencies, in a way that facilitates changes to the system. It therefore makes sense to include them in our evolutionary analysis of structure.

The *Acyclic Dependency Principle* (ADP) states that the dependencies should form a directed acyclic graph (Martin 1996). The rationale is to facilitate release management and allocation of work to developers. Mutual dependencies, and the increased change propagation they cause, makes it harder to independently release the various components and to coordinate multiple developers.

9. Does the architecture follow the ADP?

The basic principle which is then refined by several of Martin's proposals is the *Open Close Principle* (OCP) coined by Meyer (1988). It states that entities should be open for extension but closed for modification, i.e. the ideal way of changing software should be by adding code instead of by modifying already working code. By attempting to answer the previous questions, we can see whether the architecture is changing mostly through extensions rather than through modifications inside the components.

10. Does the architecture follow the OCP?

The OCP is an ideal that is hard to achieve in practice, as modifications are often unavoidable. To facilitate change, Martin (1997) introduces the notion of *stability*, meaning resistance to change, and the *Stable Dependencies Principle* (SDP), stating that dependencies should be in the direction of stability, 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 (the number of outgoing dependencies) divided by the total number of dependencies, i.e. the sum of the element's fanout and fanin (the number of incoming dependencies). 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 comment on elements that are irresponsible and independent, and therefore have undefined instability.

11. Does the architecture follow the SDP?

Martin (1996) takes the OCP's idea of extension further, proposing the *Reuse/Release Equivalence Principle* (REP), which states that the unit of reuse should be the unit of release, i.e. effective reuse consists not of copying fragments of code but instead of using code as a pre-packaged product that is independently released. One can argue that component-based development is one manifestation of the REP. In the same article, Martin states two principles that can be seen as a corollary of the REP. One is the *Common Reuse Principle* (CRP), reinforcing that all elements (e.g. classes) inside the same reuse unit have to be reused together. Hence, all elements that collaborate together should be put inside the same reuse unit. The other is the *Common Closure Principle* (CCP), stating that a reuse unit closes (in the OCP sense) all elements inside it to the same kinds of changes. The rationale is that changes that cross-cut reuse units force new versions of all those units to be released (REP). By looking for independent release units within the architecture and by doing an historical analysis of changes, we should be able to address two more questions:

12. Does the architecture follow the CRP?
13. Does the architecture follow the CCP?

2.2 Research questions

The previous assessment questions, together with the tool-supported measurement and visualization techniques that we propose in Sections 3 and 4 to answer them, form the approach we wish to propose to fulfil the paper's first aim. However, in proposing such an approach, we must be confident it is fit for purpose. Such confidence might be undermined in various ways.

First, the gathered data and the proposed metrics may not capture the design guidelines and principles mentioned in the assessment questions. That is an issue of construct and internal validity and will be dealt in Section 5.3.

Second, even though the metrics may be appropriate for the principles, the latter may not be meaningful at the architectural level because most of the design concepts and principles were developed for structuring lower level abstractions, like classes and methods, and their relationships. This issue will be addressed by comparing our work with another evolutionary analysis of the same case study, but done at a lower design and implementation level (Mens et al 2008).

Third, even if the principles can be stated at the higher architectural level, they may not be very relevant for architectural evolution practice, i.e. they may not capture the developer's intents and concerns when designing for evolution and when evolving an architecture. In other words, sustainable and effective architectural evolution might be achieved without following any (or most) of the principles and guidelines used in our assessment questions. This issue will be addressed by validating our analysis of the case study with its developers.

Our first research question is thus:

RQ1 Are the assessment questions and the tool-supported measurements and visualizations fit for the purpose of assessing the architectural evolution of a system? In particular, are the principles and guidelines covered by the questions
- meaningful at the architectural level?
- relevant for architectural evolution practice?

A positive answer to these questions will mean the paper's first aim has been achieved.

We have chosen the Eclipse Software Development Kit (SDK) as the case study to help answer the previous research question. The SDK is the core of the Eclipse framework and has been evolving for several years. Hundreds of tools and add-ons have been built, by a large community of developers, on top of the SDK. The SDK is thus a critical case of well carried out architectural evolution because otherwise the applications built on top of it would have problems or require changes with every new release of the SDK, which in turn would alienate developers. While other aspects, like the SDK's features and documentation, may also play an important role in Eclipse's continuously growing and thriving eco-system of applications and developers, the architectural qualities and the evolutionary process of the SDK are undoubtedly major factors, because the SDK is the foundation on which the whole range of Eclipse projects and third-party applications stands or falls.

The Eclipse SDK is also a paradigmatic case study, representing open source projects that are led by organizations with full-time developers on the project, like MySQL and JBoss, and representing extensible systems based on plug-in architectures, like the Mozilla projects. These two characteristics may bear on architectural concerns and the evolutionary process in such projects. Any lessons learned from the architectural evolution of the Eclipse SDK may therefore have wider applicability.

Once the first research question has been answered, we can discard those assessment questions that were found to be not fit for purpose, and distil from the remaining ones the practices that have led to the sustainably effective architectural evolution of the case study, thus answering our second research question:

RQ2 What can we learn from the architectural evolution of the Eclipse SDK?

If several and diverse insights can be obtained, the paper's second aim of presenting the SDK as an exemplar will be achieved.

3 Data collection

Having defined the assessment and research questions we want to answer, we now detail the case study, define the historic structural measurement approach, and summarise how it was implemented.

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 (except for release 1.0) provides one or more high-level *features*, which are used by Eclipse's update manager to allow users to selectively and incrementally upgrade their installation of Eclipse. Each feature may be composed of other, more specialised, features, i.e. features are organised hierarchically. For example, feature org.eclipse.sdk has sub-feature org.eclipse.platform which in turn includes feature org.eclipse.rcp (Rich Client Platform).

Each feature is implemented by a set of *plugins*, Eclipse's components. For example, in build 3.3.1.1, the feature org.eclipse.platform is implemented by more than 70 plugins, including org.eclipse.core.boot, org.eclipse.compare, and org.eclipse.platform, to name a few. Notice that features and plugins may have the same name. In the remaining of the paper, we omit the org.eclipse prefix from feature and plugin names. Each plugin may *depend* for its compilation on Java classes that belong to other plugins. In the remaining of the paper, we say that plugin X *statically depends* on plugin Y if the compilation of X requires Y. 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. We will say that X *dynamically depends* on Y if X requires an extension point provided by Y. Note that the dynamic dependencies are at the architectural level; they do not capture run-time calls between objects. 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.

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, the main ones being the *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, going through some *milestones* followed by some *release candidates*. For example, release 3.1 was followed by six milestones of release 3.2 (named 3.2M1, 3.2M2, etc.) and seven release candidates (3.2RC1, 3.2RC2, etc.), culminating in minor release 3.2.

Figure 1 shows that builds can be organised in chronological or logical order. Logical order is indicated by solid arrows: a 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 omitted builds.

For the purposes of analysing architectural changes, it makes more sense to follow a logical rather than the chronological order. For this paper we analysed two logical 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).

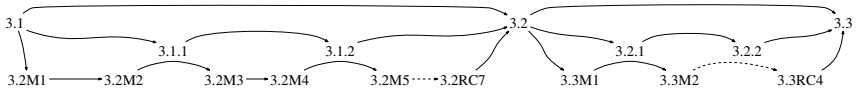

Fig. 1 Chronological and logical sequences of some of the analysed builds

3.2 The Data Model

In order to perform the architectural evolution assessment in a systematic and generic way, whilst avoiding certain threats to validity, we adopt the structural model and axiomatic measurement framework of Briand et al (1996), with the corrections presented by Briand et al (1997). We first briefly summarize the framework—formal definitions can be found in the cited papers—and then we describe how we apply it for our purposes. We justify the adoption of the framework when discussing the threats to validity of our work (Section 5.3).

3.2.1 The measurement framework

A *system* is a set of *elements* and their *relationships*, represented as a directed graph, i.e. elements are nodes and relations are directed arcs. For example, nodes can represent individual statements, methods, or classes, and arcs can represent control flow or inheritance. A *module* is a subset of the elements, and all the existing relationships among that subset. For example, if elements are methods, then modules might represent classes; if classes are elements, then packages are modules. Two modules are said to be *disjoint* if there is no element (and hence no relation) belonging to both, and *unrelated* if there is no relation between an element of one module and an element of the other. In a *modular system* each element belongs to exactly one module, i.e. modules partition the set of elements. Relations are classified as *intra-module* (i.e. between two elements of the same module) or *inter-module* (i.e. between two elements of two different modules). Different kinds of measures can be taken on systems and modules, as long as they obey the following axioms.

The *size* of a module or system is a non-negative value. It is zero if there are no elements (null value axiom). The size of a modular system is the sum of the sizes of its modules (additivity axiom). A consequence of these properties is that adding elements to a system doesn't decrease its size (monotonicity axiom).

The *complexity* of a module or system is a non-negative value. It is zero if there are no relationships (null value). The complexity doesn't depend on the direction of the relations, e.g. if we replace the relation 'calls' by relation 'is-called-by' (symmetry). The complexity of a system is at least the sum of the complexities of any two of its modules that have no common relationships. If a system is the union of two disjoint and unrelated modules, then the system complexity is the sum of the module complexities (additivity). A consequence of these properties is that adding relations to a system doesn't decrease its complexity (monotonicity).

Cohesion aims to capture the tightness in which related elements are encapsulated together in a module. The cohesion of a module or modular system can only range from zero to a fixed maximal value, regardless of how much the module or system grows, i.e. cohesion is non-negative and normalized. Cohesion is zero if there are no intra-module relations (null value), because there is no evidence the elements should be encapsulated together in the same module. Adding intra-module relations is further evidence the elements in that module belong together and hence does not decrease the cohesion (monotonicity). The cohesion of a module that is the union of two unrelated modules is not greater than the maximum

cohesion of the two original modules (non-additivity). Therefore, if two unrelated modules are merged in a modular system, the system's cohesion doesn't increase (monotonicity), because unrelated elements are being grouped into the same module.

The *coupling* of a module or modular system is a non-negative value. It is zero if there are no inter-module relations (null value). Adding inter-module relations does not decrease coupling (monotonicity). The coupling of a module obtained by merging two modules is not greater than the sum of the original modules' coupling; it is exactly the sum if the two modules are unrelated (additivity). The coupling of a modular system does not increase if two of its modules are merged; it stays the same if the two modules are unrelated.

3.2.2 Applying the framework

For our purposes, the system is the Eclipse SDK, the elements being plugins and the relations being static and/or dynamic dependencies. To define the model of a single Eclipse build, we start with the set of plugins, and, for each plugin, the sets of provided and required extension points and the set of plugins on which it statically depends. This allows us to establish the dynamic and static dependencies between plugins, and to compute the missing (i.e. needed but undefined) and unused (i.e. defined but not needed) entities, in order to provide an indication of how self-contained and open the system is. For example, an extension point is missing if it is required but not provided, and unused if it is provided but not required within the SDK. We haven't found any missing plugins or extension points, and report on unused extension points when addressing the OCP.

We modularise the SDK in three different ways. In the first modularisation the system consists of two modules: the *internal plugins IP*, all those named org.eclipse.*, and the *external plugins EP*, like org.apache.ant. We ignore plugins whose name ends in source because they don't provide added functionality; their inclusion would therefore invalidate the analysis of Lehman's 6th law (see start of Section 1). Each of those plugins 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.

Since the name of a plugin determines whether it is an internal or external plugin, this in turn allows us to compute the sets ISD, ESD, IDD, EDD of internal and external static dependencies, and internal and external dynamic dependencies, respectively. Internal dependencies are those among the internal plugins, while external dependencies are those between an internal and an external plugin. We focus on the internal plugins module and its intra- and inter-module dependencies, while ignoring intra-module dependencies of the external plugins module, because those plugins are just wrappers of third party code. We compute the sets of internal and external dependencies as $ID = ISD \cup IDD$ and $ED = ESD \cup EDD$, respectively.

We define two other modularisations to check the CRP and CCP: features and subsystems (to use a term of Mens et al (2008)), i.e. we group plugins from the user and developer perspective, respectively, to then assess which modularization is more effective to encapsulate reuse and changes (Sections 4.9 and 4.10). We just consider top-level features and subsystems because the structural model and its axioms only allow one level of system decomposition. To be more precise, in the feature modularisation, the module corresponding to top feature F groups all plugins that implement F or one of its sub-features, while in the subsystem modularisation, the module corresponding to subsystem S groups all plugins named org.eclipse.S.*. For example, plugins org.eclipse.ui and org.eclipse.help.ui both implement top-level feature org.eclipse.ui. Hence both plugins will be part of module ui in the feature modularisation, but in the subsystem modularisation, one will be in module ui

and the other in module help. Finally, each dependency between a pair of plugins is classified as intra- or inter-module dependency, based on the module(s) to which the two plugins belong.

The three types of modularisation methods—internal vs. external plugins, top-level features, and top-level subsystems—partition the set of plugins and hence we have a modular system as required by the measurement axioms. We now define the metrics directly on those entities referred to by the axioms. This not only ensures we adhere to the axioms, it also makes our metrics as simple and as general as possible.

The size axioms are based on elements; we thus define the size metric as the number of elements, i.e. the number of plugins, in our case. For example, the size of the internal plugins module is $|IP|$.

The complexity axioms are based on relations, hence we define the complexity metric as the number of relations, i.e. the number of dependencies. For example, the complexity of the internal plugins module can be $|ID|$, $|ISD|$ or $|IDD|$, depending on which dependencies (overall, just static, or just dynamic) one wishes to consider.

The coupling axioms are based on inter-module relations, hence we define the coupling metric as the number of inter-module relations. For example, the coupling of the internal plugins module is $|ED|$, $|ESD|$ or $|EDD|$, depending on which dependencies to the external plugins module one wishes to consider.

The cohesion axioms are based on intra-module relations, as evidence for elements to be encapsulated together in the same module. We hence define cohesion as the ratio between the actual and the potential (i.e. maximally possible) intra-module relations. The rationale is that the existence of all possible e^2 directed intra-module relations among the module's e elements is the strongest evidence for those elements to belong together in that module, which thus would have maximal cohesion. In other words, $cohesion(M) = d/p^2$ for a module M with p plugins and d dependencies among them (i.e. intra-module dependencies). The number d ranges from zero, for a discrete graph, to p^2, for a complete graph, and therefore cohesion is always in the interval [0,1], satisfying the non-negativity and normalization axioms. For our case study, the maximal value is theoretically possible if each plugin dynamically depends on every other plugin and itself. The null value and monotonicity axioms are trivially satisfied by definition: if there are no dependencies, d is zero, and so is the cohesion, and if dependencies are added (to a fixed set of plugins), d increases and so does the cohesion. To check the non-additivity axiom, we assume we have two unrelated modules M_1 and M_2 such that, without loss of generality, $cohesion(M_1) \leq cohesion(M_2)$, i.e. $d_1/p_1^2 \leq d_2/p_2^2$, which can be rewritten as $d_1 p_2^2 \leq d_2 p_1^2$ (A). The cohesion of the merged module will be $(d_1+d_2)/(p_1+p_2)^2$ because the axiom's pre-condition (absence of inter-module dependencies between the two modules to be merged) means there are no further intra-module dependencies in the resulting merged module. The non-additivity axiom then states that $(d_1+d_2)/(p_1+p_2)^2 \leq d_2/p_2^2$, which can be rewritten as $d_1 p_2^2 \leq d_2 p_1^2 + 2d_2 p_1 p_2$ (B) through simple algebraic manipulations. Inequality B follows directly from inequality A and the non-negativity of the number of plugins and dependencies. Our cohesion metric thus adheres to all cohesion axioms.

Finally, to address evolution, we repeat the model construction and metric computations for each build. We allow the user to define a sequence $b_1, b_2, \ldots$, in any order they wish, of a subset of all builds. As said in the previous section, we looked at two sequences, one with b_1=1.0, b_2=2.0, …, b_{26}=3.5.1, and the other with b_1=3.1, b_2=3.2M1, …, b_{30}=3.3. Given a sequence, we can compute the architectural differences and similarities between successive builds in the sequence. For example, if P_n is the set of plugins in a given module or system at build b_n, then we can compute the sets of plugins that b_n added ($P_n \setminus P_{n-1}$), deleted

($P_{n-1} \setminus P_n$), and preserved ($P_n \cap P_{n-1}$). Because we are also interested in the unchanged architectural core, we divide the preserved plugins into those kept from the first build in the sequence ($P_n \cap P_{n-1} \cap P_1$), and those from the previous build ($P_n \cap P_{n-1} \setminus P_1$). We do likewise for extension points and static and dynamic dependencies.

3.3 The tool infrastructure

We developed a pipeline of small scripts that operate on text files, to first extract the data, then compute the metrics, and finally visualize the results.

First, we downloaded, for each of the builds we analysed, the source code of the whole SDK from http://archive.eclipse.org or its mirrors. We wrote some shell, AWK and XSLT scripts that extract the basic architectural information mentioned in the previous section: for each feature there is a metadata file `feature.xml` that lists the feature's sub-features and the plugins that implement it, and for each plugin there are two metadata files (`plugin.xml` and, since release 3.0, `MANIFEST.MF`) that list the plugin's required and provided extension points and the plugins it depends on. The output of this fact extraction process is a set of features, plugins and extension points and their basic relations, e.g. the hierarchical containment relation between features and the 'provides' relation between plugins and extension points. Since a set can be represented as a unary relation that holds for each element that is a set member, we represent all facts as relations in the popular text-based Rigi Standard Format (RSF) (Wong 1998).

Next, we used the relational calculator Crocopat (Beyer et al 2005), which operates on RSF files, to compute: derived relations, like the transitive closure over dependencies, in order to detect cycles for the ADP, and the union of all plugins that implement a given feature or one of its sub-features, in order to obtain the feature modules; the metrics for each build; the differences and similarities between consecutive builds, given an extra successor relation that defines the sequence of builds as described in the previous section. The result of all these computations are further relations in RSF, e.g., one binary relation per metric stating the value of the metric at each build.

Finally, the third phase of the pipeline uses Crocopat and AWK to generate the input files for various visualization tools. We use graphviz[1] to display architectural structures (as in Figure 3), CCVisu (Beyer 2008) to show clusterings (as in Figure 15), and Google Interactive Charts[2] to plot the evolution of metrics with line and bar charts embedded in web pages. The input files for the charts are comma separated value files, generated with Crocopat for each build sequence analysed, that we uploaded to Google Spreadsheets.

The rationale for our approach was to build on top of existing tools and data formats to avoid reinventing the wheel while providing a flexible, light-weight and interoperable tool infrastructure.

For example, using Google tools has several advantages. First, the data is made public in various formats (HTML, OpenOffice, Excel) without additional effort from us. Second, the charts are large and interactive, allowing the reader to click on the data points and see the exact values, instead of just perceiving generic trends from small, static, and grey charts on 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, and upload the new spreadsheet.

[1] http://www.graphviz.org/

[2] http://code.google.com/apis/visualization/interactive_charts.html

The tool's flexibility and interoperability is 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 (Bois et al 2008), and to modify the 'back-end' fact extraction phase to handle other systems besides Eclipse.

Indeed, the fact processing and visualization phases only require minimal information in RSF: the components and their provided and required services (plugins and extension points, in the case of Eclipse); the different kinds of relationships (static and dynamic dependencies in our case); and all this for each snapshot to be analysed. In particular, it is irrelevant for those two phases whether the snapshot structural models were extracted from the metadata of various source tarballs (as in our case), or from applying an architectural reverse engineering tool on a single software configuration management repository, or even from system specifications written in some architecture description language. The structural model presented in the previous section is so general, due to its graph-based nature, that more expressive ADLs can be mapped to it, thus allowing our approach to be applied even in cases where code is not available, as long as architectural models are.

The Eclipse metadata is such an architectural model, leading to very efficient and accurate fact extraction, without requiring code analysis to reverse engineer a possibly not very reliable architecture. We fully agree with Alex Wolf's argument, in his WICSA'09 keynote, that configuration files are an under-explored source of architectural information.

A second source of efficiency is the small size of the database of extracted facts (less than 90,000 tuples for the 53 builds analysed) because there are many fewer elements and relationships at architectural level than at implementation level. Beyer et al (2005) applied Crocopat to the Eclipse 2.1.2 graph of 7,081 classes and their 59,344 call dependencies, whereas each build's architectural graph of Eclipse plugins and their dependencies has only hundreds of nodes and arcs (Section 4). Our tool chain thus remains efficient even though we store the data in RSF text files, instead of using an SQL-based relational database manager. Moreover, Beyer reports that Crocopat is much more efficient than MySQL for computing cycles and transitive closures of relations, which are important features for our work and for structural analysis of design in general.

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, following the order of the questions in Section 2. The charts presented in this section (and additional ones) can be interacted with on the 'web companion' to this paper (see appendix A).

To show the evolution of the metrics over the two build sequences, we use mostly stacked bar charts, with each bar segment showing a particular subset of the total number of items (plugins, extension points or dependencies). The segments are stacked, from bottom to top, as follows: unforced deletions, forced deletions, kept items (i.e. since the first build 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 electronic version of this paper and on the web companion 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 Architecture

We have visualised the architecture of each major and minor release with graphviz and noticed that Eclipse largely follows a layered architectural style. For each release, we calculated the number of layers and how many plugins were in each layer. The root layer (layer 0) contains all plugins with fanin zero. A plugin p is in layer n if $n-1$ is the maximum layer of all the plugins that depend on p. Figure 2 uses alternatively dark blue and red for each layer, from the root layer at the bottom, to the deepest layer at the top of the chart. The width of each colour band is proportional to the number of plugins in that layer. Eclipse had 6 layers until release 3.0, 13 in release 3.0, 11 in 3.1, 15 in 3.3 and 16 in 3.4. Most of the growth happens, as it should according to Martin's principles, in the layer with the highest instability values: the root layer, where all plugins are by definition irresponsible. Variations in the size of other layers are relatively small.

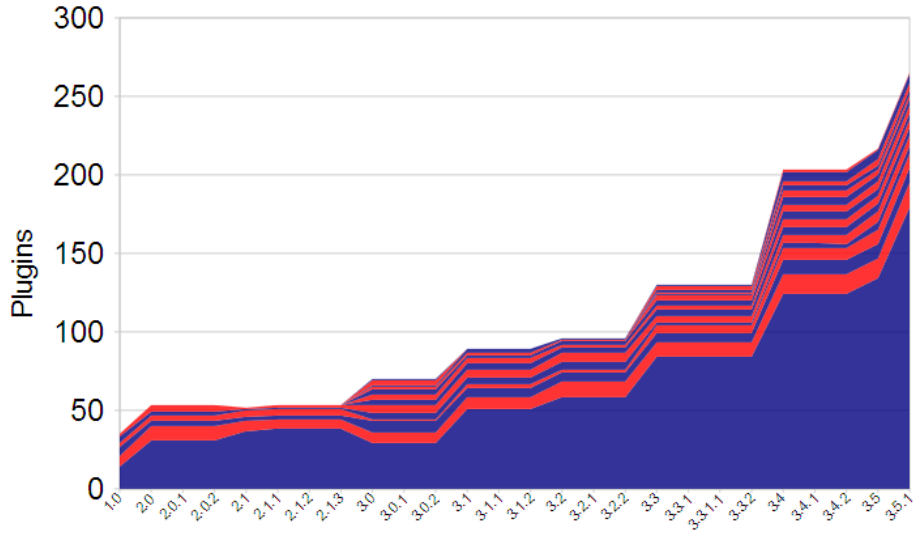

Fig. 2 Evolution of the architecture's layers

Eclipse exhibits a non-negligible and stable architectural core of internal plugins and dependencies that have been kept since release 1.0 and are shown in Figure 3, with dotted lines denoting dynamic dependencies and solid lines representing static dependencies.

The core follows the same layered style as the overall architecture and accounts for 48% of the overall dependencies and 69% of the plugins of the original architecture and 6% of the overall dependencies and 9% of the plugins of the final architecture.

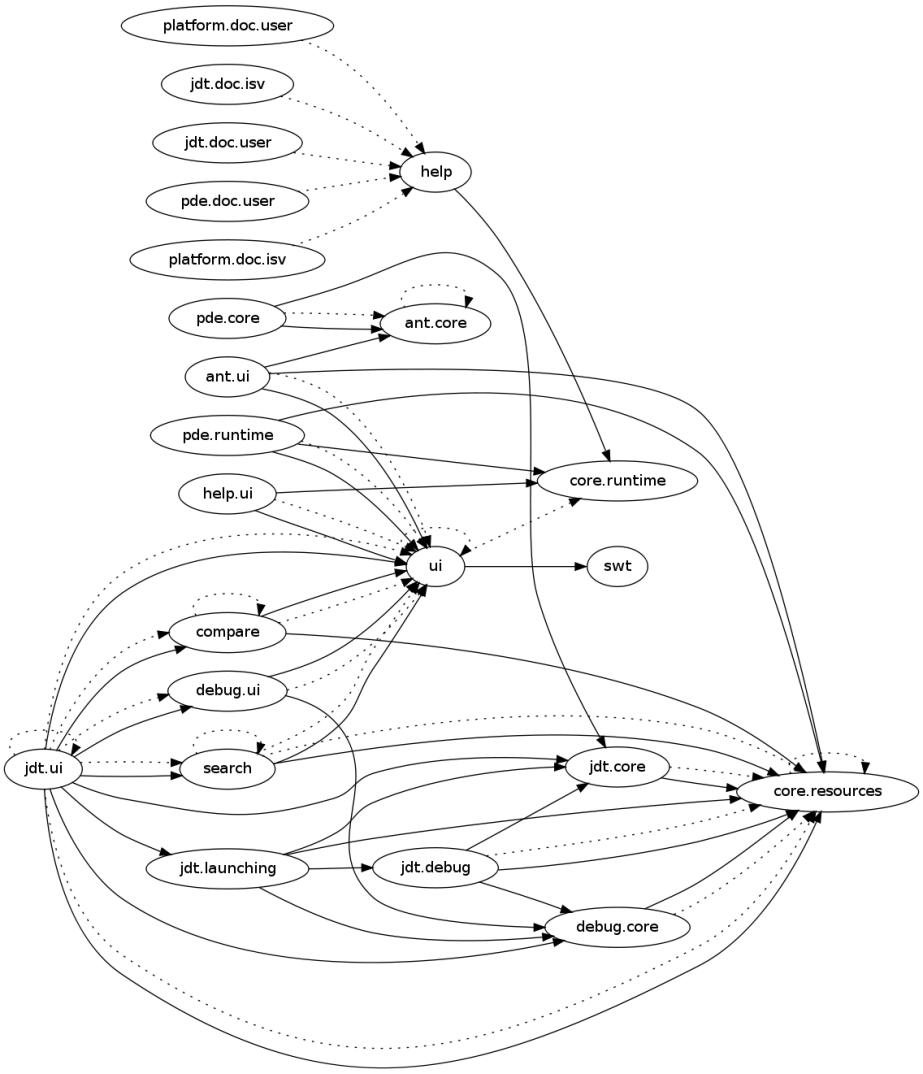

Fig. 3 The architectural core, with layers shown as columns

Notice that there are no static dependencies between documentation plugins because they do not contain any source code, but they use the extension point mechanism to document Eclipse in an incremental way.

4.2 Lehman's 6th Law

Figure 4 shows the evolution of Eclipse's size, along the two build sequences. Note that the number of kept plugins is relative to the first release in the sequence, i.e. 1.0 or 3.1. We consider all additions and deletions of plugins as unforced, because they are architectural choices.

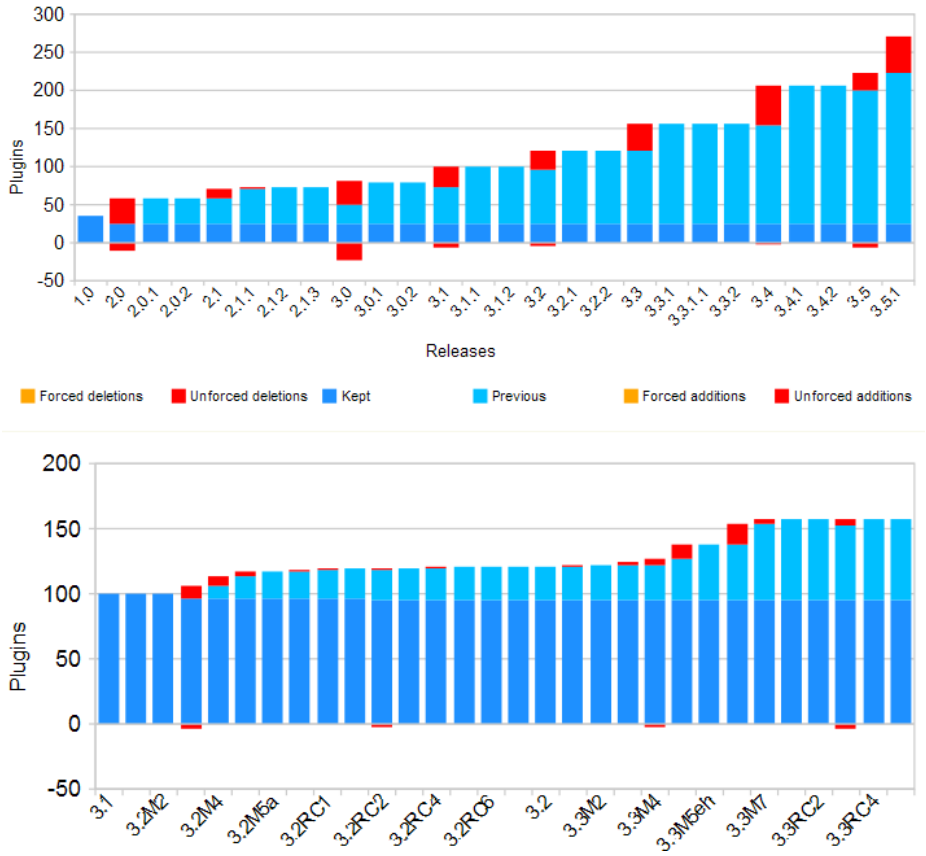

Fig. 4 Evolution of the size

We can observe that, over all releases, the size of the architecture increases more than sevenfold, from 35 to 271 plugins. The evolution follows a segmented growth pattern, in which different segments have different growth rates. In particular, the rate is positive during major and minor releases and mostly zero during service releases. Exceptions are service releases 2.1.1 (one plugin added) and 3.5.1 (48 new plugins). A closer look at 3.5.1 reveals that most of the added plugins are non-functional, providing support for the run-time infrastructure (the equinox plugins that implement OSGi) and for Eclipse's development (the release engineering tool and the Java Management Extensions plugins).

We further notice that no service release ever deleted a plugin, and that most architectural 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 by Fernández-Ramil et al (2008). Some studies observed superlinear growth, i.e. growth with increasing rates, in the number of source code files, and the same happens here at architectural level. Plotting the total number of plugins against the major and minor releases and service release 3.5.1, we found that the quadratic function $p = 1.9886r^2 + 3.5553r + 36.283$, where p is the number of plugins and r the release order number from 1 to 10, provided a better fit, with $R^2 = .9917$, than a linear or exponential regression model.

4.3 Lehman's 2nd Law

Figure 5 plots the overall complexity, i.e. relation *ID* (Section 3.2), over the main build sequence. The web companion 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 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).

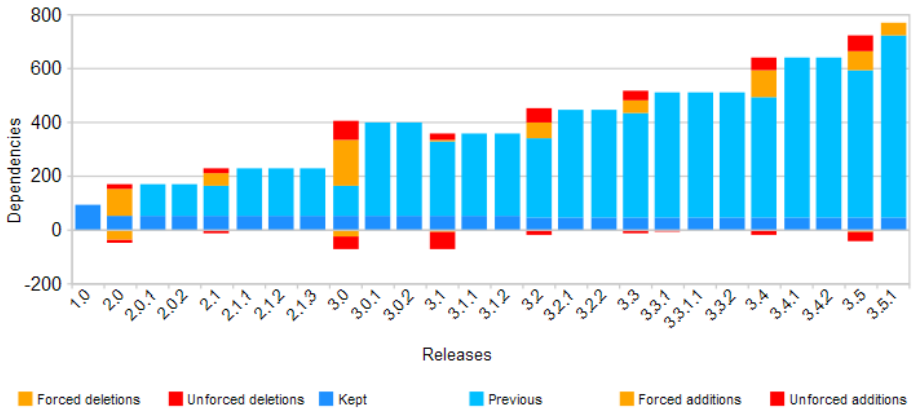

Fig. 5 Evolution of the overall complexity

We found again that dependencies change mostly during milestones. Compared to plugins, there are more changes during service releases, but still very few (e.g., 3 static dependencies removed in 3.3.1). All but one of the new dependencies in 3.5.1 are due to its new plugins.

Contrary to the continuous net increase of size, there has been a decrease of complexity by 11% 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.

It's unclear whether the evolution of overall dependencies follows a linear or superlinear trend: a straight line fits the data points with $R^2 = 0.9766$, a second-order polynomial with $R^2 = 0.9768$.

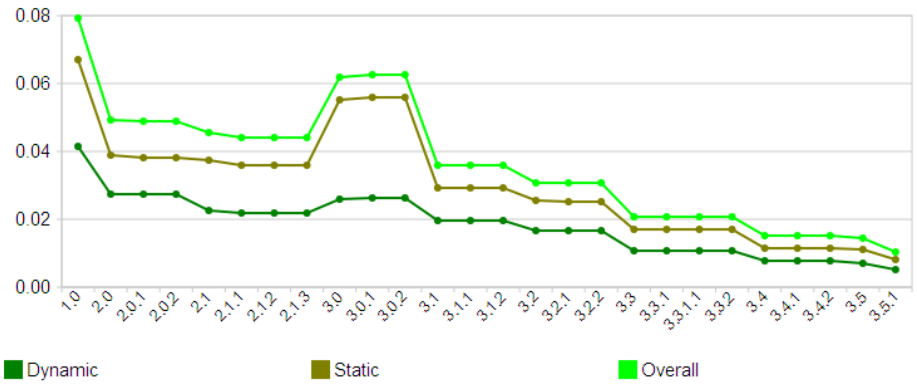

Fig. 6 Evolution of the cohesion

We point out that the constant height segments of kept elements in Figures 4 and 5 immediately indicate the existence of a stable architectural core.

4.4 Cohesion

The previous charts show that the number of plugins and dependencies grow 'in sync', following the same pattern. There are however two exceptions: release 3.0 substantially increased the number of dependencies while only slightly increasing the number of plugins, and release 3.1 reduced the dependencies while increasing the number of plugins. Moreover, although there are many more dependencies than plugins, the latter increase faster.

As stated in Section 3, the cohesion of a module is the ratio between the actual and the potential number of dependencies, which is the square of the number of plugins. Thus, the above observations entail that the cohesion is constantly decreasing, only increasing between 3.0 and 3.1, as one can see in Figure 6.

4.5 Coupling

The evolution of the coupling between the internal and external plugin modules also follows a segmented growth pattern, but with a substantial decrease in release 3.0, which replaced all external dependencies (Figure 7). 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 the internal 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 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 7 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.

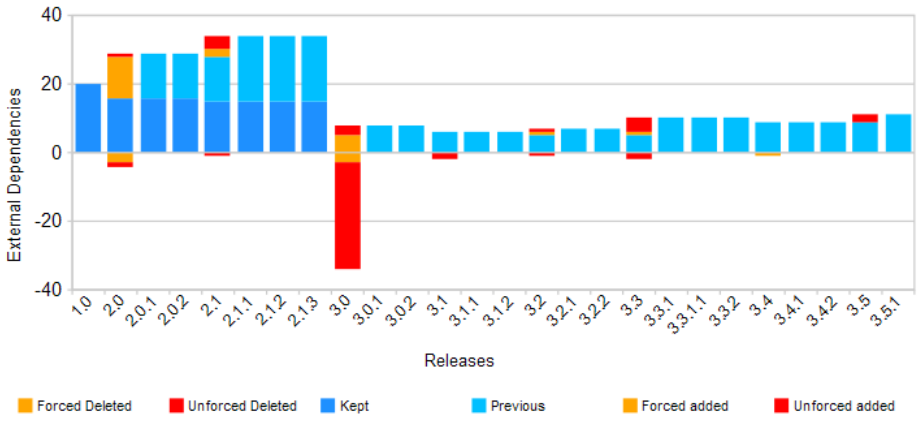

Fig. 7 Evolution of the coupling

4.6 Acyclic Dependency Principle

To check the ADP, we considered the union of static and dynamic dependencies, i.e. relation *ID* (Section 3.2), in order to be able to detect as many cycles as possible. We consider 'self-cycles' to be harmless, as they indicate plugins using their own extension points. In the case of longer cycles, we further check if they involve plugins from different features, because that indicates potential refactoring opportunities.

As the chart in the web companion shows, our scripts report a growth of self-cycles, from 8 in release 1.0 to 41 in 3.5.1, that follows the same segmented pattern as plugins and dependencies, with alternating big and small increments. Moreover, between releases 3.1 and 3.3 changes only occurred during milestones, which is coherent with our previous observations. Interestingly, there was a small reduction of self-cycles in 3.5, but more were added in 3.5.1. Except in 2.0, all self-cycle removals were unforced.

More interestingly, the only cycle we found with length over 1 involved just three plugins: ui, ui.editors, and ui.workbench.texteditor, all in feature platform. It appeared in release 2.1 and disappeared in release 3.0. None of the plugins were deleted, i.e. the cycle removal was unforced, but one plugin was moved to a different feature.

4.7 Open/Closed Principle

The previous charts show that there are far more additions than deletions. Also, unforced dependency changes, which require internal modification of existing plugins, do not outweigh forced dependency changes, which occur with the introduction or removal of plugins.

To get a richer assessment of the OCP, we also measured the absolute and relative number of extension points provided by the Eclipse SDK and how the numbers changed over time. The addition (deletion) of an extension point is considered forced if the plugin that provides the extension point has been created (resp. removed) simultaneously.

Although we see again the familiar growth pattern, we note that most extension points are added to already existing plugins (unforced additions in Figure 8). Hence, new plugins have few or no extension points, leading to a decrease in the average number of extension

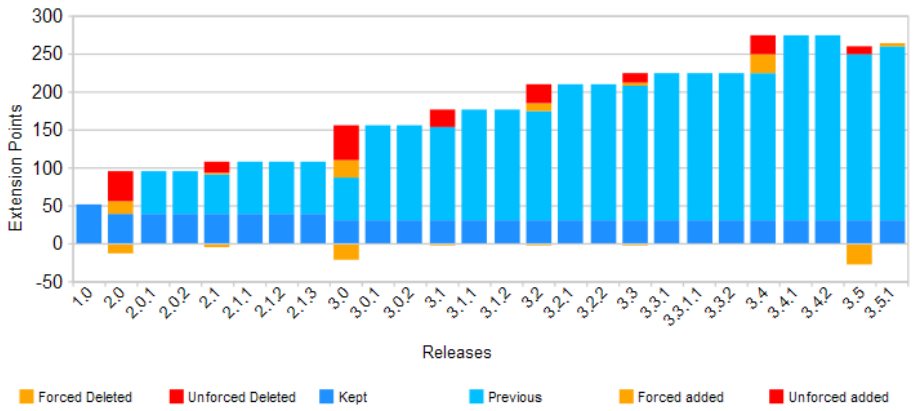

Fig. 8 Evolution of the absolute number of extension points

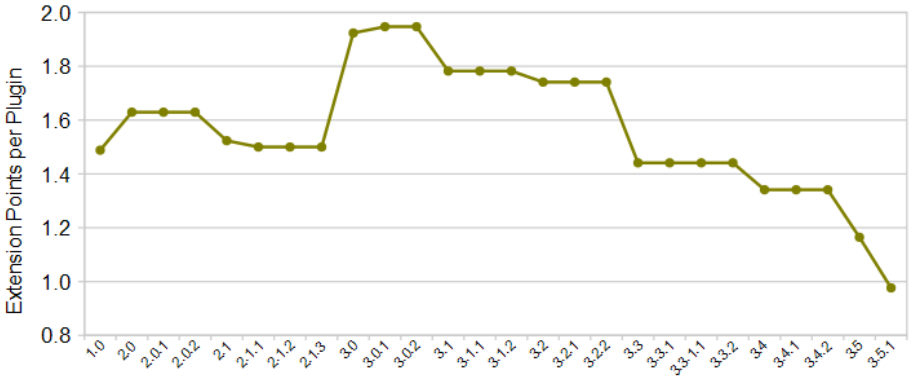

Fig. 9 Evolution of the plugin to extension point ratio

points per plugin (Figure 9). We can also see in Figure 8 that all deletions are forced, i.e. an extension point is never removed unless its host plugin is.

Also interesting is to see how many unused extension points there are, i.e. whether the SDK is “eating its own food” or providing extension points mostly for other Eclipse projects and 3rd-party plugins. The addition of an unused extension is considered to be forced if it is due to the deletion of a dynamic dependency, and hence the extension point became unused. The deletion of an unused extension point is forced if due to the deletion of the corresponding plugin.

We note that most additions are unforced (Figure 10) and that overall the absolute and relative number (Figure 11) of unused extension points grew, although it’s early to say if the decreasing trend started in release 3.5 (due to the removal of plugins) will continue. Until then, very few unused extension points had been deleted. The percentage of unused extension points has doubled from 9.4% in release 2.0, which deleted most of the extension points of 1.0, to 19.3% in 3.5.1 (Figure 11).

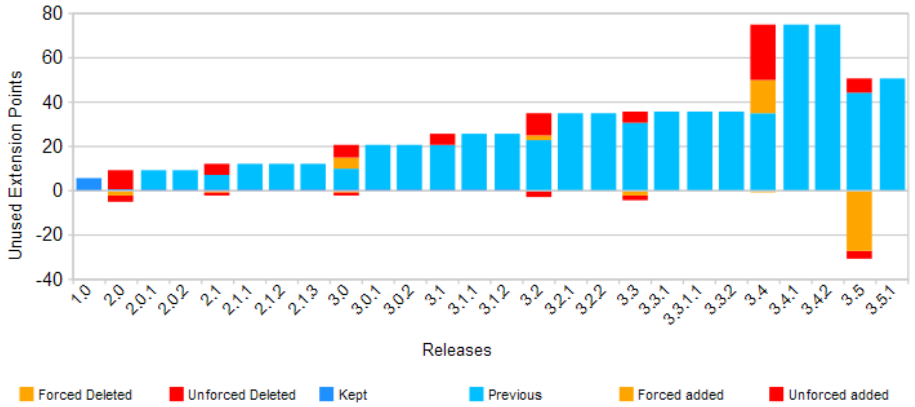

Fig. 10 Evolution of the absolute number of unused extension points

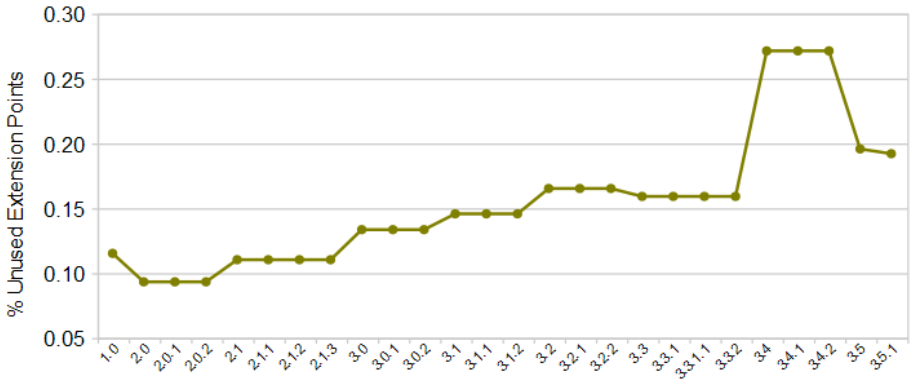

Fig. 11 Evolution of the percentage of unused extension points

4.8 Stability Dependency Principle

We checked the SDP by computing the instability of each internal plugin separately for static, dynamic and overall dependencies for all releases. We ignored self-cycles because Martin (1997) says that instability has to decrease in the direction of the dependency. Hence, a dependency between the same component (with a single instability value) necessarily violates the principle. The average plugin instability per release fall in the ranges [0.197, 0.286], [0.290, 0.371] and [0.249, 0.351] for static, dynamic and overall dependencies, respectively. Recalling the definition of instability from Section 2, this means plugins tend to be independent and responsible.

Figure 12 shows the very small number of dependencies that violate the SDP. Dynamic dependencies introduce fewer violations than static ones. In addition, the former are decreasing while the latter are increasing. Although the absolute number grows, the relative number of dependencies violating the SDP has been kept below 4% for most builds, as an additional chart in the web companion shows. Figure 12 shows that the overall and static dependencies violating the SDP are very similar (or even equal from 3.1 to 3.3.2), i.e. the dynamic dependencies contribute very few or even no extra violations.

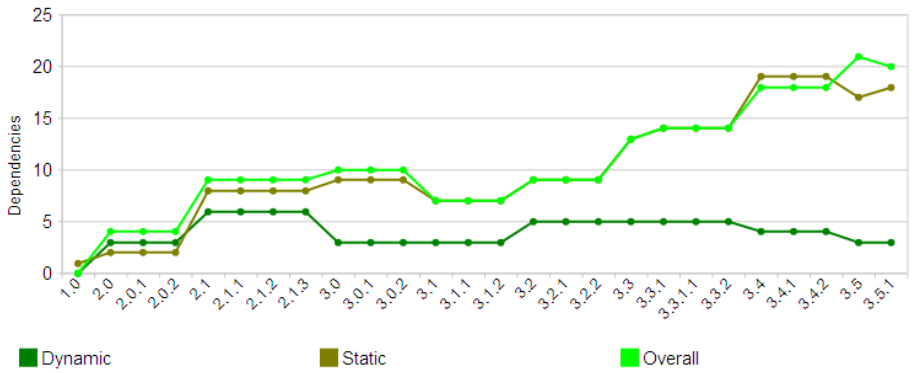

Fig. 12 Evolution of the number of dependencies violating the SDP

4.9 Common Reuse Principle

If plugin *A* depends on plugin *B*, then *A* is reusing *B*'s functionality. We use CCVisu (Beyer 2008) to visualise clusters of plugins according to their dependencies: groups of plugins with many dependencies among them will be put close together in the resulting layout, mutually independent plugins will be positioned far apart. Clusters show plugins that collaborate together and hence should be reused together (Section 2).

Given that the CRP builds on the assumption of the Reuse/Release Equivalence Principle, we wish to assess whether the clusters cross-cut a release unit. For that purpose, we aggregate plugins into two different kinds of release units, one from a user's perspective (features) and another from a developer's perspective (subsystems). If plugins in the same reuse cluster belong to different features or subsystems, then the release unit is not a reuse unit, a violation of the REP and CRP. In practical terms, it means that users have less choice of freely picking and mixing features, because inter-feature dependencies require additional features to be installed, and that software developers have to coordinate dependent plugins across subsystem boundaries.

Figure 13 was generated by CCVisu from the overall internal dependencies in release 3.5.1. Each circle represents a plugin, the radius being proportional to the number of that plugin's incoming and outgoing dependencies. A dependency between two plugins will bring them closer together in the layout. In the top part of the figure, plugins are coloured by feature, while in the lower part they are coloured by subsystem. Since features and subsystems are hierarchical, we only consider top level features and subsystems (like org.eclipse.sdk), which have the coarsest granularity and hence are more likely to encapsulate reused plugins. As we can see, even in the few cases of clear, albeit small, clusters, they bring together plugins from different features of subsystems. In the web companion we present the complex dependency graph of 3.5.1 at feature level; it confirms that many plugin dependencies cross-cut features, without any apparent cluster of more strongly connected features.

Instead of presenting one clustering graph per build, we used the x/y-coordinates computed by CCVisu for displaying the clustering, to measure the average intra-feature (or intra-subsystem) distance as the average distance between the centres of all pairs of circles with the same colour, and the average inter-feature (or inter-subsystem) distance by taking circles of different colours. Since there is only one clustering graph per release, but with two different colourings, all average distances are measured on the same graph and hence can

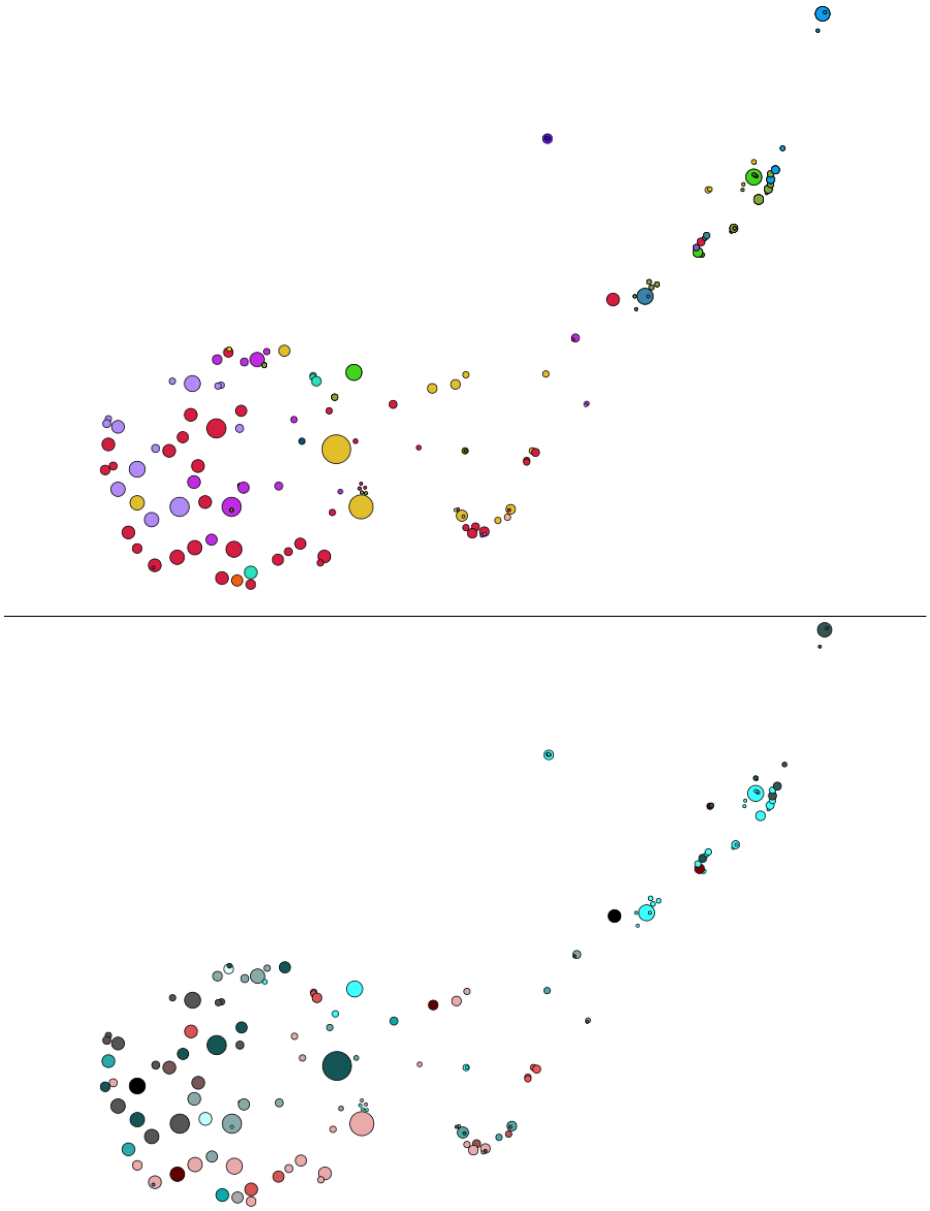

Fig. 13 Assessing the CRP in release 3.5.1 for features (top) and subsystems (bottom)

be compared. Figure 14 shows that the average intra-unit distance is always lower than the inter-unit one, but the former is approaching the latter. The average distance within subsystems is always smaller than within features, i.e. the subsystem decomposition of the SDK groups dependencies better than the feature decomposition.

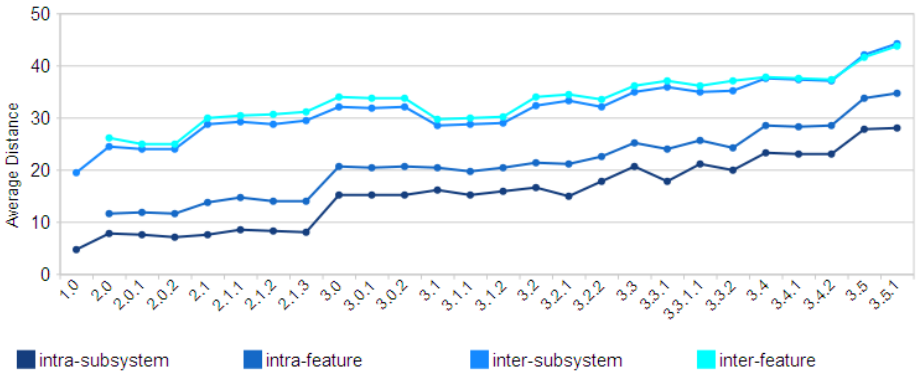

Fig. 14 Evolution of the average intra- and inter-unit distance in CCVisu's graphs

4.10 Common Closure Principle

We assess the CCP in a similar way, colouring plugins according to the feature or subsystem they belong to, but this time using CCVisu for its original purpose: to visualise co-change (hence its name). Clustered plugins will have changed together often; if they are of the same colour, the CCP is followed.

To obtain co-change clusters, we used Crocopat to compute from existing relations a new binary relation $UC(r, p)$, stating that plugin p was subject to an unforced change in major, minor or service release r. As change we consider the creation and removal of p and any change to p's provided extension points or p's fanout, because all those changes point to internal modifications of p. CCVisu's clustering algorithm will bring together all plugins that are connected to multiple releases, i.e. plugins that are more often changed together. CCVisu only shows the plugins (Figure 15), not the intermediate nodes corresponding to releases. The average distances in the figure are as follows: intra-feature 78.2521, inter-feature 155.161, intra-subsystem 103.837 and inter-subsystem 144.412. Like for the dependency clustering, average intra-unit distances are better than inter-unit, but this time the feature decomposition is better: co-changes cross-cut subsystems more than they cross-cut features.

The web companion includes interactive versions of the CCVisu figures, allowing one to see the name of each plugin by moving over or clicking on each circle.

5 Discussion

First, taking just the quantitative results observed from the historic structural measurements, we attempt to answer the assessment questions (Section 2.1) as directly as possible. Based on those answers and further information from developers and other studies, we then address the research questions, which includes the qualitative assessment of the architectural evolution of the Eclipse SDK. We conclude with potential threats to the validity of our approach and of our answers to all questions, and with lessons learned from performing this research.

5.1 Assessment questions

The architecture

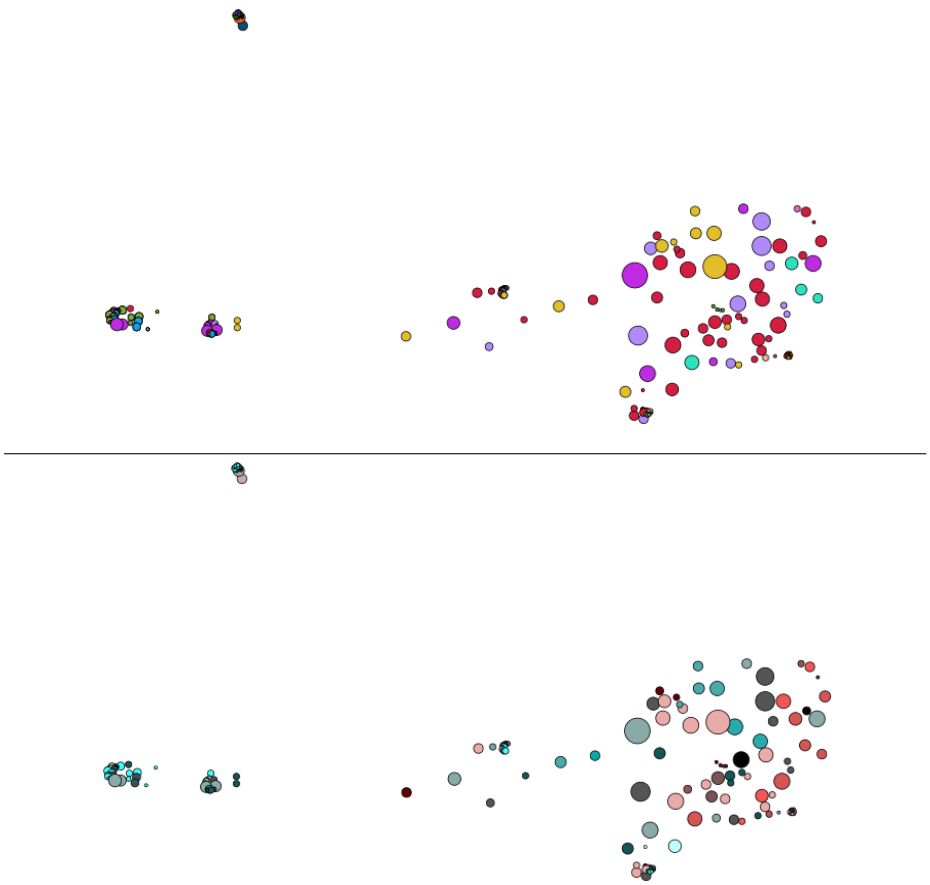

Fig. 15 Assessing the CCP, over all releases, for features (top) and subsystems (bottom)

1. Eclipse follows a layered architectural style throughout its evolution, the root layer being always the largest layer and where most plugins are added. Lower layers are relatively stable. The number of layers almost tripled over the period studied.
2. About half of the initial architecture has formed for the past 8 years an architectural core that contains fundamental documentation and functional plugins, forming almost one tenth of the latest architecture.

Lehman's laws

3. The architecture's size is always growing and as such follows Lehman's 6^{th} law of evolution. Growth follows a segmented pattern and is superlinear if we ignore the periods of no growth. Both characteristics have been observed for the source code evolution of other systems.
4. The complexity, as measured by the number of dependencies among plugins, also steadily increases, following the same segmented growth pattern observed for the plugins. However, whether the rate is linear or superlinear is unclear.
5. There has been some effort to reduce the system's growth, but overall deletions are far fewer than additions. The major reduction efforts have been in releases 3.0 (small

size increase, decrease of coupling), 3.1 (decrease of complexity and coupling) and 3.5 (decrease of extension points). Of these, 3.0 can be considered a major architectural restructuring due to the many unforced deletions and additions of plugins and dependencies.

6. The segmented evolution patterns are due to a systematic development 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 introduce (almost) no architectural changes, with the exception of the latest service release. All this means that the evolution of Eclipse's architecture follows a *punctuated equilibrium* pattern observed by Wu et al (2004) for other systems: long equilibrium periods alternate with relatively short punctuation periods (mainly a few milestones).

Structured Design

7. The architecture's cohesion has been steadily decreasing, except for the increase in 3.0, when the number of dependencies grew much more than the number of plugins. Overall, and independently of the type of dependencies considered, cohesion in 3.5.1 is about 1/8th of the value for 1.0.
8. Coupling, i.e. the number of external dependencies, is very small (Figure 7), especially compared to the number of internal ones (Figure 5), and has been kept largely constant after a substantial and explicit (i.e. unforced) reduction in release 3.0.

Martin's principles

9. The Eclipse architecture follows the Acyclic Dependency Principle: the only non-trivial cycle was removed during the major restructuring that led to 3.0.
10. The Open/Closed Principle is largely followed, i.e. Eclipse's architecture evolves more by extension than modification: the architecture has been changed more by addition of plugins rather than through their internal modification (made visible through unforced changes to dependencies). Additionally, many of the newly added extension points are unused (as one can see by comparing the additions in Figure 8 and Figure 10), which means that the absolute and relative number of unused extension points has substantially increased since 1.0. This in turn means that the SDK is opening up its plugins to provide services to external plugins. However, there is also some indication that adherence to the OCP might be diminishing: the average number of extension points per plugin has halved since 3.0 (Figure 9) and unused plugins have been diminishing since 3.5. It should be noted that unforced deletions of unused extension points are not problematic for the OCP: the extension point has become used by the SDK, but remains available to third-party plugins.
11. The architecture largely follows the Stable Dependencies Principle: since release 3.0, less than 3% of overall dependencies violate it.
12. Overall, the CRP is not violated, because the average intra-unit distance is lower than the inter-unit distance, i.e. plugins within the same top-level feature or subsystem are on average clustered (according to use dependencies) closer together than those in different features or subsystems. The distances are shorter for subsystems than features, i.e. the CRP applies better to the former. However, the difference between intra- and inter-unit distances has been decreasing, showing a deterioration of alignment with the CRP (Figure 14). A visual inspection shows that there are no clear clusters of reused plugins

belonging to a single feature or subsystem (Figure 13). One can hence argue that the CRP, while not violated, is far from being followed in an optimal way.

13. Like the CRP, the CCP is not violated, but neither does it seem to be strongly followed. The reason is similar: on average, plugins within the same feature or subsystem are changed more often together than those in different features or subsystems, but there are no clear clusters in the visualization with only circles of the same colour (Figure 15). Contrary to the CRP, features follow the CCP better than subsystems. This means that closely dependent subsystems are not necessarily changed together. This can be confirmed by comparing Figures 13 and 15, noting that both use the same colour for the same feature (or subsystem).

5.2 Research Questions

To answer the research questions, we have validated our results against another study and with developers. We e-mailed a very brief summary of our work to some SDK developers, asking them to confirm, reject, or explain specific findings. We formulated the questions so that they were focussed and could be answered with a simple yes/no or a brief sentence (see appendix B), in order to increase the chances of getting an answer from those busy developers. The e-mail was sent to developers of the IBM Rational Lab in Zurich whom the first author had briefly met in February 2009. The Rational Lab team has made substantial contributions to the SDK, in particular to the Java Development Tools, and is led by Erich Gamma, a key figure in Eclipse since its origins at OTI.

5.2.1 Fitness for purpose of the approach

As explained in Section 2.2, the first research question asks whether our approach is fit for the purpose of assessing architectural evolution. To answer the question, we must draw on an analysis of the threats to validity, on related work, and on developer feedback in order to consider whether our structural model and measurements are appropriate, the adopted design guidelines and principles are meaningful at architectural level, and the approach is relevant in practice.

While Lehman's 6^{th} law is certainly followed, the actual growth trend seems to be not very relevant, because the IBM developers told us that the superlinear growth of the architecture has not caused any significant problems, e.g., of comprehension (Q2 in appendix B). We conjecture this may be due to the just linear increase of the number of Java files, classes and lines of code in the SDK (Mens et al 2008). The superlinear growth of the number of plugins together with the linear growth of the code means that the average size of each plugin is decreasing. We computed a boxplot of the number of classes per plugin (Figure 16) and found that the median decreased from 117 in 2.0 to 55 in 3.5, while a few large plugins became larger, i.e. have more classes.

The superlinear growth in the number of plugins, at a rate that matches or surpasses the growth rate of number of dependencies, leads to the observed decrease in cohesion (Figure 6), contrary to expectations. The Eclipse SDK therefore falsifies the increased cohesion guideline, showing that it is possible to sustain evolution and a thriving eco-system of applications for years, in spite of a decreasingly cohesive architecture. We therefore conjecture this structured design guideline to be less relevant at architectural level, at least for systems that offer a flexible set of components on which to build on, like the Eclipse SDK. Of course, only further investigation with other case studies can confirm or refute our conjecture.

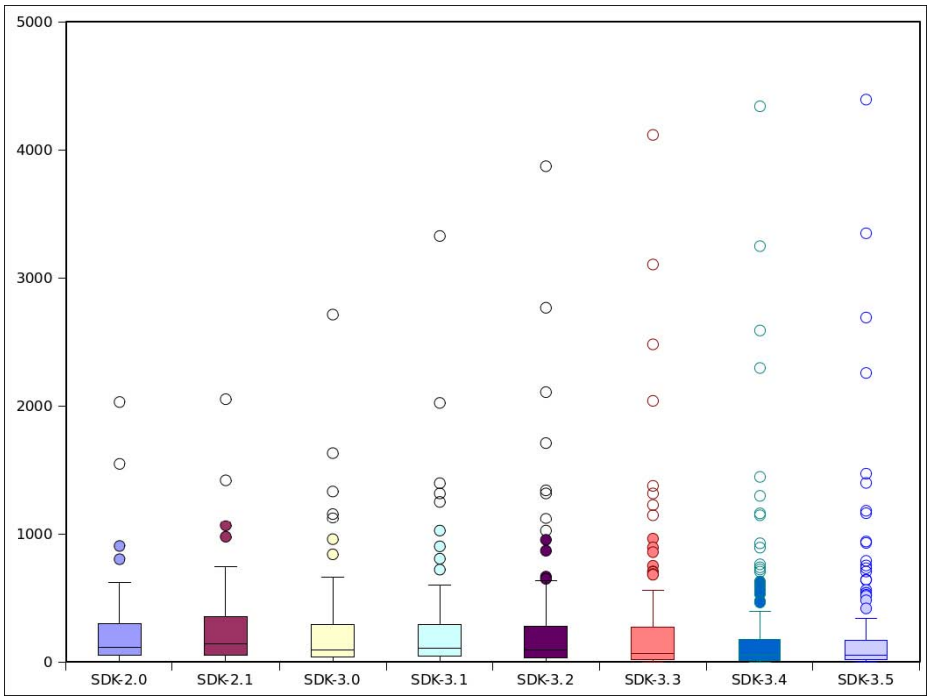

Fig. 16 Evolution of the number of classes per plugin

Our analysis of the OCP by comparing architectural additions to modifications may not reflect development reality: Mens et al (2008) found that more files are modified than added, while we found that more plugins are added than modified. The results are not contradictory because we consider only a subset of modifications, those that change dependencies between plugins. However, this points to a potential threat to validity of our first OCP metric, as it is incapable of capturing all internal modifications. On the other hand, our analysis of the OCP as the architecture's *potential* extension, by measuring extension points (Section 4.7), is likely to be more relevant for infrastructure architectures like the SDK's: the developers confirmed that they are 'opening up' existing plugins (Q6). However, whether there is *actual* extension of the SDK through those extension points requires an analysis of many other Eclipse and third-party plugins. Overall, we feel that our current approach is not fit for purpose to assess the OCP at architectural level of a single system. Some variation of our second metric might be needed to assess whether eco-systems of plugin-based architectures are adhering to the OCP by extending existing plugins rather than adding new ones, but that requires further investigation.

The IBM developers told us:

- that they adhere explicitly to the architectural change process we found (Q3);
- that the small amount of deletions (and the absence of unforced deletions of extension points) is to preserve backwards compatibility (Q1 and Q6);
- that cyclic dependencies are removed to keep plugins independently usable (Q4);
- that 3.0 was indeed a major architectural change due to the introduction of the Rich Client Platform and they suggest that the complexity decrease in 3.1 might have been a clean-up (Q5);

– that they are aware of the architectural core and that it was designed as such (Q9), and
– that extension points tend to be created in existing plugins because new plugins tend to be in the root layer and hence are less good hosts for extension points than the more 'central' plugins, i.e. in deeper layers (Q6).

The developers also confirmed that having finer-grained plugins is an explicit architectural aim (Q2), which means that while the actual growth rate may be of little relevance for architectural evolution, the corresponding assessment question (number 3 in Section 2.1) can help uncover architectural intentions. Our finding about the CCP was partly confirmed, in that developers told us that team structure loosely maps to features (Q7). Last but not least, we were explicitly told that not breaking APIs is a major goal, with extension points being part of a plugin's API. Hence, several of our observations, like few (mostly forced) deletions, are actually manifestations of this more fundamental aim. As a further example, the architectural core is stable because it provides many APIs. All this confirmed that many of our observations stem, directly or indirectly, from intentional choices, thereby supporting the relevance of our approach for architectural evolution practice.

To sum up, except for cohesion and the OCP, our assessment approach is largely fit for purpose of assessing architectural evolution. We would furthermore suggest that the actual growth rate of an architecture is not critical for its sustainable evolution, but we nevertheless keep assessment question 3 because it asks about growth patterns in general.

5.2.2 Assessment of the case study

Applying the assessment approach to the Eclipse SDK has provided quantifiable evidence, mostly confirmed by developers, of the following good architectural evolution practices:

– an upfront good design that provides a stable architectural core;
– the use of a single, familiar architectural style throughout history;
– the top layer, with highest instability, is indeed where most additions occur;
– a systematic architectural change process of punctuated equilibrium;
– few deletions overall, in particular unforced ones, to help keep backward compatibility;
– decreased coupling to external components;
– acyclic dependencies, and in the direction of stability;
– unforced addition of extension points, i.e., existing plugins are opened for extension.

From developer feedback we further learned that the team structure loosely follows the modularization that best matches the CCP, i.e., features. This may facilitate coordination and labour division of changes, and hence be one more lesson for architectural evolution.

We have however also observed a trend that may cause concern. Throughout history, the average intra-unit distance is approximating the inter-unit distance, i.e. dependencies tend to be less well contained within subsystem and feature boundaries. This is a sign of architectural decay: the higher-level modules are becoming less effective as release units to users and structuring devices for code.

To sum up, while the Eclipse SDK adheres to several principles and guidelines through a set of clearly observable practices, there is room for improving the architectural alignment of plugins with subsystems and features. Overall, we can present the Eclipse SDK as a pedagogical example of good architectural evolution practice.

5.3 Threats to validity

Like any other empirical study, the validity of ours is subject to several threats. In the following, we discuss threats to construct validity (relationship between theory and observation), internal validity (whether confounding factors can influence your findings), and external validity (whether results can be generalized).

5.3.1 Construct validity

A potential threat when proposing any metrics is that they don't adequately measure the intended concepts. In our case, the construction of the structural model and its metrics (Section 3.2) might not capture the concepts of size, coupling, cohesion and complexity at the architectural level. However, the adoption of a general and unifying framework (Briand et al 1996) that covers various existing metrics, as the authors show in their paper, and that distils fundamental intuitions (e.g., that the more relations a module has with others, the more coupled it is, or that the more relations a system has between its elements, the more complex it is) into a set of axioms, means that any metrics adhering to those axioms will also capture those intuitions, no matter at what level the metrics are defined. The minimalist and abstract graph-based nature of the framework (requiring just the definition of elements, their relations and the modules they belong to) makes it possible to define elements and modules at any granularity level (e.g. an element may be a single datum or instruction or a complete architectural component), and thus to have meaningful metrics at different levels, as Briand et al (1996) exemplify in their paper. Moreover, because the axioms are expressed in terms of different entities (e.g. cohesion is based on intra-module relations while size is based on elements), by basing our metric definitions directly on those entities, we are assured that our metrics are conceptually coherent with each other (e.g our cohesion metric is not measuring size or vice-versa)[3]. To sum up, the adoption of the measurement framework assures us that our metrics are appropriate to capture size, cohesion, coupling and complexity at the architectural level.

Even though the metrics are meaningful, they and the structural model may be too simple to provide adequate answers to the architectural evolution assessment questions. However, as shown in Section 4, even though our structural model has only a handful of elements (plugins, extension points, dependencies, modules, and a build sequence), over which simple counts are plotted, its flexibility (e.g., allowing different dependencies, modules and sequences) and generality allow us to go a long way and obtain a rich picture of unforced and forced changes, growth rates, change patterns, historic trends, and clustering analysis within and across modules. Moreover, the approach allows the answers to the assessment questions to reinforce (e.g. in the case of the segmented change pattern) or to complement (e.g. in the case of the CRP and CCP) each other.

While the use of a single axiomatic framework means that our basic metrics are well defined and consistent with each other, the use of multiple metrics to analyse the OCP, CRP and CCP is a threat to construct and internal validity for those principles, hence leading to less clear answers for them in Section 5.1. On the one hand, the two ways we measure adherence

[3] We should point out that the axioms are only meant to distinguish the concept(s) being measured by a metric; the axioms don't enforce independence of the metrics. For example, coupling and cohesion aren't completely independent of complexity because the latter is based on all relations, whereas the former two are based on the subsets of inter-module and intra-module relations, respectively. This means that the various monotonicity axioms together impose that adding intra-module relations will decrease neither cohesion nor complexity, and likewise adding inter-module relations doesn't decrease coupling and complexity.

to the OCP confuses *potential extensions for third parties*, via the (unused) extension points, and *actual extensions of the system itself*, via a comparison of additions vs. modifications. On the other hand, our approach to the CCP and CRP may be rather sensitive to the chosen modularization: the choice of top-level features and subsystems may have been too coarse-grained and therefore may include confounding factors. However, using multiple metrics for the same principle provides a richer and more realistic picture of the necessarily complex and multi-faceted nature of software systems. For example, our proposal to measure two average distances per modularization and per build goes towards providing an objective answer about adherence to the CCP and CRP, making it possible to clearly see trends that would be nigh impossible to spot by visual comparison of CCVisu's pictures over multiple builds.

A further identified construct validity threat is missing primary data. Our assumption that the main punctuation periods are milestones relies only on the milestones and release candidates for releases 3.2 and 3.3. If we had the interim builds for all the other minor releases, the result might be completely different. This was mitigated by the developers' feedback.

Another example of a threat to construct validity is that the dependencies defined in the metadata might be missing relevant dependencies between the plugins' classes. That would explain why we found no cycles, while Melton and Tempero (2007) report thousands of cycles among Eclipse's classes, some of them involving hundreds of classes. Another possibility is that the granularity of plugins is such that cycles remain within the same plugin. Only a replication of their study, but considering plugin boundaries, can tell.

5.3.2 Internal validity

Besides the threats to internal validity of the OCP, CCP and CRP analysis mentioned above, bugs in our tool infrastructure will lead to wrong measurements which in turn might invalidate some of our analysis. Given our multi-faceted measurements over the same Eclipse build data, we can cross-check the outputs for coherence. For example, we noticed our scripts were missing several features when we saw the layouts of the bottom and top of Figure 13 were different, and corrected the mistake.

Although we did not account for the renaming of plugins (Section 3.2), we do not think it is a serious threat to validity. Renamings are currently a subset of the intersection of deletions and additions and hence must be few in number, because there are not many deletions. Accounting for renamings would just slightly increase the number of previous plugins and slightly decrease the number of additions and deletions, without changing the substance of our findings.

Finally and more importantly, small variations in the data to be included or discarded can make a big difference in the results. For example, we initially considered all plugins and obtained a growth in size that was truly astounding. Looking at the data, we saw the many added org.eclipse.*.source plugins, which we subsequently ignored. Other examples are the exclusion of self-cycles for computing instability and checking the SDP, and the definition of unforced changes. All these decisions are part of the empirical study design and we have justified them throughout the paper.

A further threat is that the opinion of very few developers might not be representative of the wider Eclipse SDK team. However, the contacted developers are very experienced and contribute to important parts of the SDK, like the Java Development Kit. We therefore assume their feedback carries authority.

5.3.3 External validity

Our analysis of the relevance of questions might be biased by two characteristics of the Eclipse SDK: the relatively 'homogeneous' development team, mainly provided by IBM, and the nature of the system, an application framework. These two factors mean, for example, that the relevance of a stable architectural core might be smaller for a 'black-box' end user application, and that the rapid increase of an architecture and its loss of cohesion might be more relevant for open source projects with fluctuating development teams. However, this threat is somewhat mitigated by having chosen a paradigmatic case study that represents an important set of widely used professional open source software systems. Moreover, given that most of the principles we analysed are very general, they should be relevant for most large and complex systems that require sustainable and effective architectural evolution, although the relative importance of each assessment question may vary from system to system.

5.4 Lessons learned

Analysing a software system's evolution from its repository raises several general issues and challenges (Wermelinger and Yu 2011). In particular, we learned some valuable lessons in the process of doing the research reported in this paper.

Keeping a local copy of primary data, even if it is easily and freely accessible on the internet, is highly recommended. The Eclipse project no longer keeps older milestones and release candidates in their archive, in order to save storage and bandwidth. Therefore, if we hadn't kept a copy of those builds from the initial stages of our work, it would have been more challenging to analyse what happens between major and minor releases, i.e. where exactly architectural changes are introduced.

Developers of successful projects are extremely busy: providing a succinct list of focussed questions, preferably with multiple choice options, is a good way to elicit feedback. Although we sent some drafts of this work to the IBM developers for their feedback, we only got a reply when we e-mailed a short list of specific questions. Moreover, one should ideally avoid contacting developers during major release periods, but paper deadlines may not give much leeway.

Empirical research is a constant feedback loop between the data and the researchers. Results of one iteration inform the next iteration. For example, as we said in Section 5.3, the initial results of the size chart led to the filtering out of source plugins, because it would be effectively double counting of the same plugin, once as functionality provider and once as documentation provider. The use of particular tools may also lead to unexpected further investigations. For example, we adopted CCVisu to visualise the clustering of plugins according to dependencies, and it was the tool's use of physical 2D locations to layout the clusters that triggered our idea to check the CCP and CRP by capturing a whole picture as the average distances of the x/y-coordinates of plugins and then plotting them over time.

6 Related Work

This paper complements a large body of empirical work on design heuristics and principles, on evolution patterns, and on Eclipse. We necessarily refer to only a small subset of it.

6.1 Design

Although many design heuristics have been proposed (Parnas 1972; Liskov 1987; Johnson and Foote 1988; Lieberherr et al 1988; Meyer 1992; Lakos 1996; Riel 1996), few of them have been validated through empirical studies. One early proponent of the study of violations of design heuristics was Ciupke (1999). Since then, all kinds of approaches have been taken, from the very concrete ones that negate the statements of the design heuristic and convert them into logic clauses (Ciupke 1999), to the very abstract ones, like bad smells (Fowler et al 1999) and anti-patterns (Brown et al 1998), which are symptoms of potentially inappropriate design abstractions. Algorithmic mechanisms to automatically detect such symptoms use metrics (Marinescu 2001; Tahvildari and Kontogiannis 2003; Crespo et al 2005; Munro 2005; Walter and Pietrzak 2005), logic clauses (Tourwe and Mens 2003), historical information about source code changes (Xing and Stroulia 2004), or a combination of approaches (Moha et al 2006).

Other work attempts to evaluate the effect of following (or not) a design heuristic or pattern. For example, Juergens et al (2009) aim to correlate code cloning with fault proneness, and Ratiu et al (2004) find that bad smells like god- and data-classes are persistent but stable. More related to this paper are the studies of Basili et al (1996) and Briand et al (2000), showing that fault proneness might be correlated with lack of cohesion and high coupling. However, it seems that these results come from the confounding effect of the size of classes (Emam et al 2001). Nevertheless, this does not refute the findings with respect to other software quality attributes such as maintainability, understandability, or reliability. For instance, Dagpinar and Jahnke (2003) showed that cohesion and indirect/export coupling measures are not correlated with maintenance, but coupling defined as direct import relations (which correspond to the static dependencies in this paper) is inversely correlated with maintainability.

Simon (1962) argued that using hierarchical structures reduces system complexity, but Briand et al (2000) used software metrics to show that deep hierarchies are correlated with fault proneness, and probably indicate a high cognitive complexity of classes. That might explain why the depth of Eclipse's hierarchical structure has grown slowly after it doubled from 6 to 13 in 3.0 (Section 4.1).

As for the empirical analysis of Martin's principles, Melton and Tempero (2007) also looked at the ADP, by measuring across a wide range of Java applications the lengths of all cycles present in each application, finding many long cycles that may impede comprehension. One must however note that several kinds of dependencies (inheritance, calls, imports, etc.) were brought together in the same cycle, which may inflate results.

Hansen et al (2009) computed three architectural metrics and four product metrics on the Java source code of 1,141 SourceForge projects, and applied regression models to see if architecture has an effect on product quality. One of the architectural metrics is the average, over all packages of a project, of a metric proposed by Martin (1997) to check his *Stable Abstractions Principle* (SAP): "the abstraction of a package should be in proportion to its stability". Modifying abstract packages, i.e. those that contain interfaces implemented by other packages, can have ripple effects, hence they should be stable.

Hansen et al found that mid-range values of the SAP-adherence metric lead to better values of the product metrics, e.g. a lower ratio of open to total defects. However, the authors point out that while their findings have high statistical significance due to the number of projects analysed, the spread of values is large. The regression models are therefore not very adequate for assessing single projects.

Although we initially considered the SAP, we couldn't come up with any suitable architectural level metric in alternative to Martin's metric, which requires the number of abstract classes and interfaces, information that is usually not available at architectural level, and in particular can't be obtained from Eclipse's metadata. We hence discarded the SAP from our assessment because it doesn't satisfy our second and third requirements (Section 2.1): 'abstractness' is not a concept that is meaningful or measurable of architectural components in general.

6.2 Evolution

The pioneering work of Lehman and Belady (1985), a systematic and longitudinal study of software evolution, originally involved the IBM OS/360 operating system. The main findings showed that its growth in size (number of source files) was at first linear, but after a specific point it became unpredictable. In contrast with that result, thirty years later Godfrey and Tu (2000) highlighted that the growth of the Linux kernel followed a superlinear rate (e.g. a quadratic curve). Investigating the subsystems composing the Linux kernel, additional findings suggested that the drivers module was the fastest growing of all. Considering the architecture of Linux, this highlights the fact that a plugin-based component (e.g. the I/O component) can achieve alone a largely different evolution with respect to other components in the same system, fundamentally raising the question of whether or not Lehman's laws should be considered valid for the case of open source systems (OSS) systems.

That question was investigated by Fernández-Ramil et al (2008), who surveyed several studies on the evolution of OSS. Most of them 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. The authors concluded that OSS systems seem to have less regular, and hence less easy to predict, growth patterns than proprietary systems, and that evolutionary trends exhibit discontinuities. Our work provides one more data point about OSS evolution, this time at the architectural level, contrary to most existing studies. We were able to also detect a discontinuous (i.e. segmented), and yet quite regular, superlinear growth pattern.

Eldredge and Gould (1972) proposed a theory of evolutionary biology, called punctuated equilibria, stating that biological evolution occurs mostly in rapid bursts of speciation, contrary to the until then assumed slow and continuous development of new species. Wu et al (2004) adopted the same viewpoint, with architecture taking the analogue of biological morphology. They 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 sources of information. Instead of reverse engineering the architecture from files (or classes, for OO code), we use configuration metadata about architectural components. Instead of hoping that a chronological analysis of fixed time periods might reveal architectural changes, we start with the logical sequence of builds. Moreover, we sample different kinds of builds so that we can check

whether they correspond to particular periods: indeed, service releases and most release candidates represent equilibrium periods, while milestones are the punctuation periods.

In his PhD thesis, van Belle (2004) proposes two system evolution metrics, breadth and weight, to quantitatively analyse Martin's Common Reuse and Common Closure Principles. The breadth is the average number of units touched by each change made to the system, while the weight is the average size of all units touched by a change. Such metrics were used to compare the reuse units (packages) defined by developers against those automatically generated by restructuring the code in such a way that the breadth and the weight are reduced. The author concluded that although automatically generated modularity can outperform a human-generated one, suggesting more efficient ways of organizing the code entities with respect to past evolution, the semantic coherence required by humans is hard to capture by automatic clustering techniques.

Van Belle also proposed to measure the likelihood, impact and acuteness of change of individual elements (e.g. methods). A high acuteness means that the element changes infrequently (low likelihood), but when it does, it has a big impact (many co-changes). The author then relates his metrics with Martin's SDP 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 goes further than Martin's, because it analyses the closure of change propagation (by observing all co-changes) and not just an element's immediate structural neighbours.

Change likelihood, impact and acuteness are suited for a low-level analysis (i.e. at code level) of fine-grained evolution (e.g. after each commit transaction), because the approach depends on an assurance that the observations do indeed capture change propagation, i.e. intentional co-changes. In our case study, each SDK build includes multiple, unrelated architectural changes over various subsystems and features, and therefore the impact and acuteness measures would be inflated and coarse-grained. In fact, in previous work (Wermelinger et al 2008), we found some relation between stability and acuteness for the Eclipse SDK, but not strong enough for the former to predict the latter. Indeed, it's unclear how van Belle's metrics could be used for assessment, the aim of this paper. Observing that an element had high change acuteness in the past is by itself not an indicator of good or bad design. What is important is for that element to have dependencies in the direction of stability to facilitate any future changes, and that is what we assess by checking adherence to Martin's SDP principle.

6.3 Eclipse

Although there are several empirical papers using Eclipse as case study, most are about unrelated topics, like bug prediction.

Mens et al (2008) also analyse the evolution of the Eclipse SDK, but have different goals, concentrating just on the verification of Lehman's first, second and sixth laws: the common aims with this paper are hence only Questions 3 to 5 (Section 2). They also rely on different primary data, mainly the binary code for Windows, and only for major and minor releases up to 3.3, whereas we look only at metadata, but for more builds. On the other hand, their narrower scope and use of bytecode allowed a more in depth treatment of size and complexity, with various metrics being applied at different levels. For example, Mens et al also find that the number of plugins increases superlinearly, but that the number of classes, files and lines of code grows linearly, and that the size of each subsystem relative to the overall system size remains remarkably uniform. The authors state that if one assumes complexity

is a function of size, then complexity is increasing, which is compatible with our structural view of complexity. They add that if complexity is assumed to be an inverse function of productivity, with sublinear growth suggesting increased complexity, then there is no evidence of increasing complexity. Furthermore, they analyse the code of each subsystem with the commercial STAN[4] tool for structural complexity code analysis, and find that the number of warnings increases linearly for the `jdt` and `ui` subsystems, and remains constant for the other ones. Since the overall number of complexity warnings and the size of code (in files, classes, or lines) both increase linearly, this third view of code complexity suggests it remains constant per code unit. Together, the authors' findings at lower granularity levels and our assessment of Eclipse's architectural evolution, might explain why the increase of architectural complexity does not lead to unsustainable growth.

Hou (2007) 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 echoes our finding of a stable layered core, and reinforces the lesson that well known architectural styles and design patterns can provide a solid foundation for system evolution.

Like any complex software system, there are many different ways of analysing and measuring Eclipse and its changes. The above two studies and ours thus offer complementary perspectives, together providing richer insights into Eclipse and the general lessons that can be learned for software design and evolution.

7 Concluding remarks

The paper makes two significant contributions. The first is a replicable architectural evolution assessment framework, made of assessment questions, a data model, and a tool infrastructure, that share various qualities: they are generic, flexible, light-weight, and extensible. The assessment questions probe basic and fundamental principles that are independent of particular domains, paradigms and languages and may have a wide-ranging effect on system development. The questions can hence be posed in other case studies, complementing fine-grained perspectives on design. The relational data model can be applied at various levels of abstraction. It is based on a simple core of concepts and measurement axioms by Briand et al (1996), which we instantiated with particular metrics and extended to handle build sequences and service-providing components, leading to a rich picture of an evolving system: unforced changes, cycle analysis, clustering distances, etc. Finally the tool infrastructure generates the data and the corresponding visualizations that help spotting deviations from established trends and practices, and hence could be part of an architect's or project manager's dashboard.

Our assessment framework was applied to a paradigmatic and 'most likely' critical case study (Flyvbjerg 2006): the Eclipse SDK, a large, mature, complex, and successful application framework, on which many other Eclipse and third-party projects depend, and therefore requires good architectural evolution practice for its continuing success. This choice, together with developer feedback and information from another study on the SDK's code base evolution, allowed us to validate that our assessment approach was largely fit for purpose. We found that some design guidelines and principles are not quite measurable or meaningful at architectural level or not relevant for sustainably effective architectural evolution.

[4] http://www.stan4j.com

The second significant contribution is the rich narrative account of the architectural evolution of a single case study, which can therefore be offered as a pedagogical exemplar of bad and/or good practices. In particular, our assessment concludes that the Eclipse SDK follows several practices that support sustainable architectural evolution, and that have direct impact on its success as application framework, because several of the practices emerge from the more fundamental aim of managing APIs carefully, like avoiding to break existing APIs, and adding APIs to existing responsible components. However, we also observed a trend that may suggest some architectural decay, but developers weren't able to comment on it.

A single case study is of course not enough to conclusively prove the relevance and adequacy of our proposal for architectural evolution assessment, and we certainly do not claim the list of assessment questions is necessary and sufficient for all kinds of systems. We therefore hope that the light-weight, generic and tool-supported approach, together with the presented developer-supported validation of a non-trivial case study, will inspire further case studies. They would be a worthwhile source of concrete knowledge for anyone interested in evolving a system the success of which crucially depended on its architecture.

Acknowledgements

We thank Markus Keller, Daniel Megert, and Martin Aeschlimann from the IBM Zurich Rational Lab for their feedback, partly given during the preparation of release 3.6, Dirk Beyer for helping us use CCVisu more effectively, and the anonymous reviewers for their detailed and insightful comments, which helped improve the paper.

The third author has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State, Belgian Science Policy.

References

Basili V, Briand L, Melo W (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761

van Belle T (2004) Modularity and the evolution of software evolvability. PhD thesis, University of New Mexico

Ben-Ari M (1982) Principles of Concurrent Programming. Prentice-Hall

Beyer D (2008) CCVisu: automatic visual software decomposition. In: Proc. Int'l Conf. on Software Engineering, companion volume, ACM, pp 967–968

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

Bloch J (2001) Effective Java. Addison-Wesley

Bois BD, Rompaey BV, Meijfroidt K, Suijs E (2008) Supporting reengineering scenarios with FETCH: an experience report. Electronic Communications of the EASST 8, selected papers from the 2007 ERCIM Symp. on Software Evolution

Briand LC, Morasca S, Basili VR (1996) Property-based software engineering measurement. IEEE Trans Software Eng 22(1):68–86

Briand LC, Morasca S, Basili VR (1997) Response to: Comments on "property-based software engineering measurement: Refining the additivity properties". IEEE Trans Software Eng 23(3):196–197

Briand LC, Wüst J, Daly JW, Porter DV (2000) Exploring the relationship between design measures and software quality in object-oriented systems. J Syst Softw 51(3):245–273

Brown W, Malveau R, Mowbray T (1998) AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley

Ciupke O (1999) Automatic detection of design problems in object-oriented reengineering. In: Proc. 30th Int'l Conf. on Technology of Object-Oriented Languages and Systems, IEEE, pp 18–32
Crespo Y, López C, Marticorena R, Manso E (2005) Language independent metrics support towards refactoring inference. In: Int'l Workshop on Quantitative Approaches in Object-Oriented Software Engineering
Dagpinar M, Jahnke JH (2003) Predicting maintainability with object-oriented metrics - an empirical comparison. In: Proc. Working Conf. on Reverse Engineering, IEEE, pp 155–164
Eldredge N, Gould SJ (1972) Punctuated equilibria: an alternative to phyletic gradualism. In: Schopf T (ed) Models in palaeobiology, Freeman and Cooper, San Francisco, pp 82–115
Emam KE, Benlarbi S, Goel N, Rai SN (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–650
Eysenck HJ (1976) Case studies in behaviour therapy, Routledge, chap Introduction
Fernández-Ramil J, Lozano A, Wermelinger M, Capiluppi A (2008) Empirical studies of open source evolution. In: Software Evolution, Springer Verlag, chap 11, pp 263–288
Flyvbjerg B (2006) Five misunderstandings about case-study research. Qualitative Inquiry 12(2):219–245
Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional
Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley
Godfrey MW, Tu Q (2000) Evolution in open source software: A case study. In: Int'l Conf. on Software Maintenance, IEEE, pp 131–142
Hansen KM, Jónasson K, Neukirchen H (2009) An empirical study of open source software architectures' effect on product quality. Tech. Rep. VHI-01-2009, Engineering Research Institute, Univ. of Iceland
Hou D (2007) Studying the evolution of the eclipse java editor. In: Proc. OOPSLA Workshop on Eclipse Technology eXchange, ACM, pp 65–69
Johnson RE, Foote B (1988) Designing reusable classes. Journal of Object-Oriented Programming 1(2):22–35
Juergens E, Deissenboeck F, Hummel B, Wagner S (2009) Do code clones matter? In: Proc. Int'l Conference on Software Engineering, IEEE, pp 485–495
Kuhn TS (1987) What are scientific revolutions? In: The probabilistic revolution, vol 1, MIT Press, pp 7–22
Lakos J (1996) Large-Scale C++ Software Design. Addison-Wesley Professional
Lehman MM, Belady LA (1985) Program evolution: processes of software change. Academic Press
Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution - the nineties view. In: Proc. Symp. on Software Metrics, IEEE, pp 20–32
Lieberherr KJ, Holland I, Riel A (1988) Object-oriented programming: An objective sense of style. In: Proc. Int'l Conf. on Object Oriented Programming, Systems, Languages, and Applications, pp 323–334
Liskov B (1987) Data abstraction and hierarchy. In: Proc. Int'l Conf. on Object Oriented Programming, Systems, Languages, and Applications, ACM, pp 17–34
Marinescu R (2001) Detecting design flaws via metrics in object oriented systems. In: Proc. Int'l Conf. on Technology of Object-Oriented Languages and Systems, IEEE, pp 173–182
Martin RC (1996) Granularity. C++ Report 8(10):57–62
Martin RC (1997) Large-scale stability. C++ Report 9(2):54–60
Medvidovic N, Dashofy EM, Taylor RN (2007) Moving architectural description from under the technology lamppost. Information and Software Technology 49(1):12–31
Melton H (2006) On the usage and usefulness of OO design principles. In: Companion to the 21st OOPSLA, ACM, pp 770–771
Melton H, Tempero E (2007) An empirical study of cycles among classes in Java. Empirical Software Engineering 12(4):389–415
Mens T, Fernández-Ramil J, Degrandsart S (2008) The evolution of Eclipse. In: Proc. 24th Int'l Conf. on Software Maintenance, IEEE, pp 386–395
Meyer B (1988) Object-Oriented Software Construction. Prentice Hall
Meyer B (1992) Applying 'design by contract'. Computer 25(10):40–51
Moha N, Guéhéneuc YG, Leduc P (2006) Automatic generation of detection algorithms for design defects. In: Proc. Int'l Conf. on Automated Software Engineering, IEEE, pp 297–300
Munro M (2005) Product metrics for automatic identification of "bad smell" design problems in Java source-code. In: Proc. Int'l Symp. on Software Metrics, IEEE, pp 15–24
Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058
Popper KR (1959) The Logic of Scientific Discovery. Hutchinson
Ratiu D, Ducasse S, Girba T, Marinescu R (2004) Using history information to improve design flaws detection. In: Proc. European Conf. on Software Maintenance and Reengineering, IEEE, pp 223–232

Riel A (1996) Object-Oriented Design Heuristics. Addison-Wesley Professional
Simon HA (1962) The architecture of complexity. Proceedings of the American Philosophical Society 106(6):467–482
Stevens W, Myers G, Constantine L (1979) Structured design. In: Classics in software engineering, Yourdon Press, pp 205–232
Tahvildari L, Kontogiannis K (2003) A metric-based approach to enhance design quality through meta-pattern transformations. In: Proc. European Conf. on Software Maintenance and Reengineering, IEEE, pp 183–192
Tourwe T, Mens T (2003) Identifying refactoring opportunities using logic meta programming. In: Proc. European Conf. on Software Maintenance and Reengineering, IEEE, pp 91–100
Walter B, Pietrzak B (2005) Multi-criteria detection of bad smells in code with UTA method. In: Extreme Programming and Agile Processes in Software Engineering, Springer, pp 154–161
Wermelinger M, Yu Y (2011) Some issues in the 'archaeology' of software evolution. In: Generative and Transformational Techniques in Software Engineering III, LNCS, vol 6491, Springer, pp 426–445
Wermelinger M, Yu Y, Lozano A (2008) Design principles in architectural evolution: a case study. In: Proc. 24th Int'l Conf. on Software Maintenance, IEEE, pp 396–405
Wong K (1998) The Rigi User's Manual, Version 5.4.4
Wu J, Spitzer C, Hassan A, Holt R (2004) Evolution spectrographs: visualizing punctuated change in software evolution. In: Proc. 7th Intl. Workshop on Principles of Software Evolution, pp 57–66
Xing Z, Stroulia E (2004) Understanding class evolution in object-oriented software. In: Proc. Int'l Workshop on Program Comprehension (IWPC), IEEE, pp 34–43

A Web companion

The 'web companion' to this paper is a compressed folder, available from http://oro.open.ac.uk/28753, containing all the figures of Section 4, and additional ones, in a larger and easier to read format than in this paper. Once the folder has been downloaded and uncompressed, open the included HTML file in a web browser. The file is fed by the Google spreadsheets mentioned in Section 3.3 and displays interactive visualizations in SVG format.

B Questionnaire

The following are the questions sent by e-mail to the developers mentioned in the acknowledgements.

Q1 We found that very few plugins, extension points and dependencies are deleted (compared to additions). Is this an explicit aim of yours? If yes, is it to keep backwards compatibility of existing 3rd-party plugins, or some other reason? If no, is there some other reason that might lead to few deletions?

Q2 We found that the number of plugins and their dependencies (as declared in the plugin.xml and manifest.mf files) grows more than linearly and that the average size (number of classes) of a plugin has been decreasing. Is this finer-grained modularization of Eclipse an explicit architectural design aim? Has the rapidly growing number of plugins and dependencies caused any development problems or delays, e.g. because it is harder for each developer to keep in their head an overview of the architecture?

Q3 We found that architectural changes (plugin and dependency additions and removals) were done (a) mostly in early milestones, (b) some in the release candidates but freezing the architecture in the last release candidates, and (c) hardly ever in service releases. Is that an explicit architectural change process you follow?

Q4 We found only one cyclic static dependency during the whole history, existing from 2.1 until 2.1.3. The cycle is between 3 plugins ui, ui.editors, ui.workbench.texteditor in the platform feature. The cycle was broken in 3.0. Is the absence of cyclic dependencies an explicit aim of yours?

Q5 Release 3.0 had the biggest growth in number of dependencies; and Release 3.1 is the only one in which that number decreased. Was that reduction of dependencies an explicit aim of release 3.1, i.e. an architectural cleanup after 3.0, to reduce the complexity of the architecture?

Q6 We found the following about extension points.
- Most new extension points are added to existing plugins (rather than appearing with new plugins). Is this to 'open up' existing functionality? Why tend new plugins not to have extension points?
- Extension points are never deleted by themselves, they disappear when their plugin is deleted. Is this to keep backward compatibility?

- Those extension points not used by the SDK itself more than duplicated in 3.4 but then were deleted in 3.5. What is the reason for such big changes?

Q7 We clustered plugins according to whether they were created, deleted, or had their dependencies or extension points changed in the same release. Plugins closer together have more co-changes. We found that on average plugins in different subsystems (org.eclipse.ui.*, org.eclipse.pde.*, etc.) are closer together than those in different features, i.e. there are more co-changes across subsystem boundaries than across feature boundaries. This finding means that it might be slightly harder to implement changes if developers are allocated to subsystems rather than to features. Could you briefly comment on this and how work is allocated among the SDK team?

Q8 We also clustered plugins according to how many dependencies there are between them, and did so release by release and saw a trend that the difference between intra-unit (subsystem or feature) and inter-unit clustering is diminishing, i.e. plugin dependencies across subsystems and features tend to be on par with dependencies within subsystems and features. This could be interpreted as a sign of architectural decay or lack of cohesion, as coarse grained modules (subsystems and features) are becoming less the containers of strongly dependent plugins. Could you comment on this? Have you experienced increasing difficulties in coordinating architectural changes across sub-teams? Or are subsystems and features not very important for architectural design, plugin modularisation, and work allocation?

Q9 We have found an architectural core (set of plugins and dependencies) that remains unchanged since 1.0. Is this something you are aware and hence avoid making architectural changes there? Was it designed from the start to be a stable core?