Co-evolution of the Eclipse Framework and its Third-party Plug-ins

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duin, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 4 juli 2013 om 16.00 uur

door

John Businge

geboren te Fort Portal, Oeganda

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. M.G.J. van den Brand
en
prof.dr.ir. T.P. van der Weide

Copromotor:
dr. A. Serebrenik

Co-evolution of the Eclipse Framework and its Third-party Plug-ins

John Businge

Promotor: prof.dr. M.G.J. van den Brand (Eindhoven University of Technology)
prof.dr.ir. T.P. van der Weide (Radboud University Nijmegen)

Copromotor: dr. A. Serebrenik (Eindhoven University of Technology)

Additional members of the core committee:

prof.dr. J.J. Lukkien (Eindhoven University of Technology)
prof.dr. R. Lämmel (Universität Koblenz-Landau)
prof.dr.ir J. Visser (Radboud University Nijmegen)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2013-09.

A catalogue record is available from the library of the Eindhoven University of Technology
ISBN: 978-90-386-3380-0
Accompanying data used in the thesis is available via the following DOI:
doi:10.4121/uuid:ce5e73ba-4087-4a7a-afb1-e442b4b6c0ec

Printed by the print service of the Eindhoven University of Technology

Acknowledgements

No problem can withstand the assault of sustained thinking.

(Voltaire)

I do believe that no goal can ever fail to be achieved when one puts his mind to it. In the last four years I enjoyed working on the research presented in this thesis. When I visited the Netherlands in May 2009, my promoter Mark van den Brand and my co-promoter Alexander Serebrenik asked me what I would like to do. I told them that I would like to do my research on open-source software. My promoters then suggested for me to research on the topic *Co-evolution of the Eclipse Plug-in Architecture and its Third-party Plug-ins*. I did not comment on the topic that was suggested to me, but in my mind I was like WHAT?! (☹). I did not have the slightest idea of what they told me. From that day onwards, I put my mind to the suggested research-line and in this thesis the work of my sustained thoughts is presented.

None of this would have been possible without the financial support I acquired from a number of sources. First and foremost, I would like to thank the Dutch government through the NUFFIC project for having offered me part of the funding for my PhD. I would also like to thank my second promoter Theo van der Weide who helped me to secure more funding from the NUFFIC project again. Furthermore, I would like to thank my promoter Mark van den Brand who secured funding from the MDSE group of the TU/e so that I can complete my PhD studies.

Regarding my research, I owe many thanks to a number of people: First of all I would like to thank Mark van den Brand again. Mark, many thanks for giving me the opportunity to carry out my research under your supervision. You gave me guidance and always gave me the freedom to do what I wanted. Second, I would like to extend my sincere thanks to Alexander Serebrenik. Alexander, I enjoyed working with you during the course of my research. Your efforts regarding mentoring me have been enormous. I was always motivated by your quick feedbacks whenever I would send you my work. Sometimes I would send you a 10 page pdf at 8pm and by midnight the feedback is already in my inbox ☺ (not to mention with comments literally on every line especially at the beginning of my PhD ☺). Mark and Alexander, I think we make a good paper writing team. I look forward to keep working with you in future. Third, to Koo Rijpkema, I do owe you many thanks for the help on statistics. In the thesis I use a body of statistical

methods most of which you helped me to understand. Many thanks to Tom Verhoeff, Ruurd Kuiper, Anton Wijs, Bogdan Vasilescu, and Marcel van Amstel who helped me with proof reading my papers. Finally, thanks to Martijn Klabbers who helped me with designing the questionnaire used in Chapter 7.

I would also like to extend my special thanks to the Eclipse developer community. The openness of the Eclipse community helped me carry out my research using the developers source code. Special thanks go to Markus Gebhard, an Eclipse plug-in developer. Markus, I owe you many thanks for helping me with my research. The few e-mails I exchanged with you when I was beginning this research helped me focus this thesis. I also would like to thank Mike Milinkovich and Wayne Beaton who helped me get respondents for my survey I used in Chapter 7.

I would like to thank the members of the my thesis reading committee for their time and valuable feed back: Johan Lukkien from Eindhoven University of Technology, Ralf Lämmel from Universität Koblenz-Landau, and Joost Visser from Radboud University Nijmegen. I would also like to thank Gerard Renardel de Lavalette for accepting to be part of the opposition team during my PhD defense.

Many thanks to my colleagues who helped me in different ways in preparing my thesis. My officemates Amina Zawedde, Jackline Ssanyu and Arjan van der Meer. I enjoyed sharing an office with each and everyone of you and the formal and informal discussions we had. Arjan, thank you for your opinions regarding a few questions that I asked you. Many thanks to Luc Engelen who helped me to design the cover of my thesis. Finally, I owe many thanks to Sacha Claessens who helped process my travels from time to time in moving to and from Uganda as well as to the conferences I attended. Because of my frequent travels, sometimes things would not go well and I disturbed you, but still you did not hesitate to help me out in my next travels.

I also want to thank my close friends that I used to joke with, most of the time online as I was very far from them. Peter Babu, Simon Byamukama, Richard Kateera, Ryan Mugabe, Moses Mujuni, Evarist Nabaasa, Lauryn Kahunde, Fred Kaggwa, Reuben Kamugisha to name but a few. You guys used to keep me busy online discussing everything and nothing ☺. Sometimes our conversations would distract me from working on my thesis, but sometimes I needed this distraction to be more productive: “all work and no play makes Johnny a dull boy” ☺. Many thanks Elke Vandamme and Bogdan Vasilescu for accepting to be my Paranymphs.

Last and certainly not least, my sincere thanks go to my family. Dad and Mum, you always encouraged me to keep working whenever I would talk to you on phone. Like you always told me, I am sure you never stopped mentioning me in your prayers. To my siblings, Alex, Sarah, Norah, Moses, and Dinah you always kept me motivated whenever I would call you on phone. To my uncles, aunties and cousins, I thank you for your continued support. Special thanks go to my uncle, Peter Abigaba. Peter, you have never stopped loving, inspiring, and encouraging me. My only request is to please keep it up.

John Businge
April 2013

Table of Contents

Chapter 1

Introduction

Developing an application faster without, compromising its quality, which is easily maintainable and extensible, is a dream for every software developer, or a company involved in software development. Over the last decades, a number of software development paradigms and software technologies have been introduced, mainly by the quest to help software developers reduce the complexity in software systems, build software faster, and deliver more value to the end-users. An example of a software paradigm that helps reduce software complexity, is object-oriented development [28]. An example of a software technology that helps build software faster is software reuse [61, 105].

The object-oriented paradigm provides techniques for reducing software complexity by introducing concepts such as abstraction, encapsulation, aggregation, inheritance and polymorphism [28]. Unlike the procedural software development paradigm that is based on algorithms and procedures, the object-oriented paradigm emphasizes binding of data structures with methods that operate on the data. The object-oriented paradigm allows programmers to use the language effectively to model and solve real-world problems. The programmer can also split problems into (small), manageable fragments of code (modules) where the principle of encapsulation insulates developers from having to know the implementation details.

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch [61]. Software reuse provides a foundation of improving the way software is developed over its life-cycle. Software reuse fosters modularity and interoperability of applications. The potential benefits of software reuse range from decreased development time and increased product quality to improved reliability and decreased maintenance costs [105]. A reusable component may be code, but bigger benefits of reuse come from a broader and higher-level view on what can be reused [105]. Software specifications, designs, tests cases, prototypes, documentation, frameworks and templates, are all candidates for reuse.

Characteristics of the object-oriented paradigm and the technologies that support successful reuse seem to complement each other. When object-oriented paradigm and software reuse technology are integrated together, one can achieve increased modularity

and interoperability of applications [105]. Object-oriented application frameworks are interesting technologies that are based on both object-oriented paradigm and fostering reuse practices [105].

1.1 Object-oriented Application Frameworks

Object-oriented application frameworks were invented at the end of the 1980s [19]. An application framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [55]. Frameworks can be defined in a large variety of domains, such as user-interfaces, operating systems, fire-alarms systems and process control systems. Frameworks can exist as Open Source or commercial software. Example of Open Source frameworks are those that exist as part of the Mozilla, OpenOffice.org, GNOME, KDE, NetBeans and Eclipse projects. Examples of commercial frameworks include: MacApp$^{\delta}$ (Apple Computer) and OWL• (Borland International). According to Johnson and Foote [55], an object-oriented framework is a set of classes that embodies an abstract design for solutions to a family of related problems. Frameworks provide a standard structure or template of a working program. For example, a framework can provide support or default behavior for drawing windows, scrollbars and menus for an application. The following notions related to frameworks, which were introduced in [19], will be frequently used:

- *Framework core design*: The framework core design comprises both abstract and concrete classes in the domain of the framework. The concrete classes are intended to be invisible to the framework user and the abstract classes are intended to be either invisible to the framework user or to be subclassed by the framework user.
- *Framework internal increments*: These are additional classes to the framework core design that make the framework usable. The internal increments consist of classes that capture common implementations of the core framework design. The two common framework internal increments are: 1) subclasses representing common realisations of the concepts captured by the superclasses; 2) a collection of (sub)classes representing the specifications for a complete instantiation of the framework in a particular context.
- *Application*: This is a software system composed of one or more core framework designs, each framework's internal increments (if any) and an application-specific increment.
- *Application specific increments*: This is the part of the application comprising specific classes and objects whose functionality could not be found in the reusable classes of the framework.

The building blocks (reusable components) of object-oriented application frameworks can also be informally defined as semi-complete applications that can be specialized to produce custom applications [55]. The presence of the reusable components provided by the framework influences the development process of an application. The following phases are involved in framework-centered software development: *framework development*, *framework usage*, *framework maintenance and evolution*, and *co-evolution of the framework and its applications*.

1.1.1 Framework Development

Developing a framework is somewhat different from developing an application. The framework developer has to have a deep understanding of the domain of the envisaged applications and should also anticipate needs of future applications [19, 79]. The development of the framework goes through a number of phases that include: domain analysis, framework design, framework implementation, framework testing, and documentation [19].

- *Domain analysis*: This phase aims at describing the domain to be covered by the framework [93]. To capture the requirements, one may refer to previously developed applications in the domain, domain experts, and existing standards for the domain. The result of the activity is a domain analysis model, containing the requirements of the domain, the domain concepts, and the relations between those concepts.

- *Framework design*: This phase takes the domain analysis model as input. The designer then decides on a suitable architectural style underlying the framework that defines the top level design of the framework. Examples of architectural styles include: client/server, component-based architecture and object-oriented [2]. During the framework design phase, the top-level framework design is refined and additional classes are designed. Results from this activity are the functionality scope given by the framework design, the framework's reuse interface, design rules based on architectural decisions that must be obeyed, and a design history document describing the the encountered design problems and the solutions selected.

- *Framework implementation*: This phase concerns the coding of the *framework core design*, i.e., abstract and concrete framework classes, and the *framework internal increments*, i.e., additional classes that make the *framework core design* more usable.

- *Framework testing*: This is a way to prove whether the framework provides the intended functionality and also to evaluate the usability of the framework. Compared to testing applications, i.e., deciding whether an application is usable or not, testing a framework is far from trivial. Johnson and Russo [56] state that the only way to find out if something is reusable is to reuse it. For frameworks, this implies developing test applications. Testing the framework with the test applications aims at deciding whether the framework needs to be redesigned or if it is sufficiently mature for release.

- *Framework documentation*: Unlike the documentation of traditional software systems that is intended for one type of user, i.e., the end user, framework documentation targets both the end user and the application developer. In addition to the user manual describing how the framework works, the framework application developer also requires a document explaining the functionality of the reusable interfaces provided by the framework.

1.1.2 Framework Usage

Without applications that use the framework, it is not worth investing resources in developing a framework. The procedure followed when developing framework-based applications is similar to that used in developing traditional software applications with a few adjustments [19].

- *Requirements analysis*: This phase, like when developing traditional applications, is aimed at collecting and analyzing the requirements of the application to be built, using one or more framework(s).
- *Conceptual Architecture*: Based on the results of the requirement analysis phase, the conceptual architecture is developed. The adjustments made in developing framework-based application of this phase include: the association of functionality to the reusable components, the specification of the relationships between them, and the organization of the collaboration between them.
- *Selection of framework(s)*: Based on the conceptual architecture developed for the application, one or more frameworks are selected based on the functionality to be offered by the application as well as the non-functional requirements.
- *Defining application-specific increments (ASIs)*: Unlike the traditional applications where all classes are designed from scratch, framework-base applications have got two types of classes (see beginning of Section 1.1). After defining the classes of the application that are going to reuse the functionality from the selected framework(s), the remaining functionality with respect to the requirements of the application is defined for the application itself, i.e., the *application specific increments*.
- *Design and implementation of ASIs*: After the ASIs are defined, there is need to design and implement them. The difference between ASIs and classes for traditional applications is, when designing ASIs, developers are advised to follow the design rules stated in the framework documentation of one of the frameworks, if more than one selected.
- *Testing application*: Before the ASIs and the framework-based classes are put together into the final application, the ASIs need to be tested individually to simplify debugging and fault analysis in the resulting application. Finally, the complete application is verified and tested according to the application's original requirements.

1.1.3 Framework Maintenance and Evolution

Due to the high initial investment in the development effort and being a long-lived entity, a framework has to evolve over time and needs to be maintained. Frameworks can be described as E-Type systems [75]. An E-Type system can be defined as a real world system, i.e., components of real world processes [65]. Lehman states that evolution and maintenance of E-Type systems is a necessity, otherwise they become less satisfactory or unstable [65]. During the initial stages of the development, the framework has to undergo several iterations to remove the shortcomings in order to become reusable [83]. After the first official release, evolution and maintenance of the framework have to continue because of two main factors: *fixing faults found* and *domain change*.

After the first official release of the framework, applications will be built to reuse the functionality provided by the framework. During the reuse, framework-related faults may be found that require changes to the framework. The faults could either be due to the fact that the domain covered by the framework is incomplete, i.e., failure to capture the domain in the domain analysis, or due to the fact that the application does not match the intended framework domain, i.e., a developed application whose requirements have badly fit the domain covered by the framework. In both cases, the framework will

generally be changed, forcing the organization to face the problem of selecting maintenance strategies [19].

Furthermore, as we stated in Section 1.1.1, the domain for developing a framework should be carefully defined in the domain analysis phase. Unfortunately, the domain can never be exhaustive, but is often weakly defined and unstable over time [19]. If faults resulting from application mismatch are discovered in the framework, the framework domain can be expanded to cater for the mismatch. When the domain changes frequently, the framework will be more difficult to reuse, and even worse, if not maintained, it can be completely useless after a rather short time span [19]. For the framework to be continuously usable, the domain changes captured by the framework have to be adapted during the evolution and maintenance of the framework.

Given the situation that the framework has to be changed, either because of *fixing faults found* or *domain change*, a maintenance strategy is necessary, i.e., the developers of the framework need to decide whether to *redesign* the framework or to do a *work-around* for the specific application(s) to overcome the problem [19]. If one decides to *redesign* the framework, the problematic application(s) have to be delayed until a new version of the framework is available. In this case, the framework developers are forced to maintain two versions of the framework, i.e., the earlier framework version since there may exist applications based on this version and the redesigned version for future applications to be developed. In the case of performing a work-around in the application, there will be no additional maintenance to the framework since there is only a single framework. Instead, the maintenance problem will occur for the developed application. However, the choice of *work-around* may not be feasible for successful and widely adopted frameworks, as they may have so many applications reusing their functionality.

1.1.4 Co-evolution of a Framework and its Applications

Frameworks are constantly evolving as a result of improving the quality of the existing interfaces and also as a result of extension by introducing new features. Frameworks usually provide two types of interfaces to the applications that reuse the framework functionality: stable interfaces and unstable interfaces. Examples of frameworks that make this distinction are Eclipse [5] and Netbeans [104]. Unstable interfaces are usually accessible but reusing them is discouraged by framework developers as these interfaces may change arbitrarily and without notice in new versions of the framework. Stable interfaces isolate framework-based applications from changes, under the assumption that the framework developers limit themselves to only extending the interface as opposed to changing it.

In practice, however, new versions of successful and widely adopted frameworks frequently violate the assumption of preserving the interfaces that are labeled stable. This makes applications that only use the stable interfaces of the framework to fail when ported to new releases of the framework [115]. For this reason, if framework developers are to avoid breaking any client code, they have to refrain from making changes to the stable interfaces, even when these changes might improve the quality and performance of the framework. The framework developers have to make a tradeoff between improving the quality and performance of the stable interfaces in a new release and breaking some of the client code. Normally, framework developers have to violate the latter so as to evolve the framework and keep it alive.

By ignoring the incompatibilities, if applications are to benefit from the improved quality and added functionality in the new versions of the framework, then they have

co-evolve with the framework. Applications have to evolve due to the following possible reasons: First, evolve in order to benefit from improved quality and new functionality from the framework. Second, from the survey we report in Chapter 7, we discovered that developers try to keep their plug-ins working with the latest Eclipse release so as to keep their customer base as well as to avoid missing out on new customers. This is because customers are more confident about the plug-in if it works with the latest release of Eclipse. For this reason, developers are forced to co-evolve their plug-in with every new release of Eclipse even if they are not interested in the improved quality and new functionality from the framework. Lastly, evolve as a result of specific requirements from the stakeholders of the application that can not be found in the reusable functionality from the framework.

Although the reuse of unstable interfaces by application developers is discouraged, it is common that these unstable interfaces are used. For example, in the study [23] we discovered that about 44% of Eclipse third-party plug-ins on SourceForge use at least one unstable interface from the Eclipse SDK framework. Therefore, if applications that use the unstable interfaces are to co-evolve with the framework to benefit from the improved quality of the interfaces, they also have to fix any existing interface incompatibilities.

1.1.5 Advantages of Framework-based Applications

The overall benefit of frameworks is that they enable a higher level of code and design reuse than what is practical with other application design approaches [19]. The benefits of reusing functionality from the framework are not gained from the first of second use, but rather gained over time by multiple uses of the technology. Below are some of the major benefits gained by reusing the functionality of the framework borrowed from [44]:

- *Modularity*: Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.

- *Reusability*: The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer's productivity, as well as enhance the quality, performance, reliability, and interoperability of software.

- *Extensibility*: A framework enhances extensibility by providing explicit hook methods [86] that allow applications to extend their stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

- *Inversion of control*: The run-time architecture of a framework is characterized by an *inversion of control*. This architecture enables the canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's

dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end-users or packets arriving on communication ports).

1.1.6 Disadvantages of Framework-based Applications

Although the reuse of the functionality provided by application frameworks simplifies the designing of new applications and maintenance of the existing ones, it implies that the design and implementation of these applications heavily depends on the frameworks they reuse. For this reason, the software development industry still faces issues along the development cycles [4,115]. Below we state some of downsides of building framework-based applications:

- *Development process management*: As stated in Section 1.1.2 developing framework-based applications undergoes a similar development process as traditional applications. Developing traditional applications starts with specification of the work to be done. Explicitly writing such specifications in framework-based applications is more challenging. One has to clearly identify the two types of the application entities: 1) those that require reusing framework core designs and/or framework internal increments, and 2) those that should be developed from scratch, i.e., application specific increments [19]. The difficulty of clearly identifying the two types of application entities makes the development process of framework-based applications ad-hoc [19].

- *Framework domain applicability*: When developing a framework-based application, the challenge is to clearly determine whether the application matches the domain of the framework. Deciding applicability of the domain to the application is rather complex as it contains several dimensions. One dimension is interoperability, where aspects such as co-existence of other applications is discussed, for example, if there are restrictions in the usage of the other applications. Another dimension could be, for example, aspects such as performance, fault-tolerance, and capacity. An examples could be a framework intended for batch processing and not for real-time data processing. Based on how the domain is understood, the developer has to decide with a reasonable degree of confidence on whether to build a framework-based application or to build it from scratch [19].

- *Effort estimation*: Estimating the effort spent on developing a framework-based application can be difficult since the developer spends a considerable amount of time understanding the framework and very little actual coding is done. This makes the traditional effort estimation techniques based on metrics, e.g., number of produced lines of code, complexity and function points, inadequate in the framework-based applications [12, 19]. Furthermore, although complex applications can be built very fast by reusing the framework, it can be difficult to foresee if a given functionality required by the application is fully supported by the framework. If the functionality is not fully supported, there is a risk of potential mismatch, where in this case, the resulting effort may be even larger than developing the application from scratch. Therefore, determining accurate effort estimations in developing framework-based application is much more challenging than in traditional applications [19].

- *Application evolution*: During evolution, framework-based applications may be subject to two independent, asynchronous, and potentially conflicting evolution processes [115]. The first evolution process is driven by the need to co-evolve the application with every new release of the framework in order to use new functionality and benefit from the improved quality of the interfaces the application uses. When the applications are ported to the new releases of the framework, there is a danger of incompatibilities with the new version of the framework. The second evolution process is motivated by the continued evolution of requirements from stakeholders of the application. If the specific requirements are not fully supported by the framework, the developer has to decide on whether to reuse the framework of design ASIs.

1.2 The Eclipse Framework

The studies addressed in this thesis focus on the Eclipse application framework [45]. Eclipse is an Open Source, extensible plug-in system, an integrated development environment (IDE) as well as an application framework. Eclipse is written in Java and by means of various plug-ins, other programming languages including Ada, C, C++, COBOL, Fortran, Haskell, Perl, PHP, Python, R, and Ruby are used. In addition, a small part of the Eclipse implementation requires specific code for each of the platforms (e.g., Windows, Mac OS, Linux) to improve the interoperability and performance. Tool builders contribute to the Eclipse framework by wrapping their tools in pluggable components (update sites for the plug-ins into the framework), called *Eclipse plug-ins*, which conform to the Eclipse's plug-in contract. Plug-ins are bundles of code and/or data that contribute functionality to a software system. The Functionality can be contributed in the form of code libraries (Java classes with public application programming interfaces (APIs)), platform extensions, or even documentation [18, 36].

1.2.1 Eclipse plug-ins

We categorize the plug-ins in the Eclipse framework into three main groups: Eclipse core plug-ins, Eclipse extension plug-ins, and Eclipse third-party plug-ins.

- *Eclipse core plug-ins (ECPs)*: These are plug-ins present in and shipped as part of the Eclipse SDK. In this thesis, the ECPs and Eclipse SDK are interchangeable. The ECPs provide core functionality upon which all plug-in extensions are built. The ECPs also provide the runtime environment in which other plug-ins are loaded, integrated, and executed. The fully qualified names of ECP packages start with `org.eclipse`. Examples of ECPs include Java development tools (JDT), Standard Widget Toolkit (SWT) and Platform runtime and resource management (Core) [45].

- *Eclipse extension plug-ins (EEPs)*: These are plug-ins built with the main goal of extending the Eclipse SDK. Most EEPs are large, generic, applications frameworks with tool plug-ins to build other specialized applications. Like the ECPs, fully qualified names of EEP packages start with `org.eclipse`, but as opposed to the ECPs, the EEPs are not shipped as part of the Eclipse SDK. Popular EEPs include: J2EE Standard Tools [9], Eclipse Modeling Framework [8], and PHP Development Tools [85].

- *Eclipse third-party plug-ins (ETPs)*: These are the remaining plug-ins. The size of these plug-ins ranges from very large application frameworks to small specialised applications. Unlike ECPs and EEPs, the package names of ETPs do not have a prefix `org.eclipse`. All ETPs use at least some functionality provided by ECPs but also may use functionality provided by EEPs. The ETPs are also sometimes referred to as Eclipse products/solutions and sometimes Eclipse extensions.

The main focus of the studies in this thesis is on the Eclipse SDK and ETPs. There are several reasons for this focus: First, Eclipse SDK provides core functionality upon which all plug-in are built. Second, the Eclipse SDK has been undergoing substantial evolution for more than a decade now and therefore, we were interested to see to what extent its evolution affects the applications that reuse the Eclipse SDK. Third, Eclipse SDK is extensively used: by the time we started this research in May 2009, the Eclipse marketplace[1] listed around 1,000 plug-ins reusing the SDK and currently it lists 1,470 plug-ins. Finally, as opposed to EEPs that are predominantly application frameworks, i.e., large software systems, and also covering a small domain, we have chosen the ETPs because we wanted to study how the Eclipse SDK is being used since ETPs come in a variety of domains and sizes. Wermelinger, Yijun and Lozano have also extensively analyzed ECPs [109].

1.2.2 Eclipse Interfaces

The Eclipse has two main types of interfaces, i.e., visible features that can be reused, it provides to applications that reuse its functionality: non-APIs and APIs [5, 37, 38].

- ***Eclipse non-APIs ("bad"):*** The non-APIs, which we also call *bad interfaces*, are internal implementation artifacts that are found in a package with the substring "internal" in a fully qualified package name according to Eclipse naming convention [37]. The internal implementations include public Java classes or interfaces, or public or protected methods, or fields in such a class or interface. Users are strongly discouraged from using any of the non-APIs since they may be unstable [38]. Eclipse clearly states that clients who think they must use these non-APIs do it at their own risk as non-APIs are subject to arbitrary change or removal without notice. Eclipse does not usually provide documentation and support to the non-APIs.

- ***Eclipse APIs ("good"):*** The APIs, which we also call *good interfaces*, are the public Java classes or interfaces that can be found in packages that do not contain the segment "internal" in the fully qualified package name, or a public or protected method, or field in such a class or interface. Eclipse states that, the APIs are considered to be stable and therefore can be used by any application developer without any risk. Furthermore, Eclipse also provides documentation and support for these APIs.

1.2.3 Co-evolution of the Eclipse SDK and its Applications

Since its inception in 2001 the Eclipse SDK framework has been constantly evolving. At the time of writing this thesis (November, 2012), the Eclipse SDK has produced 14 major releases and 25 milestone releases presented in Table 1.1. The research carried out in this

[1] http://marketplace.eclipse.org

Major Release	# Milestones	API change	Release Date
4.2	1	non-breaking change	Wed, 27 Jun 2012
4.1	2	non-breaking change	Mon, 20 Jun 2011
4.0	2	breaking change	Tue, 27 Jul 2010
3.7	2	non-breaking change	Mon, 13 Jun 2011
3.6	2	non-breaking change	Tue, 8 Jun 2010
3.5	2	non-breaking change	Thu, 11 Jun 2009
3.4	2	non-breaking change	Tue, 17 Jun 2008
3.3	3	non-breaking change	Mon, 25 Jun 2007
3.2	2	non-breaking change	Thu, 29 Jun 2006
3.1	2	non-breaking change	Mon, 27 Jun 2005
3.0	2	breaking change	Fri, 25 Jun 2004
2.1	3	non-breaking change	Thu, 27 Mar 2003
2.0	2	breaking change	Thu, 27 Jun 2002
1.0	0	breaking change	Wed, 7 Nov 2001

Table 1.1: Eclipse releases. # Milestones–Represents the number of milestone releases before the next major release.

thesis is based on 11 of the 14 major releases, SDK 1.0—3.7 (see column *Major Release* in Table 1.1).

As the Eclipse SDK framework evolves from one release to the next, it makes changes to its interfaces during maintenance activities and also introduces new interfaces to extend the framework. As stated in Section 1.2.2 the Eclipse SDK framework, which is part of the general Eclipse framework, has got two types of interfaces: APIs and non-APIs. Since the non-APIs are undocumented and unsupported, clients encountering incompatibilities in a new SDK release, must dig into the Eclipse source code to understand how the non-APIs have changed. For APIs, changes made to the APIs are usually documented and communicated to the clients that use them. The changes made to APIs are categorized into two types: API breaking changes and API non-breaking changes (see column *API change* in Table 1.1) [11].

- *API breaking changes*: When API breaking changes are made from one Eclipse SDK release to the next, compatibility of the applications that depend on these changed APIs is broken in the newer Eclipse SDK releases [11, 38]. During the evolution of the Eclipse SDK, when API breaking changes are made from one release to the next, the first number of the release is incremented by one and the second number is reset to zero. For example in Table 1.1, there were API breaking changes from Eclipse SDK release 2.1 to 3.0. Examples of API breaking changes include, e.g., change of a public method name and adding or deleting formal parameter from a public method [38].

- *API non-breaking changes*: When API non-breaking changes are made from one Eclipse SDK release to the next one, compatibility of applications that depend on these changed APIs is preserved in the newer Eclipse SDK releases [11, 38]. During the evolution of the Eclipse SDK, from one release of the SDK to the next, API non-breaking changes are signified by an increment by one in the second number of the release number. For example in Table 1.1, there were only API non-breaking changes from Eclipse SDK 3.0—3.7. API non-breaking changes include compatible API

changes, performance changes or major code rework [11]. Examples of compatible API changes include, e.g., deleting a non-public type (class/interface) from API package, changing a class from abstract to non-abstract.

The results of the empirical study we carried out in Chapter 6 are concurrent with what Eclipse states in terms of the API breaking/non-breaking changes, i.e., API breaking changes took place between SDK 2.1 and 3.0 and none took place in SDKs 3.0—3.7.

1.3 Problem Statement

Based on the above discussion, we formulate the problem statement that is central to this thesis as follows:

The Eclipse SDK framework is a popular, open-source and large complex software system that provides functionality to ETPs in form of reusable interfaces. The framework is constantly evolving to improve on the quality of the existing interfaces it provides and also to extend the framework by introducing new interfaces. As stated in Section 1.1.6 for frameworks in general, the ETPs that reuse Eclipse SDK functionality are subject to two independent, asynchronous, and potentially conflicting evolution processes.

The first evolution process is driven by the need to co-evolve the ETP with every new release of the Eclipse SDK framework to use new functionality and also benefit from the improved quality of the existing interfaces the ETP uses. When the ETP is ported to the new releases of the framework, there is a danger of incompatibilities with the new release of the Eclipse SDK framework.

The second evolution process is motivated by the continued evolution of requirements from stakeholders of the ETP. If the specific requirements are not fully supported by the framework, the developer has to decide with a reasonable degree of confidence on whether to reuse the framework or to design new application specific increments.

Based on the two evolution processes of the ETP and the respective challenges, we felt the need to study Eclipse interface usage by the ETPs so as to quantify/qualify the challenges faced by the developers of the ETP. The lessons learned from this study can provide valuable information, in particular to the providers interfaces, i.e., Eclipse SDK developers, and users of the interfaces, i.e., ETP developers, in co-evolving the Eclipse SDK eco-system and in general to other frameworks eco-systems as well.

1.4 Research Questions

We formulated a number of research questions addressing the problem stated in Section 1.3. The central research question is:

RQ: *What are the challenges faced by framework-based application developers in co-evolving their application with the framework they reuse?*

The central research question is split into a number of more specific research questions. Each of the more specific research questions is addressed in the remainder of this thesis.

As a starting point in addressing the main research question we were interested in investigating the evolution of the framework-based applications with respect to known evolution of traditional applications. We formulated the following research question.

RQ1: *How does the evolution ETPs with respect to the interfaces they use from the framework compare with the known general evolution of software systems?*

After investigating how the ETPs evolve with respect to the framework interfaces they depend on, we were interested in investigating the extent of the ETPs dependency of the two types of Eclipse interfaces described in Section 1.2.2. We formulated the following research question.

RQ2: *To what extent do ETPs depend on the two types of interfaces Eclipse SDK framework provides?*

To get further insights of the usage of the two different types of interfaces, we wanted to understand the potential risks involved in evolving the ETPs with the new Eclipse SDK releases. The following research question was formulated.

RQ3: *How does the compatibility of ETPs that depend solely on Eclipse APIs compare to that of ETPs that depend on at least one Eclipse non-API in new Eclipse SDK releases?*

Based on the incompatibilities of the ETPs in new Eclipse SDK framework releases, we were interested in predicting the compatibilities of the ETPs in new Eclipse SDK releases.

RQ4: *Can prediction models based on the usage of the Eclipse interfaces in the ETPs help in predicting the compatibility of an ETP in a new Eclipse SDK release?*

The investigation of Eclipse interface usage in research questions R1—R4 were all based on the source code analysis. We also introduced a different view point in investigating the Eclipse interface usage in research question RQ5, i.e., based on state-of-practice of Eclipse interface usage by the developers. The findings of the RQ1—RQ4 are discussed in comparison to the findings of RQ5.

RQ5: *What is the state-of-practice of Eclipse interface usage by the developers of ETPs?*

1.5 Outline

In this thesis, we carry out a series of empirical studies addressing the Eclipse SDK interface usage in ETPs. This section gives an outline of the chapters in this thesis. Additionally, we summarize our contributions in terms of how we quantified/qualified the challenges faced by ETP developers and suggest possible solutions to these challenges. The work in the Chapters 2, 4, 5, 6 and 7 is based on peer reviewed publications at software engineering conferences and workshops. The work in Chapter 5 is based on a peer reviewed conference and a peer reviewed journal accepted for publication. In Chapter 3 we explain how we collected data used in Chapters 4–6. While all the papers have distinct core contributions, there is some redundancy in the introduction of the background material, definitions, and data collection. Some of this redundancy has not been eliminated to make it possible to read the extended and revised papers independently. Below we indicate the relationship between parts of this thesis to our publications.

- **Chapter 2: (RQ1) Evolution of Eclipse ETPs**. In the quest to determine how the evolution of the ETPs, with respect to Eclipse interface usage, compares to the evolution of traditional software applications, this chapter investigateS the applicability of Lehmans laws [68] on the ETPs. To our knowledge, we are the first to study the applicability of these laws on software systems that exhibit a constrained evolution process. This chapter is based on the following publication.

  [22] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. 2010. An empirical study of the evolution of Eclipse third-party plug-ins. *In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) (IWPSE-EVOL '10)*. ACM, New York, NY, USA, Pages 63–72.

- **Chapter 3: Big Data Collection**. In this chapter, we explain how we collected the data that we used in Chapters 4–6.

- **Chapter 4: (RQ2) Eclipse API usage: the good and the bad**. In this chapter, we investigate Eclipse interface usage by the ETPs based on whether they use bad interfaces from Eclipse or not. Based on the findings, we provided the characteristics exhibited by the ETPs that depend solely on Eclipse good interfaces and the ETPs that depend on at least one bad interface. Additionally, the findings in this chapter are a starting point for further investigations in chapters 4, 5 and 6. This chapter is based on the following publication.

  [23] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Eclipse API usage: The Good and The Bad, *Sixth International Workshop on Software Quality and Maintainability*, March 27-30, 2012, in University of Szeged, Szeged, Hungary, Pages 54–62.

- **Chapter 5: (RQ3) Survival of Eclipse ETPs**. In this chapter, we investigate the compatibility of the ETPs in new Eclipse SDK releases, based on whether the ETPs use bad interfaces from Eclipse or not. The findings of the investigation reveal that the use of bad interfaces from Eclipse is the major cause of the ETPs' incompatibilities in new Eclipse SDK releases. This chapter is based on one publication and on a manuscript accepted for publication.

  [26] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Survival of Eclipse Third-party Plug-ins, *28th IEEE International Conference on Software Maintenance (ICSM'12)*, IEEE Computer Society Press, 2012, Riva del Garda, Trento, Italy, Pages 368–377.

  [21] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Eclipse API usage: The Good and The Bad. *Software Quality Journal. Extension of* [23], *Accepted*.

- **Chapter 6: (RQ4) Compatibility prediction of ETPs in new Eclipse releases**. In this chapter, we propose prediction models for predicting compatibility of Eclipse-based application, that is developed on top of a given Eclipse SDK release, in a new Eclipse SDK release. We validated the built models, where the results show high precision, recall and accuracy of the prediction experiments. This chapter is based on the following publication.

[25] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Compatibility Prediction of Eclipse Third-Party Plug-ins in New Eclipse Releases, *12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, IEEE Computer Society Press, 2012, Riva del Garda, Trento, Italy, Pages 164–173.

- **Chapter 7: (RQ5) Analyzing the Eclipse API usage: putting the developer in the loop**. Due to a number of unanswered questions in our previous chapters, in this chapter we conduct a developers' survey with the aim of achieving a clear picture on the state-of-the-practice of the Eclipse interface usage. The findings of this chapter indeed reveal a clear picture on the state-of-the-practice of the Eclipse interface usage by the Eclipse-based application developers. This chapter is based on the recent work that is yet to be published.

[27] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Analyzing the Eclipse API Usage: Putting the Developer in the Loop. *17th European Conference on Software Maintenance and Reengineering*, March 5-8, 2013, Genova, Italy. *Accepted for publication*

- **Chapter 8: Conclusions**. In this chapter, we draw conclusions to reflect on our work and we give an outlook of opportunities for future work. All the research in this thesis was conducted by John Businge as the first author and his supervisors as the second authors.

- **Appendices**. The appendices contain additional details on the experiments reported in this thesis.

Chapter 2

Evolution of Eclipse Third-party Plug-ins

Since the inception of Lehman's software evolution laws in the early 1970s, they have attracted significant attention from the research community. However, to our knowledge, no study of applicability of these laws on the software systems that exhibit constrained evolution, i.e., framework-based and general evolution, process has been carried out so far. In this chapter we take a first step in this direction and investigate the constrained evolution of 21 Eclipse third–party plug-ins. We investigate the trends followed by the plug-ins in relation to plug-ins' dependencies on Eclipse over time. The study spans 6 years of the evolution of Eclipse evolving from release 3.0 to release 3.5. Our findings confirm the laws of continuing change, increasing complexity, self regulation, and continuing growth when metrics related to dependencies between the plug-ins and the Eclipse Architecture are considered. Unlike this, the conservation of familiarity and conservation of organizational stability law were not confirmed and the results for the declining quality law were inconclusive. We could not comment on the law of feedback systems since we could not have the necessary data to measure the law. Our overall observation is that the trends observed for constrained evolution are similar to those presented by earlier researches on the general evolution of software systems.

2.1 Introduction

Component frameworks simplify the work of software developers because of their standard development structure provided through their reusable components. A third-party plug-in of a component framework is a software system that on the one hand is built to extend the capabilities of a framework and on the other hand is built to reuse the some of functionality provided by the framework [111]. The ease of development and maintenance of software systems built on a component framework lead to an impressive increase on the number of third-party plug-ins: for instance, at the moment of this writing Eclipse Marketplace `http://marketplace.eclipse.org/` lists more than 1000 third-party plug-ins for Eclipse. However, as the framework evolves due to the need to extend its functionality and improve its quality, the changes frequently affect the API (dependencies)

on which the third-party plug-in relies. As a result, the plug-in may break when ported to the new version of the framework. Hence, the evolution of a plug-in is constrained by two independent, and potentially conflicting processes: 1) it has to evolve to keep up with the changes introduced in the framework (framework-based evolution); 2) it has to evolve in response to the specific requirements and desired qualities of the stake holders (general evolution) [115].

A number of studies on the evolution of software systems have been carried out in the past years [14, 54, 75, 113]. However, the research community has given little attention to potentially conflicting evolution processes in cases where the software systems are built on component frameworks. Therefore, our goal consists in understanding *applicability of traditional results on software evolution to constrained evolution of Eclipse plug-ins*. Like the previous studies that focused on general evolution [14, 54, 75, 113], to study the constrained evolution of software systems, we also employ Lehman's laws of software evolution [64, 66]. To study constrained evolution we focus on framework-based evolution of the Eclipse third-party plug-ins (ETPs) with respect to the Eclipse interfaces the ETPs use. We therefore formulate our research question.

RQ1: *How does the evolution ETPs with respect to the interfaces they use from the framework compare with the known general evolution of software systems?*

The reminder of the chapter is organized as follows. In Section 2.2 we briefly explain the applications used in carrying out the study and state some definitions. In Section 2.3 we discuss we carried out the experiments. In Section 2.4 we discuss the results of the experiments. In Section 2.5 we discuss the possible threats to validity and the ways we countered them. Section 2.6 discusses some of the related work. Finally, Section 2.7 concludes the chapter.

2.2 Data Collection

Our investigation is based upon 21, carefully selected, Open Source ETPs written in Java. 17 of the ETPs were collected from SourceForge [6] one of the popular Open Source repositories (the ones having results in the column *DownL* in Table 2.1) and the remaining ETPs were collected from their pages declared on Eclipse marketplace [3].

The selection of a plug-in is based on the following criteria. First, a plug-in should have dependencies on Eclipse. All the dependencies in a plug-in should indicate a particular Eclipse class/interface being imported and not a group of classes in a package, e.g., plug-ins that had dependencies that include, for example `org.eclipse.foo.*`, were excluded. The reason for the exclusion is that they would affect our results since from our method of data collection we cannot tell which specific class/interface from this package are being used by the plug-in. Second, at least one of the most recent versions of the ETP should have been released in 2009 (corresponding to Eclipse SDK 3.5 release) and one of the earlier versions should have been released in 2006 (corresponding to the release of Eclipse SDK 3.2), or earlier. We consider ETP versions released in 2004 (corresponding to the release year of Eclipse SDK 3.0). Third, the plug-in should have at least 12 or more versions since its first release so that we have fine-grained growth trends to reduce on the noise. Although the Eclipse plug-in architecture has got over 1,000 plug-ins, only 21 of them satisfied our selection criteria.

Table 2.1 shows the corresponding information for each of the 21 ETPs we collected. In addition to the information for each plug-in, we also observed that the ETPs are

Plug-in	Abbr	Yrs	# Vers	DownL	Version		Date	SLOC
AnyEditTools	Any	6	17	–	FV	0.93	07-07-2004	1,842
					LV	2.3.0	04-10-2009	10,237
Acceleo	Acceleo	4	13	–	FV	1.1.0	01-09-2006	172,807
					LV	2.6.0	01-06-2009	213,346
ByteCodeOutline	BCO	6	15	–	FV	1.2.0	08-11-2004	2,769
					LV	2.2.12	20-07-2009	6,918
CheckStyle	Check	6	16	40030	FV	3.3.1	22-01-2004	7,127
					LV	4.4.3	06-08-2009	23,449
Ecelemma	Ecelemma	4	20	311	FV	0.1.1	25-08-2006	4,419
					LV	1.4.3	19-10-2009	8,708
Extended VS Presentation	Extend	4	13	–	FV	1.3.0	01-08-2006	6,830
					LV	1.5.3	24-05-2009	9,429
Quantum Database Utility	Quantum	5	12	298	FV	2.4.5	23-02-2005	30,235
					LV	3.3.8	06-11-2009	73,154
TeXlipse	Tex	6	12	520	FV	1.0.0	16-03-2005	21,005
					LV	1.4.0	04-04-2010	31,327
PyDev	PyDev	5	52	2112	FV	1.0.6	24-04-2006	57,482
					LV	1.5.5	04-03-2010	205,568
Java Hex Editor	Hex	4	31	54	FV	0.0.1	05-05-2006	1,795
					LV	0.3.1	28-06-2009	10,204
Beyond CVS	Beyond	5	16	68	FV	0.5.1	11-11-2005	1,205
					LV	0.8.9	27-02-2010	7,906
My TourBook	Tour	4	26	277	FV	0.8.4	03-01-2007	32,060
					LV	10.3.0	25-03-2010	125,331
Eclim (eclipse + vim)	Eclim	6	28	360	FV	1.1.0	26-12-2005	4,760
					LV	1.5.5	23-02-2010	29,670
Verilog editor	Verlog	7	22	141	FV	0.1.0	13-03-2004	1,251
					LV	0.7.1	09-04-2010	21,248
PHPeclipse	PHP	6	13	1649	FV	1.0.9	28-06-2004	61,907
					LV	1.2.3	09-10-2009	171,647
Eclipse Metrics	Metrics	6	13	72	FV	2.7.0	04-09-2004	4,157
					LV	3.14.0	28-10-2009	7,089
GetText Editor	Gted	4	17	213	FV	0.1.0	06-01-2007	1,060
					LV	1.5.5	22-09-2009	4,602
Green	Green	5	12	343	FV	2.4.0	12-10-2005	11,416
					LV	3.4.0	22-10-2009	20,401
ClearCase	Clear	7	15	208	FV	0.9.11	14-06-2004	6,803
					LV	2.2.1	04-03-2010	9,424
JavaCC	JavaCC	6	15	207	FV	1.5.0	13-11-2005	15,292
					LV	15.16	04-10-2009	17,150
SQL Explorer	SQL	6	17	956	FV	2.2.3	27-03-2005	19,360
					LV	3.5.1SR3	31-01-2010	33,785

Table 2.1: Plug-in information. Key: FV - first version, LV - last version, DownL - average weekly download statistics, – - Download statistics could not be obtained from the respective website

associated with a range of solution domains as stated on their respective pages on Eclipse marketplace [3]. These include: editor, modeling, code management, source code analyzer, testing, UI, database, documentation, editor/IDE, rich client applications, integration, UML, build & deploy, and SCM.

2.3 Experiment Design

Since the the purpose of our study is to explore the evolution of ETPs, the type of study is *exploratory*. The selected case study aims at providing knowledge as a basis to develop further theories. In the discussion of the case study, we give give a detailed explanation about the metrics used, the tools used in metric extraction and the approach used to study the evolution of the ETPs.

2.3.1 Metric Definitions

We are interested in studying the evolution of ETPs with respect to the ETPs' dependencies from Eclipse SDK interfaces. Therefore, we chose to use metrics to help us carry out this study. Below we give definitions of the metrics used:

- *Deps-Tot*: The total number dependencies from Eclipse SDK interfaces in a version on an ETP.
- *Deps-Uniq*: The total number of unique dependencies from Eclipse SDK interfaces in a version of an ETP.
- *Total SLOC*: The total number of source lines of code in a version of an ETP.
- *SLOC-Deps*: The number of source lines of code in a version of an ETP, considering only the .java files having dependencies from Eclipse SDK interfaces.
- *NOF-Tot*: Total number of files (`.java`) in a version of an ETP
- *NOF-Deps*: Total number of files (`.java`) in a version of an ETP, having dependencies from Eclipse SDK interfaces.
- *Add-Deps*: The number of new dependencies from Eclipse SDK interfaces in a new version of an ETP.
- *Del-Deps*: The number of deleted dependencies from Eclipse SDK interfaces in a new version of an ETP.
- *Add-NOF-Tot*: The total number of new files (`.java`) in a new version of an ETP.
- *Mod-NOF-Tot*: The total number of modified files (`.java`) in a new version of an ETP.
- *Del-NOF-Tot*: The total number of deleted files (`.java`) in a new version of an ETP.
- *Add-NOF-Deps*: The number of new files (`.java`) having dependencies from Eclipse SDK interfaces, in a new version of an ETP.

Metric	Tool	Website
Deps-Tot	Windows Grep and Excel	http://www.wingrep.com
Deps-Uniq	Excel	
SLOC-Tot	SLOCCount	www.dwheeler.com/sloccount
SLOC-Deps	SLOCCount	
NOC-Tot	Windows Grep	
NOC-Deps	Windows Grep	
Add-Deps	Excel	
Del-Deps	Excel	
Add-NOF-Tot	DependencyFinder	http://sourceforge.net/projects/depfind/
Mod-NOF-Tot	DependencyFinder	
Del-NOF-Tot	DependencyFinder	
Add-NOF-Deps	DependencyFinder	
Mod-NOF-Deps	DependencyFinder	
Del-NOF-Deps	DependencyFinder	
D_n	JDepend	http://clarkware.com/software/JDepend.html

Table 2.2: Metrics, their corresponding extraction Tools and the location of the tools on the web.

- *Mod-NOF-Deps*: The number of modified files (`.java`) having dependencies from Eclipse SDK interfaces, in a new version of an ETP.
- *Del-NOF-Deps*: The number of deleted files (`.java`) having dependencies from Eclipse SDK interfaces, in a new version of an ETP.
- D_n: The perpendicular normalized distance of an assembly from the main sequence. In a software system, D_n is an indicator of the package's balance between abstractness and stability. A low value (closer to zero) of average and standard deviation of the package's D_n in a software system is an indicator of high quality [73].

The methodology used in extracting the metrics considers a moved or renamed file as deleted file and then added.

2.3.2 Metric Extraction

We looked for available tools to help us extract the metrics presented in Section 2.3.1. Table 2.2 shows the metrics and the corresponding tools used in extracting the metrics from the ETPs.

For some of the metrics, in addition to extracts from the tools, we wrote scripts to extract the metrics from the output of the tools. For example, we wrote scripts to extract the metrics that required DependencyFinder for extraction. The metric SLOC-Deps, we wrote a script that extracted only the files that had dependencies from Eclipse SDK interfaces and used the tool SLOCCount to count for the source lines of code from the files. For the metrics Del-Deps and Add-Deps, we wrote excel formulas that determines the difference between two versions of an ETP to extract added and deleted dependencies.

2.3.3 Lehman's Laws of Software Evolution

Lehman's laws describe the evolution of *E-type* software systems [68]. Similar to the study of "The Evolution of Eclipse" in [75], we consider the ETPs as *E-type* systems.

Indeed, the ETPs are actively used and hence, external pressure demands them to change over time. In our study, we investigate whether the constrained evolution of the ETPs follows Lehman's laws of software evolution [64, 66]. Below we state the Lehman's laws of software evolution with respect to the evolution of our carefully selected ETPs [64, 66].

- *Law I: Continuing Change*: The first law states that, an *E-type* system must be continually adapted to its environment else it becomes progressively less satisfactory [64]. Lehman continues to state that, all programs are models of some part, aspect, or process of the world. Therefore, there must be changes to keep pace with the needs and the potential of changing environment. If they are not, they become progressively less relevant, useful, and cost effective. In studying the law, Lehman, Perry and Ramil in [69] proposed the metric *number of modules handled*, i.e., number of added, deleted, and modified modules, in a software system. The trend followed by the different handled files over time would give and indication of how the software systems are changing.

- *Law II: Increasing Complexity*: The second law states that, as an E-type system evolves its complexity increases unless work is done to maintain or reduce it [64]. Lehman continues to state that the law arises from both local and global phenomena. On the one hand, most local enhancements require, among other things, the insertion of mechanisms such as calls to new procedures or the insertion of objects. This adds branch points to the system, increasing its complexity independent of the way it is defined. At the other extreme, functional extension such as the addition of a new device or a functional subsystem will increase functional complexity of the system as well as its global structural complexity [69]. Lehman, Perry and Ramil in [69] states that, absence of appropriate metrics consistently applicable across the systems they studied, complexity change could not be directly measured.

- *Law III: Self Regulation*: The third law suggests that the evolution of large software systems is a self-regulating process, i.e., the system will adjust its size throughout its lifetime. This translates to observing "ripples" (small negative and positive adjustments), in the growth trend of a system. Evidence of this law according to Lehman [66, 69] is that empirical growth curves show a ripple superimposed on a steady growth trend. The ripples demonstrate the interaction between the conflicting forces of desired growth and bounded resources. In studying the law, Lehman, Perry and Ramil in [69] proposed the metric *number of modules* in a software system. The incremental growth followed by the handled models over time would give an indication of the *self regulation* nature of the software systems.

- *Law IV: Conservation of Organizational Stability*: The fourth law also known as *the invariant work rate law* stipulates that the rate of productive output of an evolving E-type system tends to stay constant throughout the product's lifetime. Lehman, Perry and Ramil in [69] suggests that the work rate can be measured by work input or work output, but leaves this work for the future. For input, the authors state that, the most obvious measure of input effort is person time. For output, the authors state that, count *handlings* (i.e., the sum over all elements of the number of different changes being implemented within a release of an *E-type* system).

- *Law V: Conservation of Familiarity*: The fifth law states that, in general, the incremental growth and long term growth rate of an *E-type* system tends to remain constant or to decline. Lehman states in [66] that, the *rate of change and growth*

of the system can be slowed down as it ages. Lehman, Perry and Ramil in [69] state that, as an E-type system evolves, all associated with it, i.e., developers, sales personnel, users, must maintain mastery of its content and behaviour to achieve satisfactory usage and evolution. Excessive growth diminishes that mastery and leads to a transient reduction in growth rate or even shrinkage. On the other hand, positive feedback from, for example, marketeers and users, leads to pressure for increasing growth rate. In studying the law, Lehman, Perry and Ramil in [69] proposed the metric *number of modules* in a software system. The trend followed by the change and growth rates of these number of modules would give an indication of familiarity is being conserved by the software systems.

- *Law VI: Continuing Growth*: The sixth law states that, the functional capability of *E-type* systems must be continually increased to maintain user satisfaction over the system lifetime. The programs usually grow over time to accommodate the pressure of change and satisfy an increasing set of requitement. Lehman et al. in [70] report that, system growth can be seen as an independent and composite monitor of system evolution which, within limits, is neither planned nor managed. Some of the factors that determine system growth include: system design, programmer style and experience, development timetable and constraints, intensity of the desire to achieve compactness of clarity [70]. Lehman et al. in [70] further report that, the majority of reported software metrics has focused on LOC (lines of code) as a measure of system size. The authors further report that, in the absence of a better measure, *module count* can be used as an initial estimator of system functionality and power.

- *Law VII: Declining Quality*: The seventh law states that the quality of *E-type* systems will appear to be declining unless they are rigorously adapted, as required, to take into account changes in the operational environment. Lehman continues to state that quality is a function of many aspects; these aspects need to be quantified to be controllable. Subject of being observed and measured in a consistent way, associated measures of quality can be defined for a system, project or organization and their value may then be tracked over *releases* or *units of time* and analyzed to determine whether levels and trends are required or desired. Some of the metrics suggested by Lehman include: fault rates, rate of fixes in previous releases and many others, to observe the fitted trend line (or other models) whether it increases, decreases or remains steady over time.

- *Law VIII: Feedback System*: The eighth law states that, an E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base. Lehman, Perry and Ramil in [69] state that, the observation that the software process is a feedback system that develops a dynamics of its own and that attempts to improve the process must take its feedback nature into account. In studying the law, the authors propose the metric *mean absolute percentage error* that is based on feedback induced dynamics.

suggest that the evolution of large software systems is a self-regulating process, i.e., the system will adjust its size throughout its lifetime. This translates to observing “ripples” (small negative and positive adjustments), in the growth trend of a system.

2.4 Results

In this section we discuss the metrics used for each of the studied laws and also discuss the corresponding observed evolutionary trends for each laws.

2.4.1 Law I: Continuing Change

An E-type *system must be continually adapted else it becomes progressively less satisfactory.*

All our plug-ins are actively used as can be seen from the number of versions and weekly downloads in Table 2.1. Previous studies have mainly used number of modules handled (additions, modifications and deletions) in each release, e.g., Barry et al. [14] employs the count of additions and changes in a portfolio, Xie, Chen and Neamtiu [113] employs cumulative number of changes. To verify the law, in our study we analyzed the cumulative growth of handled (added, deleted, and modified) modules in the ETPs. The modules used in our analysis are the handled number dependencies from Eclipse SDK interfaces, the handled number of files, and the handled source lines of code, from one version of an ETP to the next, in each of the collected ETPs. We employed all the metrics presented in Table 2.2 except the metric D_n.

Figure 2.1 and Figure 2.3 present the graphs for the cumulative metrics, Add-Deps and Del-Deps, for two of the ETPs *Eclim* and *Beyond*, respectively. Figure 2.2 and Figure 2.4 present the graphs for the cumulative metrics, Add-NOF-Deps, Mod-NOF-Deps and Del-NOF-Deps, for two of the ETPs *Eclim* and *Beyond*, respectively. In Figure 2.2 we observe that the Add-NOF-Deps and Mod-NOF-Deps metrics have a similar pattern in the trend likes. We can observe that the trends exhibited by all the metrics, for both ETPs, are of general increase. The rest of the metrics and the rest of the ETPs exhibit similar trends. We conclude that the Eclipse based evolution of the plug-ins conforms to Lehman's law of continuing change.

2.4.2 Law II: Increasing Complexity

An E-type system evolves its complexity increases unless work is done to maintain or reduce it.

In Section 2.3.3 we state that one way of measuring complexity using the insertion of mechanisms such as calls to new procedures or the insertion of objects. In the context of the ETPs, we can measure complexity by looking at the interfaces from Eclipse SDK used by the ETPs. As the law states, calls to new procedures add branch points to the system and therefore the complexity of the system is increased independent of how the complexity is measured.

In measuring the complexity of the systems as they evolve, we can look at the number of interfaces from Eclipse SDK used by the ETPs. An increase in the number of interface usage indicates increasing complexity and a decrease indicates reducing complexity. Measuring complexity this way is related to how we measure the law of continuing growth discussed later in Section 2.4.6. In measuring the law of continuing growth, among other metrics we employ the count of the number of Eclipse SDK interfaces an ETP uses as it evolves. We observed that the number of interfaces used increases over time in all the 21 ETPs we studied. Like the continuing growth law, the complexity of these ETPs increases over time. Therefore, we verify that the framework-based evolution of the ETPs conforms to *Law II: Increasing Complexity* as stated by Lehman.

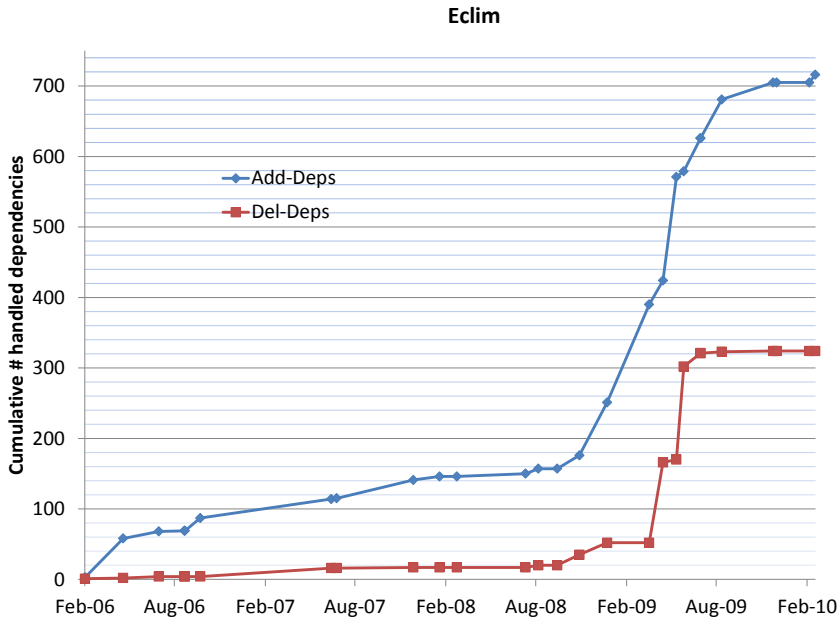


Figure 2.1: Continuing Change: Cumulative Added and Deleted Deps in ETP–Eclim

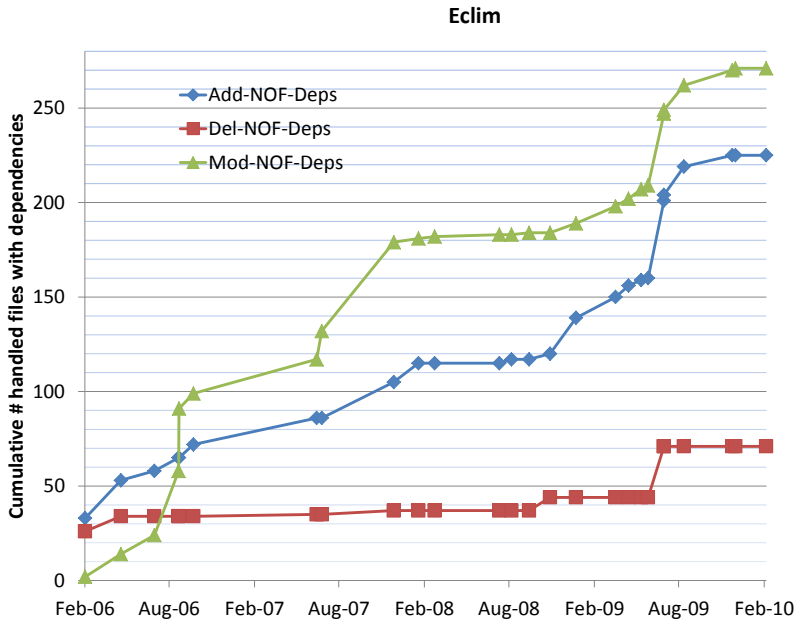


Figure 2.2: Continuing Change: Cumulative changes of files with Deps in ETP–Eclim

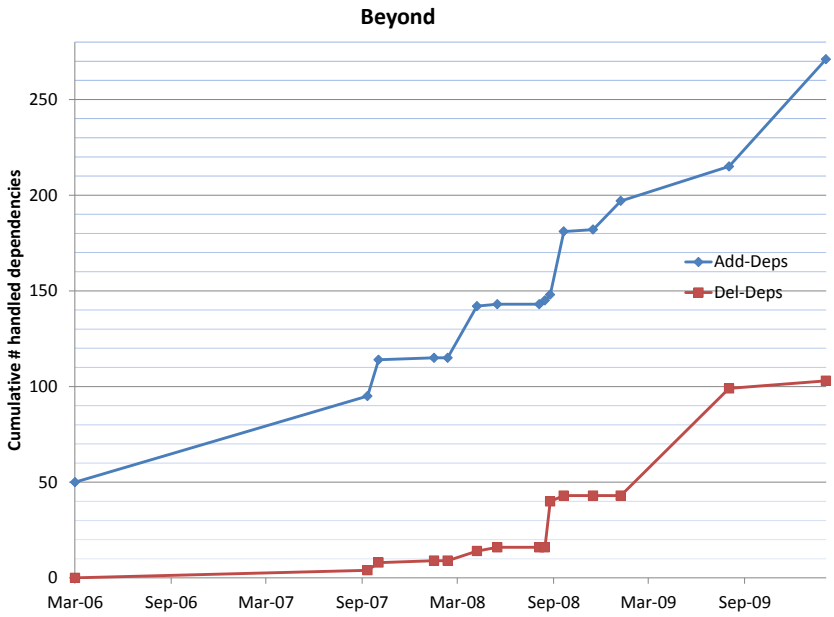


Figure 2.3: Continuing Change: Cumulative Added and Deleted Deps in ETP–Beyond

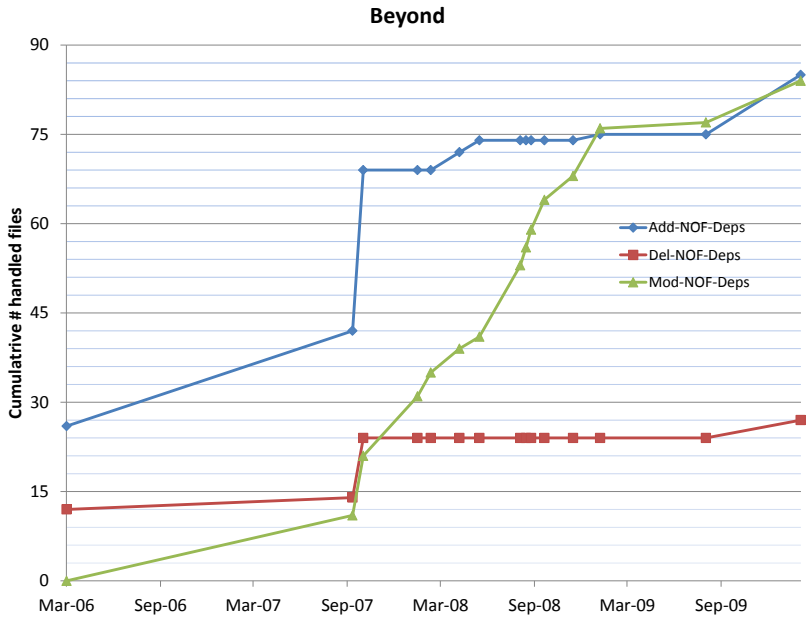


Figure 2.4: Continuing Change: Cumulative changes of files with Deps in ETP–Beyond

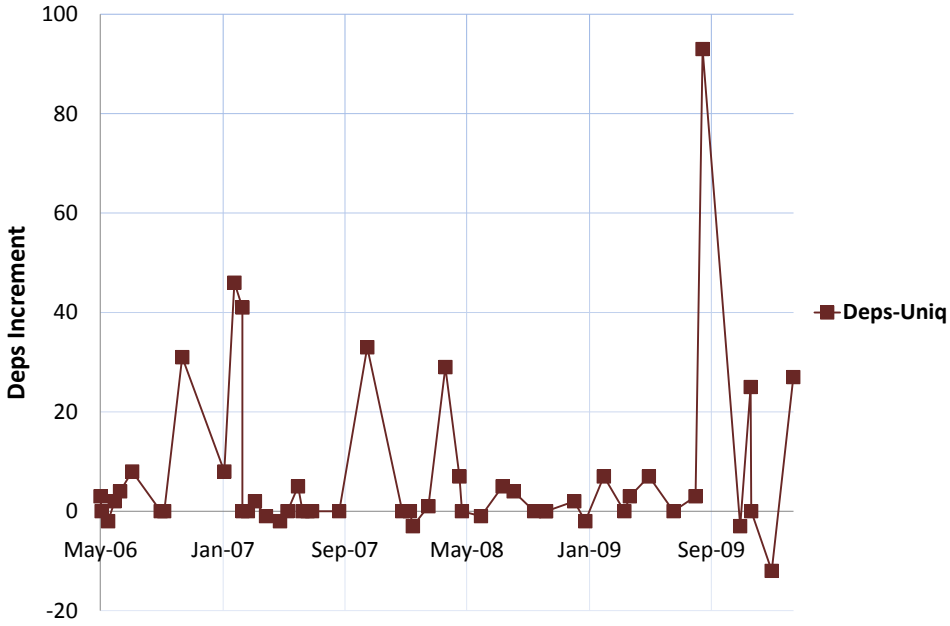


Figure 2.5: Self Regulation: Incremental Deps-Uniq growth for the ETP–PyDev

2.4.3 Law III: Self Regulation

The evolution of an E-type is a self-regulating process, i.e., the system will adjust its size throughout its lifetime. This translates to observing "ripples" (small negative and positive adjustments), in the growth trend of a system.

To verify this law, we analyzed the incremental module growth of each of the systems. The modules used in our analysis are the handled number dependencies from Eclipse SDK interfaces, the handled number of files, and the handled source lines of code, from one version of an ETP to the next, in each of the collected ETPs. We employed all the metrics presented in Table 2.2 except the metric D_n.

Figure 2.5 and Figure 2.6 present the graphs for the ETP–*PyDev* for metric Deps-Uniq and Deps-Tot, respectively. We can observe that these ripples exist with the positive adjustments occurring more frequent than the negative adjustments, a trend shared by the rest of the plug-ins. The same behavior is observed when we consider the classes i.e., NOC-Tot and NOC-Deps. We also observe the similar trends in the plots for the rest of the plug-ins that can be found in Appendix A. We therefore conclude that the Eclipse based evolution of the plug-ins conforms to Lehman's law of self-regulation.

2.4.4 Law IV: Conservation of Organizational Stability

The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

As we discussed in Section 2.3.3, project activity can be measured in terms of input and output. Lehman suggests that the most obvious measure of input is person time. Using our data we are not in a position to determine the time the developers of the ETPs invested in development. For output, the law requires count *handlings*, i.e., the sum

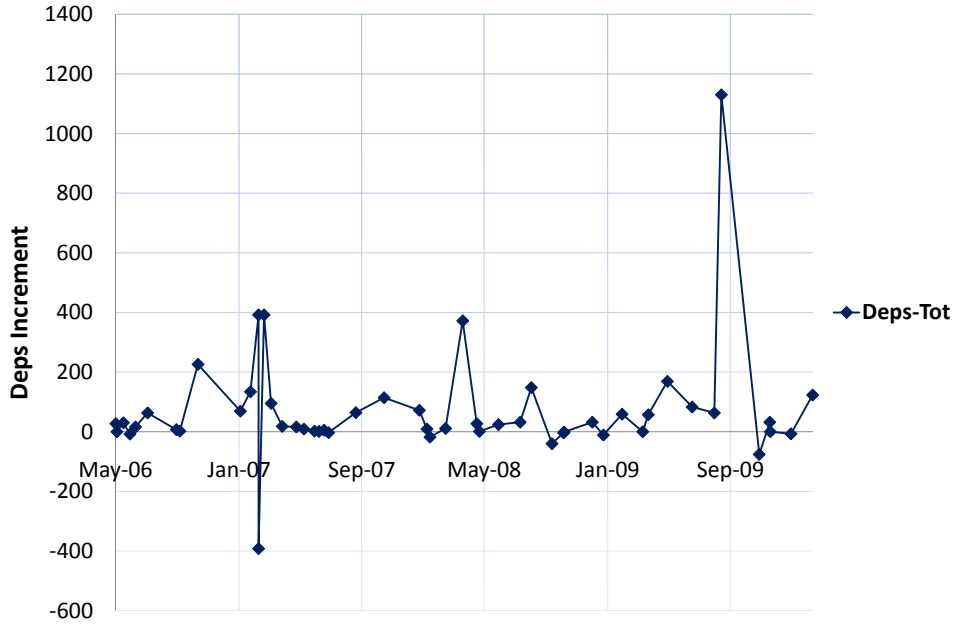


Figure 2.6: Self Regulation: Incremental Deps-Tot growth for the ETP–PyDev

over all elements of the number of different changes being implemented within a release. Measuring this law using count *handlings* is related to *Law V* in Section 2.4.5 where we consider both *change* and *growth rate* of the ETPs. Xie et al. [113] also studied this law using *change* and *growth rate*. The authors found out that both change and growth rate do not remain constant over time. In comparison to our study, in Section 2.4.5 we discovered that based on the results of *change* and *growth rates*, we observe similar trends like those in. [113]. Therefore, we also do not confirm the law of conservation of organizational stability.

Xie et al. [113] state that the invariant work rate law, in its original version, was formulated in the context of commercial software development with limited resources. The authors further state that the trends they observed make sense since the programs they studied are open-source and the number of developers tend to increase over time [77]. The authors did not have enough data to determine if this law is applicable to their examined open source programs or not. Similarity, we also do not have enough evidence to determine if this law is applicable to our examined open source programs.

2.4.5 Law V: Conservation of Familiarity

In general, the incremental growth and long term growth rate of an E-type *system tends to remain constant or to decline.*

In a study by Xie et al. [113], the authors suggests that both growth rate and change rate are neither invariant nor decreasing. The authors do not verify this claim statistically. The conclusion was based on visual inspection of the observed trends on the graphs. Barry et al. [14] considered only growth rate and found it decreasing and the result was statistically significant. In our study, we statistically test both *change* and *growth rate* and present results both for Eclipse based evolution and for general evolution of the ETPs. For

change rate we use the percentage of *handled files* (additions, deletions, modifications) in the previous version of an ETP as the *dependent* variable. The metrics used for verifying *change rate* are those of Add-NOF-Tot, Mod-NOF-Tot, Del-NOF-Tot, Add-NOF-Deps, Mod-NOF-Deps, and Del-NOF-Deps. For *growth rate* we use the percentage of *added Deps* and percentage of *added NOF* in the previous version of a plug-in as the *dependent* variables respectively. The *independent* variable in all the cases is the ETP age measured as the number of weeks since the first release considered.

A series of hypotheses were formulated to determine whether the evolution of the ETPs with respect to the ETPs' *growth rate* and *change rate* conforms to the stated law. We use both linear and quadratic models for the regression lines to determine the relationship between the dependent and the independent variables. We employ both linear and quadratic regression t-test to verify the statistical significance of the relationships. For each ETP, we first test relationship of the dependent and the independent variables with the linear model. Second, for ETPs that cannot be explained by the linear model ($p - value > 0.1$), we employ the quadratic model: if the relationship can be explained by the simplest model, the linear model, then further analysis with a nonlinear model will not produce a significant change in the result. For the choice of $p - value$ (= 0.1), we follow [67].

$$Y_t = \alpha + \beta * AGE_t + \varepsilon_t \tag{2.1}$$

$$Y_t = \alpha + \beta_1 * AGE_t + \beta_2 * AGE_t^2 + \varepsilon_t \tag{2.2}$$

Equations 1 and 2 represent the linear and quadratic model respectively. The common terms in both models: Y_t represents the particular metric (dependent variable) used to evaluate the law for a time period (weeks)t, α is the Y_t intercept or constant term, AGE_t is the variable for plug-in age that varies with time period (weeks)t and ε is the error term of the model. For the linear model, β represents the coefficient of the independent variable. For the quadratic model, AGE_t^2 is the term added to allow quadratic relationship, β_1 is the same as β and β_2 is is the coefficient of the AGE_t^2.

2.4.5.1 Hypotheses

We present a series of hypotheses for change rate and growth rate: Hypotheses *H2*, *H4* and *H5* test Eclipse based evolution while Hypotheses *H1* and *H3* test evolution.
Change Rate:

- ***(Handled NOC-Tot)*** $H1_0$*:* $\beta > 0$ *(change rate increases over time)* $H1_a$*:* $\beta \leq 0$ (change rate is invariant or declines over time)
- ***(Handled NOC-Deps)*** $H2_0$*:* $\beta > 0$ *(change rate increases over time)* $H2_a$*:* $\beta \leq 0$ (change rate is invariant or declines over time)

Growth Rate:

- ***(Deps)*** $H3_0$*:* $\beta > 0$ *(growth rate increases over time)* $H3_a$*:* $\beta \leq 0$ (growth rate is invariant or declines over time)
- ***(Handled NOC-Tot)*** $H4_0$*:* $\beta > 0$ *(growth rate increases over time)* $H4_a$*:* $\beta \leq 0$ (growth rate is invariant or declines over time)

Plug-in		**Adj R^2**	β	**Std err**	**t**	$P > \|t\|$	**Support**	$P - val$
Any	Tot	0.593	**-0.427**	0.098	-4.356	**0.001**	**YES**	0.993
	Deps	0.603	**-0.506**	0.114	-4.442	**0.001**	**YES**	0.993
Acceleo	Tot	0.324	**-0.155**	0.065	-2.408	**0.039**	**YES**	0.804
	Deps	0.390	**-0.198**	0.075	-2.717	**0.024**	**YES**	0.596
BCO	Tot	0.192	**-0.112**	0.055	-2.020	**0.066**	**YES**	0.176
	Deps	0.148	**-0.082**	0.046	-1.804	**0.096**	**YES**	0.389
Check	Tot	-0.071	-0.021	0.078	-0.272	0.790	NO	0.061
	Deps	-0.070	-0.025	0.089	-0.281	0.783	NO	0.056
Eclemma	Tot	0.022	-0.121	0.103	-1.177	0.256	NO	0.062
	Deps	0.026	-0.152	0.126	-1.207	0.245	NO	0.069
Extend	Tot	0.116	-0.128	0.082	-1.564	0.149	NO	0.998
	Deps	0.116	-0.128	0.082	-1.564	0.149	NO	0.998
PyDev	Tot	-0.018	0.004	0.009	0.371	0.712	NO	0.037
	Deps	-0.021	-0.002	0.008	-0.181	0.857	NO	0.115
Hex	Tot	0.118	**-0.090**	0.041	-2.209	**0.036**	**YES**	0.799
	Deps	0.109	**-0.113**	0.053	-2.130	**0.042**	**YES**	0.281
Beyond	Tot	0.760	**-1.053**	0.162	-6.491	**0.000**	**YES**	0.979
	Deps	0.763	**-1.053**	0.161	-6.550	**0.000**	**YES**	0.986
Eclim	Tot	0.336	**-0.192**	0.052	-3.696	**0.001**	**YES**	0.153
	Deps	0.408	**-0.266**	0.062	-4.269	**0.000**	**YES**	0.338
Verilog	Tot	-0.029	-0.075	0.111	-0.676	0.507	NO	0.007
	Deps	0.057	-0.099	0.068	-1.465	0.160	NO	0.014
Gted	Tot	0.133	**-0.577**	0.317	-1.818	**0.091**	**YES**	0.257
	Deps	0.107	-0.847	0.506	-1.672	0.117	NO	0.081
Green	Tot	0.267	**-0.111**	0.052	-2.154	**0.060**	**YES**	0.971
	Deps	0.393	**-0.138**	0.051	-2.732	**0.023**	**YES**	0.941
Clear	Tot	0.548	**-0.090**	0.022	-4.092	**0.001**	**YES**	0.283
	Deps	0.529	**-0.093**	0.024	-3.950	**0.002**	**YES**	0.269
SQL	Tot	-0.021	-0.115	0.138	-0.831	0.420	NO	0.018
	Deps	-0.028	-0.080	0.104	-0.768	0.455	NO	0.022

Table 2.3: Summary of Conservation of Familiarity (NOC ***change*** rate) results $p < 0.1$ and $\beta \leq 0$ in bold

Plug-in		**Adj R^2**	β_2	β_1	**Std err**	t_2	t_1	$P > \|t_2\|$	$P > \|t_1\|$	**Support**	$P - val$
Extend	Tot	0.332	0.002	-0.927	0.001	2.057	-2.348	**0.070**	0.043	NO	0.392
	Deps	0.332	0.002	-0.927	0.001	2.057	-2.348	**0.070**	0.043	NO	0.392
PyDev	Tot	0.032	0.0002	-0.058	0.034	1.863	-1.718	**0.069**	0.093	NO	0.018
	Deps	0.020	0.0001	-0.052	0.030	1.697	-1.730	**0.096**	0.090	NO	0.027

Table 2.4: Summary of Conservation of Familiarity (NOC ***change*** rate) quadratic model

Plug-in	**Adj R^2**	β	**Std err**	**t**	$P > \|t\|$	**Support**	$P - val$
Any	0.69	**-0.153**	0.037	-5.869	**0.000**	**YES**	0.962
Acceleo	-0.077	0.015	0.032	0.464	0.653	NO	0.637
BCO	0.129	-0.039	0.023	-1.713	0.112	NO	0.287
Check	-0.060	-0.026	0.057	-0.459	0.654	NO	0.017
Eclemma	-0.061	-0.007	0.040	-0.172	0.865	NO	0.297
Extend	-0.060	-0.009	0.014	-0.614	0.553	NO	0.114
PyDev	-0.021	-0.0003	0.004	-0.081	0.936	NO	0.000
Hex	-0.035	-0.002	0.010	-0.163	0.872	NO	0.013
Beyond	0.732	**-0.307**	0.051	-6.039	**0.000**	**YES**	0.982
Eclim	-0.006	-0.033	0.036	-0.921	0.366	NO	0.064
Verilog	-0.032	-0.008	0.012	-0.637	0.532	NO	0.023
Gted	0.110	-0.150	0.089	-1.690	0.113	NO	0.266
Green	-0.105	0.004	0.018	0.220	0.830	NO	0.867
Tour	0.027	-0.013	0.011	-1.272	0.217	NO	0.027
SQL	-0.036	-0.023	0.033	-0.691	0.501	NO	0.035

Table 2.5: Summary of Conservation of Familiarity (Deps growth rate) results $p < 0.1$ and $\beta \leq 0$ in bold

Plug-in	**Adj R^2**	β_2	β_1	**Std err**	t_2	t_1	$P > \|t_2\|$	$P > \|t_1\|$	**Support**	$P - val$
Hex	0.133	0.0003	-0.084	4.268	2.532	-2.489	**0.017**	0.019	NO	0.180

Table 2.6: Summary of Conservation of Familiarity (Deps ***growth*** rate) quadratic model

- ***(Handled NOC-Deps)*** $H5_0$*:* $\beta > 0$ *(growth rate increases over time)* $H5_a$: $\beta \leq 0$ (growth rate is invariant or declines over time)

2.4.5.2 Results

The results used to verify the stated hypotheses are presented in Tables 2.3–2.7. The metrics used to analyze this law were extracted from both binaries and the corresponding sources; for six of the plug-ins we were not able to obtain some of the sources that correspond to the binaries. We therefore excluded them and only present 15 of 21 plug-ins. Tables 2.3, 2.5 and 2.7 present the results for the linear model and Tables 2.4 and 2.6 present results for the quadratic model. Due to space constraints only entries with interesting observations are presented for the quadratic model.

In the column headers of Tables 2.3–2.7 we use the following notation: for the common terms, $AdjR^2$ represents the adjusted R-square; used to explain the variation explained by the dependent variable, Std err represents the standard error of the slope. For the terms in linear model tables, t is the t-score test statistic and $P > |t|$ represents the $p - value$ that corresponds to t with certain degrees of freedom (number of versions in Table 2.1). For terms in the quadratic model tables, t_1 and t_2 are the t-score test statistics for β_1 and β_2 respectively and $P > |t_2|$ and $P > |t_1|$ are the resulting $p - values$ respectively.

To ensure that the results of the significance tests are meaningful, we carried out the Kolmogorov-Smirnov normality test on the error component, ε, in the models (last column). At a given significance level, $(100 - \alpha)\%$, the test compares the distribution followed by the error component and the normal distribution. The null hypothesis is that the two distributions are different and the alternative hypothesis is that they are the

Plug-in		**Adj R^2**	β	**Std err**	**t**	$P > \|t\|$	**Support**	$P - val$
Any	Tot	0.466	**-0.165**	0.045	-3.638	**0.003**	**YES**	0.920
	Deps	0.390	**-0.157**	0.050	-3.154	**0.008**	**YES**	0.635
Acceleo	Tot	-0.097	-0.006	0.053	-0.177	0.863	NO	0.794
	Deps	-0.074	-0.013	0.065	-0.493	0.633	NO	0.910
BCO	Tot	0.238	**-0.036**	0.016	-2.251	**0.044**	**YES**	0.282
	Deps	0.063	-0.021	0.009	-1.367	0.197	NO	0.257
Check	Tot	-0.066	-0.012	0.030	-0.372	0.716	NO	0.056
	Deps	-0.065	-0.013	0.034	-0.380	0.710	NO	0.000
Eclemma	Tot	-0.031	-0.013	0.018	-0.695	0.497	NO	0.075
	Deps	-0.021	-0.017	0.021	-0.811	0.429	NO	0.092
Extend	Tot	0.295	**-0.070**	0.029	-2.367	**0.039**	**YES**	0.995
	Deps	0.295	**-0.070**	0.029	-2.367	**0.039**	**YES**	0.995
PyDev	Tot	-0.015	-0.002	0.004	-0.525	0.602	NO	0.000
	Deps	0.017	-0.003	0.006	-0.424	0.673	NO	0.000
Hex	Tot	0.126	**-0.022**	0.010	-2.273	**0.031**	**YES**	0.153
	Deps	0.124	**-0.035**	0.016	-2.263	**0.032**	**YES**	0.111
Beyond	Tot	0.750	**-0.405**	0.064	-6.320	**0.000**	**YES**	0.999
	Deps	0.751	**-0.106**	0.064	-6.345	**0.000**	**YES**	1.000
Eclim	Tot	0.246	**-0.035**	0.012	-3.029	**0.006**	**YES**	0.768
	Deps	0.270	**-0.045**	0.014	-3.202	**0.004**	**YES**	0.901
Verilog	Tot	-0.054	0.010	0.075	0.140	0.890	NO	0.011
	Deps	0.016	-0.018	0.016	-1.140	0.268	NO	0.008
Gted	Tot	0.054	-0.277	0.203	-1.364	0.194	NO	0.076
	Deps	0.046	-0.512	0.390	-1.312	0.211	NO	0.078
Green	Tot	0.047	0.025	0.020	1.221	0.253	NO	0.843
	Deps	-0.107	0.003	0.019	0.182	0.860	NO	0.753
Clear	Tot	0.331	**-0.017**	0.006	-2.728	**0.018**	**YES**	0.560
	Deps	0.345	**-0.017**	0.006	-2.802	**0.016**	**YES**	0.487
SQL	Tot	-0.036	-0.029	0.042	-0.696	0.498	NO	0.028
	Deps	-0.038	-0.030	0.044	-0.675	0.511	NO	0.012

Table 2.7: Summary of Conservation of Familiarity (NOC ***growth*** rate) results $p < 0.1$ and $\beta \leq 0$ in bold

same. If the result of the test is less than α, then their is a $(100-\alpha)\%$ chance that the two distributions are different. From our results, considering a threshold, $\alpha = 0.01$, only a total of six entries in all the tables fail the normality test.

Tables 2.3 and 2.4 present the *change rate–general evolution* (Tot) and *change rate–Eclipse based evolution* (Deps) for the plug-ins. For hypothesis $H1$ we have 9 of the 15 plug-ins supporting the law and for $H2$ we have 8 of the 15 supporting the law. However, out of the 9 and 8 plug-ins in Tot and Deps respectively, only 2 plug-ins, *Any* and *Beyond*, express a strong co-efficient of variation ($AdjR^2 > 0.5$). In Table 2.4, plug-ins *PyDev* and *Extend* show a statistically significant result ($p-value < 0.1$) but their result in Table 2.3 are not statistically significant. This means that the *change rate* of these two plug-ins is more accurately modeled by the quadratic model. A more interesting observation is that as opposed to exhibiting an invariant or declining *change rate* as the law stipulates, an increasing trend is observed ($\beta_2 > 0$). Further investigation of these 2 plug-ins, show that the growth of both the NOC-Tot and NOC-Deps follows a super-linear trend. This result corresponds to Koch's observation that projects with a super-linear growth are more active in number of revisions than those with linear and more active than those with sub-linear [59]. A chaotic change rate was observed for rest of the plug-ins.

Tables 2.5–2.7 present the growth rate–general evolution (Tot) and growth rate–Eclipse based evolution (Deps) for the plug-ins. For hypothesis $H3$ we have 2 of the 15 plug-ins supporting the law which also have strong co-efficient of variation ($\approx AdjR^2 \geq 0.7$). Plug-ins BCO and Gted are nearly statistically significant at the chosen significant level 0.1. For hypothesis $H4$ we have 7 of the 15 plug-ins in support and hypothesis $H5$ we have 6 of the 15 in support; only *Beyond* has got a strong coefficient of variation. The large significant difference in the results of *Hex* in Table 2.5 and 2.6 imply that the metric can be better modeled by the quadratic model. The results in Table 2.6 show an increasing trend of the growth rate of Deps ($\beta_2 > 0$). Further investigation on this plug-in revealed that the metric Deps exhibits a superlinear growth trend. This corresponds to Godfrey and Tu's findings that Linux exhibits a super-linear growth trend and therefore it exhibits a steady growth rate trend as opposed to constant or declining trend that the law stipulates [47].

From the analysis of the hypotheses, we cannot reject or accept any of them since some plug-ins are in support of the law and others are not. The possible reason for the observed behavior is that as opposed to the commercial systems on which the Lehman investigated the law, our plug-ins are open-source. For this reason, the law of *conservation of familiarity* can be confirmed for some of the plug-ins and not confirmed for the others.

2.4.6 Law VI: Continuing Growth

The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.

Despite the fact that the law talks about functional capability, most of the research that has been carried out so far have considered metrics that measure size of the program: e.g., number of sub-projects, number of features, number plug-ins, NOC, NOF [75], number of modules [14,113], LOC [47,75,113], number of definitions [113] were considered. One of the main reasons for this choice of metrics is that it is not easy to measure the functional capability. In addition to the metrics that have been used to measure size and growth over time for the programs studied so far, we also study Deps-Tot, Deps-Uniq, SLOC-Tot, SLOC-Deps, NOC-Tot and NOC-Deps (cf. Section 2.3.1). After analysis of the results of these metrics, we have observed that there is an increasing trend for all the

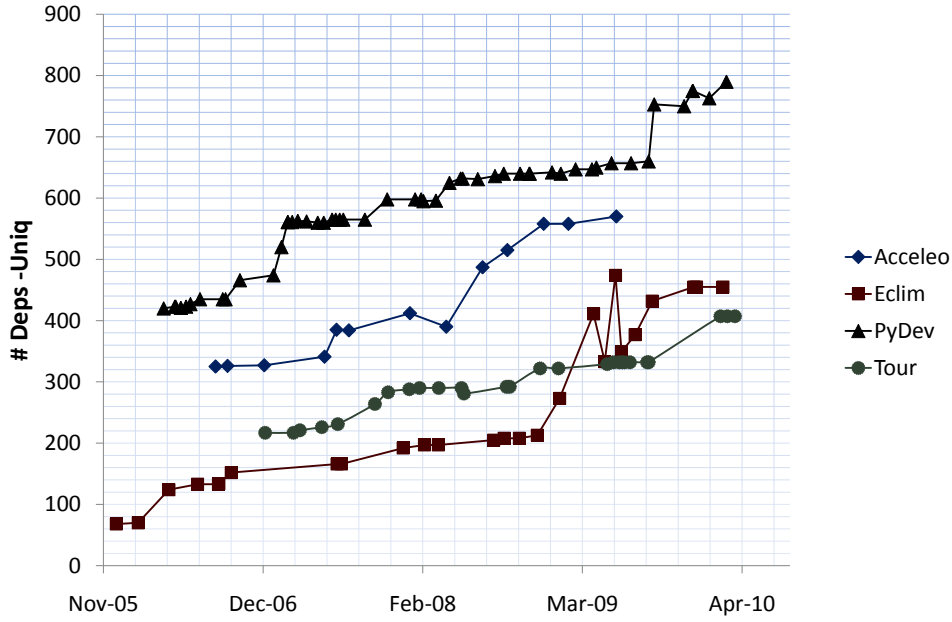


Figure 2.7: Continuous Growth: Growth trends for the metric Deps-Uniq for selected ETPs

chosen metrics on all the studied plug-ins.

Table 2.8 presents the results of the coefficient of linear regression relationship (R^2) between the considered metrics and the time in weeks. We can observe from the high values of R^2 as an indication of increasing number of modules in the ETPs with time. The ETP SQL shows a strange results for R^2 for the metrics Deps-Tot and Deps-Uniq. We could not figure out the cause of this observed trend. Figure 2.7 and Figure 2.8 present the plots the relationship between the considered metrics and the time in weeks for selected ETPs. The rest of the metrics show general increase in the trends for all the plug-ins and be found in Appendix A. From the analysis, we can conclude that the *Continuing Growth* law is confirmed for both general evolution and Eclipse based evolution of the ETPs.

2.4.7 Law VII: Declining Quality

The quality of E-type *systems will appear to be declining unless they are rigorously adapted, as required, to take into account changes in the operational environment.*

Subject of being observed and measured in a consistent way, associated measures of quality can be defined for a system, project or organization and their value may then be tracked over *releases* or *units of time* and analyzed to determine whether levels and trends are required or desired. Some of the metrics Lehman suggests include fault rates, rate of fixes in previous releases and many others, to observe the fitted trend line (or other models) whether it increases, decreases or remains steady over time.

Like the law of conservation of familiarity in Section 2.4.4, we use empirical results from the plug-ins to analyze whether the plug-ins' Eclipse based evolution conforms to the law. We employ the same methodology to evaluate the conformity of this law. The metrics used for the independent variable are obtained from Martin's [73] normalized

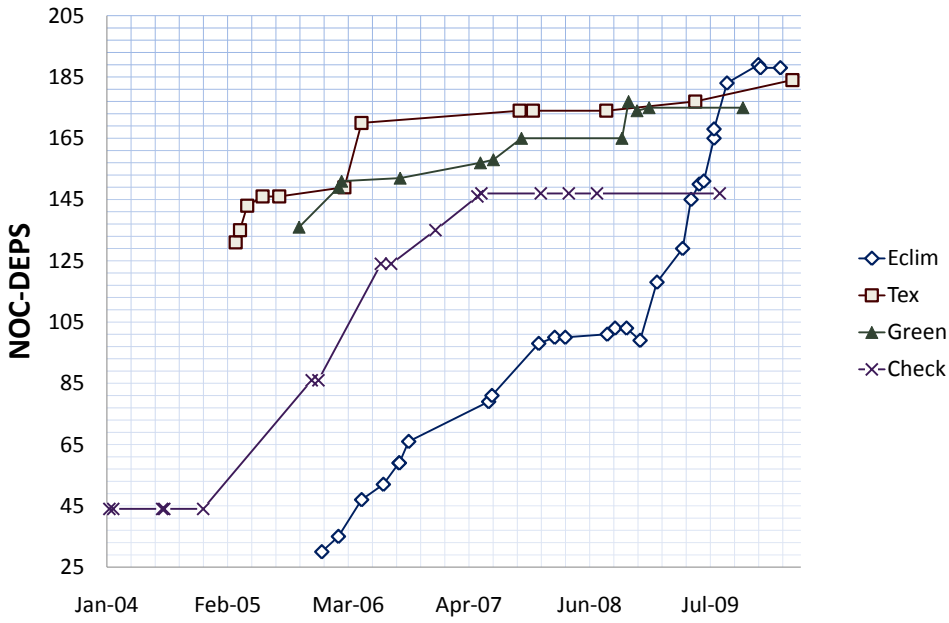


Figure 2.8: Continuous Growth: Growth trends for the metric NOF-Deps for selected ETPs

	Coefficient of regression					
ETP	**Deps-Tot**	**Deps-Uniq**	**NOF-Tot**	**NOF-Deps**	**SLOC-Tot**	**SLOC-Deps**
Any	0.952	0.954	0.905	0.901	0.964	0.964
Acceleo	0.909	0.919	0.872	0.903	0.801	0.826
BCO	0.869	0.834	0.896	0.906	0.915	0.908
Check	0.885	0.872	0.888	0.882	0.887	0.882
Ecelemma	0.743	0.783	0.907	0.887	0.806	0.799
Extend	0.891	0.767	0.851	0.851	0.895	0.895
Quantum	0.705	0.775	0.842	0.855	0.901	0.924
Tex	0.833	0.901	0.789	0.834	0.824	0.823
PyDev	0.940	0.916	0.911	0.912	0.923	0.898
Hex	0.698	0.705	0.814	0.687	0.867	0.811
Beyond	0.748	0.855	0.878	0.893	0.901	0.911
Tour	0.965	0.923	0.918	0.953	0.921	0.956
Eclim	0.882	0.841	0.929	0.923	0.963	0.887
Verlog	0.931	0.919	0.836	0.943	0.936	0.955
PHP	0.764	0.796	0.698	0.741	0.768	0.800
Metrics	0.951	0.829	0.956	0.865	0.961	0.964
Gted	0.535	0.480	0.592	0.584	0.667	0.659
Green	0.654	0.272	0.703	0.871	0.900	0.892
Clear	0.850	0.923	0.608	0.689	0.895	0.911
JavaCC	0.979	0.899	0.989	0.979	0.779	0.939
SQL	0.258	-0.632	0.911	0.944	0.940	0.970

Table 2.8: Continuous Growth: Values for the coefficient of linear regression R^2 of relationship between the metrics and time (in weeks).

distance from the main sequence, denoted by D_n. D_n is an indicator of the package's balance between *abstractness* and instability. *Abstractness* (A) of a category: (# abstract classes in a category ÷ total # of classes in category). This metric range is [0,1], $A = 0$ indicates that all classes in the category are abstract and $A = 1$ indicates that all the classes are concrete. *Instability* (I) of a category is defined by the following formula: (Ce ÷ (Ce + Ca)), where Ce (afferent coupling) is the number of classes outside this category that depend upon classes within this category. Ce (efferent Couplings) is the number of classes inside this category that depend upon classes outside this categories. The metric I has the range [0,1]. $I = 0$ indicates a maximally stable category and $I = 1$ indicates a maximally instable category. The normalized distance from the main sequence D_n is defined by the formula $|(A + I - 1)$ and has a range $[0, 1]$. $D_n = 0$. Martin reports that one should expect a design with high quality to have the *mean* and *variance* of the $D'_n s$ to be close to zero.

To test the conformance of this law we follow [98] and investigate the growth trends of averages ($\bar{D_n}$) and standard deviations ($\sigma(D_n)$) of the different versions of the plug-ins over time. The method used in determining our average D_n is borrowed from [98] in the study of formalizing D_n-based architecture assessment of Java Open Source software. To ensure the validity of our results, we follow [98] and also select plug-ins having at least thirty packages (not including third party packages). Only 8 of the 21 plug-ins satisfied this requirement.

2.4.7.1 Hypotheses

We state the hypotheses that will be used to test the evolution.

- $H6_0$: $\beta \leq 0$ *($\bar{D_n}$'s of the different packages in the different plug-in versions will not increase over time)* $H6_a$: $\beta > 0$ *($\bar{D_n}$'s of the different packages in the different plug-in versions will increase over time)*
- $H7_0$: $\beta \leq 0$ *($\sigma(D_n)$'s of the different packages in the different plug-in versions will not increase over time)* $H7_a$: $\beta > 0$ *($\sigma(D_n)$'s of the different packages in the different plug-in versions will increase over time)*

2.4.7.2 Results

Tables 2.9 and 2.10 show the results for the linear and quadratic models analyzed. For the linear and quadratic models, we use the models in Equation 6.1 and 6.2, respectively, defined in Section 2.4.5. The dependent variables as $\bar{D_n}$ and $\sigma(D_n)$ and the independent is number of weeks.

From the Table 2.9 considering the *MEAN*, we have four of the eight plug-ins in support of the law. In Table 2.10 *Tex* is in support of the law. This means that the $\bar{D_n}$'s are better modeled by the quadratic model. This increases the number of plug-ins in support of the law to 5 out of 8. Much as the coefficients of AGE, β's, appear to be small from the tables, the values are sufficient since the values of the dependent variable are large (up to 300-500 weeks) and all the Y_t intercepts are > 0. In addition, the result of the model(s) is in the range, $0 \leq Y_t \leq 1$.

For *SQL* we have an exceptional case where the $\bar{D_n}$'s are significantly decreasing and the $\sigma(D_n)$'s are significantly increasing. On inspection of the raw data we observed that in the last seven versions of the plug-in introduced many packages that had $D_n = 0$, this

Plug-in		Adj R^2	β	Std err	t	$P > \|t\|$	Support	$P - val$
Acceleo	MEAN	0.275	**3.9E-05**	1.7E-05	2.355	**0.038**	**YES**	0.331
	STDV	-0.032	2.7E-05	3.4E-05	0.789	0.447	NO	0.159
Quantum	MEAN	0.902	-3.7E-05	3.6E-06	-10.126	0.000	NO	0.641
	STDV	0.822	-4.9E-05	6.9E-06	-7.199	0.000	NO	0.244
Tex	MEAN	-0.024	-1.2E-05	1.4E-05	-0.860	0.410	NO	0.602
	STDV	0.905	**3.6E-05**	3.5E-06	10.313	**0.000**	**YES**	0.717
PyDev	MEAN	0.307	**7.8E-05**	1.6E-05	4.813	**0.000**	**YES**	0.168
	STDV	0.013	-8.0E-06	6.2E-06	-1.292	0.203	NO	0.071
Eclim	MEAN	0.124	**7.8E-05**	3.6E-05	2.199	**0.037**	**YES**	0.020
	STDV	0.786	**1.5E-04**	1.5E-05	10.022	**0.000**	**YES**	0.997
PhP	MEAN	0.027	-1.3E-05	1.1E-05	-1.157	0.272	NO	0.250
	STDV	0.031	-8.6E-06	7.3E-06	-1.177	0.264	NO	0.541
Metrics	MEAN	0.774	**1.9E-04**	2.6E-05	7.368	**0.000**	**YES**	0.781
	STDV	0.018	**2.5E-05**	2.2E-05	1.128	0.278	NO	0.648
SQL	MEAN	0.516	-7.4E-05	1.9E-05	-3.994	0.000	NO	0.547
	STDV	0.430	**5.8E-05**	1.7E-05	3.397	**0.005**	**YES**	0.939

Table 2.9: Growth trends of the $\bar{D_n}$'s and $\sigma(D_n)$'s of the plug-ins that passed the selection criteria

Plug-in		Adj R^2	β_2	β_1	t_2	t_1	$P > \|t_2\|$	$P > \|t_1\|$	Support	$P - val$
Acceleo	MEAN	0.278	2.5E-07	-2.1E-05	1.583	-1.290	0.145	0.226	NO	0.886
	STDV	0.092	7.2E-07	-1.5E-04	1.583	-1.290	0.145	0.226	NO	0.866
Tex	MEAN	0.893	**3.4E-07**	-1.5E-04	9.329	-9.207	**0.000**	0.000	**YES**	0.990
	STDV	0.895	-1.8E-09	**3.7E-05**	-0.057	2.789	0.956	**0.021**	**YES**	0.985

Table 2.10: Growth trends of the $\bar{D_n}$'s and $\sigma(D_n)$'s of plug-ins for the quadratic model

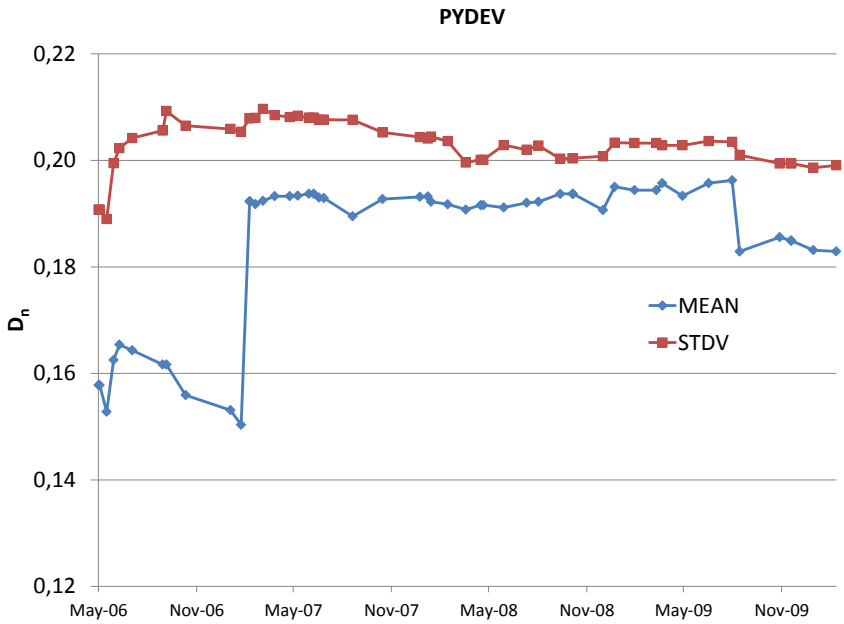


Figure 2.9: Growth trend of the mean and standard deviation of $\bar{D_n}$ for PyDev plug-in over time.

in turn lowered the averages. Therefore the increase in $\sigma(D_n)$'s tells us that the quality the old packages has not changed. *PyDev* exhibits a statistically significant increase in the $\bar{D_n}$'s and the $\sigma(D_n)$'s are close to significantly decreasing. The reason for this behavior can be obtained by observing the growth trend of D_n's, Figure 2.9. The trend is staged; where it is more or less stable in the early stages, then a sharp increase, then it becomes stable for a long period, after it goes down and finally it becomes stable again. *Quantum* exhibits a increasing quality as seen from both growth trends of $\bar{D_n}$'s and $\sigma(D_n)$'s decreasing over time. Since the law states that unless the systems being evolved are rigorously adapted, the quality is supposed to decline. It is possible that this might be the case with *Quantum* plug-in but we leave further investigation for the future study when we increase the number of plug-ins being studied.

To ensure the validity of our results, we follow [98] and also select plug-ins having at least thirty packages (not including third party packages). Only 8 of the 21 plug-ins satisfied this requirement.

In investigating the law of declining quality we employed the methodology described in [98]. In [98] the requirement was that only software systems having at least thirty packages (not including third party packages) were considered in the analysis. In our study, only 8 of the 21 plug-ins satisfied this requirement. Since we only studied 8 plug-ins our conclusions will be threatened by external validity, therefore our findings for this law are inconclusive.

2.4.8 Law VIII: Feedback System

An E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Lehman, Perry and Ramil in [68] state that the term global process as used in their study encompasses the activities, at many levels, of technical personnel, management, marketing and support personnel, users and others involved. In our study, from the data we collected, we do not have access to this kind of information. We therefore can not comment on this law.

2.5 Threats to Validity

Like any other empirical analysis, our study may have been affected by validity threats. We categorized the possible threats into construct, internal and external validity.

2.5.1 Construct Validity

This validity relies on the assumption that the independent and the dependent variables accurately model the intended characteristic we are studying. In our situation, for example, whether the time series of the chosen metrics accurately model the characteristics claimed in the laws we have studied. To mitigate this threat, we used a number of metrics for each of the studied laws. Despite the fact that a number of studies, including this one, carried out on the law of continuing growth use metrics that relate to size, the law talks about functional content. A study that validates whether a software systems' size and functional content exhibit same evolution patterns needs to be carried out.

2.5.2 Internal Validity

The key question of internal validity in our study is whether the observed changes in the dependent variable can be attributed to the changes in the independent variable and not other possible causes. This threat seems to be comparably lesser in our study since we tried to mitigate it during the plug-in inclusion/exclusion in our study by the selection criteria we set in Section 2.2. For the two laws conservation of familiarity and declining quality, we further strengthened the significance test results by making sure that the error, ε, component in the models follows a normal distribution by carrying out the Kolmogorov-Smirnov normality tests.

2.5.3 External Validity

This threat refers to the degree to which our conclusions from our findings can be generalized. In our study we observed two threats that relate to external validity. First, as we already pointed out in Section 2.4.7 our findings were inconclusive since the sample size reduced from 21 to only 8 plug-ins. Second, we are studying the Eclipse based evolution of Eclipse and its plug-ins but we only consider open-source plug-ins.

2.6 Related Work

Most of the studies that are related to our work have already been implicitly introduced in Section 2.4 while discussing the different laws we studied. Studies of Xie et al. [113] and Barry et al. [14] are yet to be discussed and are closely related to our study.

Xie et al. [113] investigate all Lehman's evolution laws by conducting an empirical study on the evolution of seven long-lived (5–15 years) Open Source programs written in the C language. We employ similar methods for verifying the laws of continuing change, self regulation, conservation of familiarity, and declining quality. In both studies, the laws of continuing change and self regulation were validated and the laws of conservation of familiarity and declining quality were not validated. In the study of conservation of familiarity, the authors base their conclusion on visual inspection of the growth trends of the metrics they employed. We based our conclusion on quantitative analysis using statistics. For the law of declining quality, their analysis was based on observing the sign of the slope and the correlation coefficient between the independent and the dependent variable. In our study, we employ hypothesis tests on the models we stated which we believe is a better way of studying growth of a time series variables since in addition to sign of slope and correlation coefficient, it incorporates other terms that strengthens the reliability of the results.

Barry et al. [14] studied Lehman's laws on 23 application systems for a large retail company that had evolved over 20 years. In the data analysis, they stated hypotheses for each law and employed a time series regression to test each of the hypotheses. Two of the laws; conservation of familiarity and declining quality, both studies (our study and their study) employed similar methods to analyze the data. The first difference between the studies is that they found support of these two laws which we did not. A summary of their findings was reported but it is not clear if all or the majority of the 23 applications studied conform to the law. If this is the case, which we could not establish with our applications, one reason for the difference could be their applications were drawn from the same domain (same development conventions are applied to all applications since they are developed from the same company). Second, the authors analyze growth rate in

verifying the conservation of familiarity law, whereas we analyze both growth and change rate. Third, the two studies employ different metrics in analyzing the declining quality law.

Wermelinger and Yu [110] present their results on the evolution of Eclipse core plug-ins (ECPs). The authors find that the development of these ECPs follows a systematic process: most architectural changes take place in milestones, and maintenance releases only make exceptional changes to component dependencies.

2.7 Conclusions and Future Work

In this chapter, we took the first step to study the fine grained evolution of software systems built from component frameworks, by investigating their evolutionary trends using Lehman's laws of software evolution. Our study was based on 21 third-party plug-ins of the Eclipse component framework, from Eclipse 3.0 to 3.5. We investigated the empirical evidence of 7 of the 8 Lehmans' evolution laws. Our findings confirm the laws of continuing change, increasing complexity, self regulation, and continuing growth when metrics related to dependencies between the plug-ins and the Eclipse architecture are considered. We could not validate the conservation of familiarity and conservation of organizational stability law, and the results for the declining quality law were inconclusive. We could not comment on the law of feedback system since we did not have the data to measure the law. Our overall observation is that the trends observed for constrained evolution are similar to those presented by earlier researches on the general evolution of software systems.

The results obtained so far and presented above are encouraging. One can extend our study by gathering more data to test the law of declining quality and the law of feedback system.

Chapter 3

Big Data Collection

In this Chapter, we explain how we collected the data that we used to carry out our studies in Chapter 4, 5, and 6.

3.1 Data Collection

For our analysis, we collected ETPs from SourceForge. Since we wanted to have a long enough history of the ETPs, we chose SourceForge as opposed to recently popular repositories like GoogleCode[1] and Github[2]. On the SourceForge search page, we typed the text "*Eclipse AND plugin*" and the search returned 1,350 hits (collected February 16, 2011).

During the ETP collection, two major steps were considered to extract and classify the *useful ETPs* from the 1,350 hits. These steps are downloading and initial classification, and removal of incomplete ETPs.

3.1.1 Downloading and Initial Classification

We obtained source code manually for the plug-ins considered. Since we wanted to have a long history of ETPs supported in the different ECPs and a substantial amount of data to draw sound statistical conclusions, we decided to collect ETPs that were released on SourceForge from January 1, 2003 to December 31, 2010. The ETPs for which none of the versions had source code but only had binaries were omitted. During the collection process, each ETP was categorized according to the year of its first release on SourceForge.

3.1.2 Removal of incomplete ETPs

To reduce threats to validity as much as possible, we removed two groups of ETPs. First, we excluded *incomplete* ETPs, i.e., ETPs having at least one version containing only

[1] http://code.google.com
[2] https://github.com/

binaries but no source files. Versions without source code may interrupt discovery of evolution trends of the ETP. Second, we eliminated ETPs that do not import packages from ECPs. These are software programs returned as noise from SourceForge search engine. As mentioned earlier in Chapter 1, ECPs and EEPs share a common prefix `org.eclipse`. To ensure that only the `import` statements from ECPs in the ETPs source code are considered, a list of all possible import statements from ECPs was compiled. We used the following plug-ins as ECPs: `ant`, `compare`, `core`, `debug`, `equinox`, `help`, `jdt`, `jface`, `jsch`, `ltk`, `osgi`, `pde`, `search`, `swt`, `team`, `text`, `ui`, `update`.

We define Dependencies-on-ECPs as import statements *from ECPs starting with the prefix `org.eclipse` in the source code of an ETP.*

3.1.3 Results of Data collection

Table 3.1 shows the summary of the data collected from SourceForge. The table is divided into four groups:

1. *Overall*: Corresponding to the total ETPs that were collected from SourceForge.
2. *Incomplete*: Corresponding to the incomplete ETPs.
3. *No Deps*: Corresponding to the ETPs that did not have dependencies on ECPs.
4. *Clean*: Corresponding to remaining ETPs that we consider in our analysis in Chapters 4–6.

The group *Overall* in Table 3.1, the cell entry (2004,2004), typeset in italics, contains a pair (83 192) indicating that there are a total of 83 ETPs that were first released in the year 2004 on SourceForge altogether having a total of 192 versions. The pair (20 57) in the cell (2004,2005) means that there were 20 ETPs of the 83 that had new versions released in the year 2005 with a total of 57 versions in that year. We observe that the trend on the evolution in the *total* number of ETPs is non-monotone, for example, (2004,2006)=16, (2004,2007)=10, (2004,2008)=11. This indicates that, while an ETP may have version(s) in a given year, it does not release any version(s) in the subsequent year(s) and resumes releasing later.

Similarly, the group *Clean* in Table 3.1, (77 171) in the cell (2004,2004) indicates that out of 83 ETPs that were first released in the year 2004, 77 are considered clean. These 77 ETPs amount to 171 versions. Cell entry (2004,2005) indicates that 19 of 77 clean ETPs had new versions released in the year 2005 with a total of 54 versions in that year. We see that the total number of clean ETPs is 512 (sum of ETPs in the diagonal) having a total of 1,873 versions (sum of the versions in the diagonal), corresponding to the sum of the lower pair values in diagonal cells. A similar explanation can be used to read the entries of the remaining groups, i.e., *Incomplete* and *No Deps*.

		2003		2004		2005		2006		2007		2008		2009		2010	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
Overall	2003	*91*	*240*	41	99	23	44	18	30	14	28	6	8	2	8	4	8
	2004			*83*	*192*	20	57	16	40	10	26	11	25	4	8	5	11
	2005					*88*	*192*	30	79	19	47	18	31	10	32	8	24
	2006							*107*	*251*	26	72	8	35	3	13	11	28
	2007									*73*	*177*	22	46	10	19	10	18
	2008											*74*	*156*	18	36	4	8
	2009													*47*	*72*	9	15
	2010															*28*	*43*
	Total	91	40	124	291	131	293	171	400	142	350	139	301	94	188	79	155
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
Incomplete	2003	*7*	*38*	3	7	1	2	1	1	1	1	0	0	0	0	1	2
	2004			*2*	*12*	0	0	2	6	1	1	1	1	1	2	1	2
	2005					*8*	*19*	4	13	3	5	5	7	1	9	0	0
	2006							*7*	*25*	2	2	1	1	0	0	0	0
	2007									*15*	*40*	3	5	2	2	4	6
	2008											*8*	*32*	4	9	0	0
	2009													*4*	*9*	1	2
	2010															*1*	*3*
	Total	7	38	5	19	9	21	14	45	22	49	18	46	12	31	8	15
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
No Deps	2003	*4*	*7*	1	5	1	3	1	2	1	2	0	0	0	0	0	0
	2004			*4*	*9*	1	3	0	0	0	0	0	0	0	0	0	0
	2005					*7*	*19*	3	6	2	2	1	3	0	0	0	0
	2006							*3*	*3*	0	0	0	0	0	0	0	0
	2007									*1*	*3*	1	4	0	0	0	0
	2008											*4*	*5*	0	0	0	0
	2009													*3*	*4*	0	0
	2010															*1*	*1*
	Total	4	7	5	14	9	25	7	11	4	7	6	12	3	4	1	1
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
Clean	2003	*80*	*195*	37	87	21	39	16	27	12	25	6	8	2	8	3	6
	2004			*77*	*171*	19	54	14	34	9	25	10	24	3	6	4	9
	2005					*73*	*154*	23	60	14	40	12	21	9	23	8	24
	2006							*97*	*223*	24	70	7	34	3	13	11	28
	2007									*57*	*134*	18	37	8	17	6	12
	2008											*62*	*119*	14	27	4	8
	2009													*40*	*59*	8	13
	2010															*26*	*39*
	Total	80	195	114	258	113	247	150	344	116	294	115	243	79	153	70	139

Table 3.1: Total numbers of ETPs collected, and the numbers of clean ETPs and their versions. Numbers of the ETPs that were first released in a certain year are typeset in italics. E–ETPs and V–Versions

Chapter 4

Eclipse API Usage: The Good and The Bad

Today, when constructing a new software system, many developers build their systems on top of frameworks. Eclipse is such a framework that has been in existence for over a decade and has so far released 11 major releases. Like many other evolving software systems, the Eclipse platform has both stable and supported APIs ("good" interfaces) and unstable, discouraged and unsupported non-APIs ("bad" interfaces). However, despite being discouraged by Eclipse, in our experience, the usage of "bad" interfaces is relatively common in practice. In this chapter, we study to what extent developers depend on "bad" interfaces. We also study whether developers continue to use "bad" interfaces, and what are the differences between the third-party plug-ins that use non-APIs and those that do not. Furthermore, we also study the commonly used "bad" interfaces.

To answer these questions, we have conducted an empirical investigation based on a total of 512 *Eclipse third-party plug-ins, altogether having a total of* 1,873 *versions (we describe data collection in Chapter 3). We discovered that* 44% *of the* 512 *analyzed Eclipse third-party plug-ins depends on at least one "bad" interfaces and that developers continue to use "bad" interfaces. The empirical study also shows that plug-ins that use or extend at least one "bad" interfaces are comparatively larger and use more functionality from Eclipse than those that use only "good" interfaces (reasons why developers use bad interfaces is discussed in Chapter 7). Furthermore, the findings show that the third-party plug-ins use a diverse set of "bad" interfaces.*

4.1 Introduction

Today, many software developers are building their systems on top of frameworks. This approach has many advantages, such as reuse of the functionality provided [80] and increasing productivity [60]. However, in spite of these benefits, there are potential challenges that come along, for both the framework developer and the developer who uses the functionality from the framework [20]. On the framework developer side, continuous evolution takes place as a result of refactoring and introduction of new functionality in the framework components [103]. A potential challenge as a result of performing these

activities is that the framework developers have to refrain from changing the existing application programming interfaces (APIs) because such change may cause applications that depend on these APIs to fail [114]. In practice, for successfully and widely adopted frameworks, it may not be possible to achieve this fully. On the framework user side, if developers are to take advantage of the better quality and new functionality introduced as a result of the evolution of the framework, then the evolution of these software systems may be constrained by two independent, and potentially conflicting, processes [22, 114]: 1) evolution to keep up with the changes introduced in the framework (framework-based evolution); 2) evolution in response to the specific requirements and desired qualities of the stakeholders of the systems itself (general evolution).

To facilitate framework based evolution Eclipse SDK distinguishes between stable and supported APIs (*good interfaces*) and unstable, discouraged and unsupported non-APIs (*bad interfaces*) [37]. The latter are indicated by the substring `internal` in the package name. While "good" interfaces can be safely used in Eclipse-based systems, the use of "bad" interfaces is at risk of arbitrary change or removal without notice. It has been shown that many "bad" non-APIs are among packages that are most likely to introduce a post-release failure [94].

In this chapter we present preliminary results on usage of Eclipse *good interfaces* and *bad interfaces*. We investigate to *what extent* third-party plug-ins use *bad interfaces*, whether developers *continue to use* them and what are the *differences* between the third-party plug-ins that use *bad interfaces* and those that do not. We also investigate commonly used *bad interfaces* by the plug-ins. The answers to these questions provide Eclipse SDK developers with feedback on the current use of APIs and non-APIs as opposed to the expected use. Furthermore, these answers also create a solid basis for continuation studies related to APIs, *bad interfaces* and their use in third-party plug-ins.

To answer these questions, we conducted an empirical study based on a total of 512 Eclipse third-party plug-ins altogether having a total of 1,873 versions collected from SourceForge. The third-party plug-ins collected date back from 2003 to 2010. The number of versions considered requires a light-weight analysis approach (cf. [46]). Therefore, our analysis incorporates naming conventions and package imports rather than more precise but also more computationally challenging techniques such as call graph construction [49].

The remainder of the chapter is organized as follows. In Section 4.2 we present the notion of Eclipse plug-ins and their interfaces. In Section 4.3 we present how we collected the data. In Sections 4.4 and 4.5 we analyze metrics for the identified groups of Eclipse third-party plug-ins. In Section 4.6 we analyze commonly used *bad interfaces*. In Section 4.7 we discuss the threats to validity. In Section 4.8 we discuss the related work and finally, in Section 4.9 we present the conclusions and future work.

4.2 Eclipse Plug-in Architecture

Eclipse SDK is an extensible platform that provides a core of services for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by wrapping their tools in pluggable components, called *Eclipse plug-ins*, which conform to the Eclipse's plug-in contract. Plug-ins are bundles of code and/or data that contribute functions to a software system. Functions can be contributed, e.g., in the form of code libraries, platform extensions or documentation.

The plug-ins in the Eclipse framework can be categorized three main groups: Eclipse core plug-ins, Eclipse extension plug-ins, and Eclipse Third-party plug-ins.

- *Eclipse core plug-ins (ECPs)*: These are plug-ins present in and shipped as part of the Eclipse SDK. In this thesis, the ECPs and Eclipse SDK are interchangeable. The ECPs provide core functionality upon which all plug-in extensions are built. The ECPs also provide the runtime environment in which other plug-ins are loaded, integrated, and executed. The fully qualified names of ECP packages starts with `org.eclipse`. Examples of ECPs include Java development tools (JDT), Standard Widget Toolkit (SWT) and Platform runtime and resource management (Core) [45].

- *Eclipse extension plug-ins (EEPs)*: These are plug-ins built with the main goal of extending the Eclipse SDK. Most EEPs are large, generic, applications frameworks with tool plug-ins to build other specialized applications. Like the ECPs, fully qualified names of EEP packages start with `org.eclipse`, but as opposed to the ECPs, the EEPs are not shipped as part of the Eclipse SDK. Popular EEPs include J2EE Standard Tools, Eclipse Modeling Framework and PHP Development Tools [45].

- *Eclipse Third-party plug-ins (ETPs)*: These are the remaining plug-ins. The size of these plug-ins ranges from large application frameworks to small specialised applications. Unlike ECPs and EEPs, the package names of ETPs do not have a prefix `org.eclipse`. All the ETPs use at least some functionality provided by ECPs but also may use functionality provided by EEPs. The ETPs are also sometimes referred as Eclipse products/solutions and sometimes Eclipse extensions.

The Eclipse has two main types of interfaces, i.e., visible features that can be reused, it provides to ETPs that reuse its functionality: non-APIs and APIs [5, 37, 38].

- ***Eclipse non-APIs ("bad"):*** The non-APIs, which we also term as *bad interfaces*, are internal implementation artifacts that are found in a package with the substring "internal" in a the fully qualified package name according to Eclipse naming convention [37]. The internal implementations include public Java classes or interfaces, or a public or protected method, or field in such a class or interface. Users are strongly discouraged from adopting any of the non-APIs since they may be unstable [38]. Eclipse clearly states that clients who think they must use these non-APIs do it at their own risk as non-APIs are subject to arbitrary change or removal without notice. Eclipse does not usually provide documentation and support to the non-APIs.

- ***Eclipse APIs ("good"):*** The APIs, which we also term as *good interfaces*, are the public Java classes or interfaces that can be found in packages that do not contain the segment "internal" in the fully qualified package name, or a public or protected method, or field in such a class or interface. Eclipse states that, the APIs are considered to be stable and therefore can be used by any application developer without any risk. Furthermore, Eclipse also provides documentation and support for these APIs.

4.3 Data Set

How we collected the data used this chapter is explained in Chapter 3. In this chapter, we only analyse the *clean* ETPs, i.e., the ETPs that remained after eliminating noise (see Chapter 3). The ETPs were classified based on their dependency on the Eclipse SDK (ECPs) interfaces. We define *dependencies-on-ECPs* as *import statements* from

ECPs starting with the prefix `org.eclipse` in the source code of an ETP. The ETPs were classified based on evolution of their dependencies on "good" interfaces and "bad" interfaces:

I ETPs with all versions dependent solely on APIs—*good ETPs*;

II ETPs with all versions depending on a non-API—*bad ETPs*;

III ETPs with earlier versions dependent solely on APIs and latter versions depending on a non-API—*good–bad ETPs*;

IV ETPs with earlier versions dependent on a non-API and latter versions depending solely on APIs—*bad–good ETPs*;

V ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API—*oscillating ETPs*.

Results of the classification are shown in Table 4.1. The table is divided into five groups: *I (good)*—Classification I ETPs (good ETPs), II (bad)—Classification II ETPs (bad ETPs), III (bad–good)—Classification III ETPs (bad–good ETPs), IV (good–bad)—Classification IV ETPs (good–bad ETPs), and V (oscillating)—Classification V ETPs (oscillating ETPs).

For Classification I ETPs (good ETPs) in Table 4.1, the cell entry (2004,2004), typeset in italics, contains a pair (33 68) indicating that there are a total of 33 ETPs that were first released in the year 2004 on SourceForge altogether having a total of 68 versions. The pair (4 11) in the cell (2004,2005) means that there were 4 ETPs of the 33 that had new versions released in the year 2005 with a total of 11 versions in that year. We observe that the trend on the evolution in the *total* number of ETPs is non-monotone, for example, (2003,2007)=1, (2003,2008)=2, (2003,2009)=0. This indicates that, while an ETP may have version(s) in a given year, it does not release any version(s) in the subsequent year(s) and resumes releasing later.

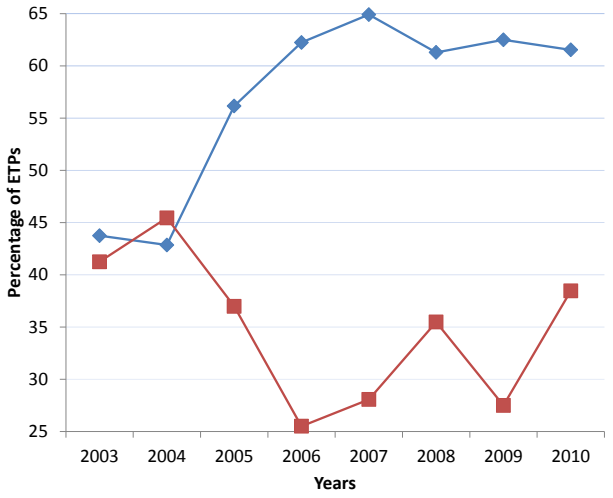


Figure 4.1: Percentages of ETPs in Classifications I (diamond) and II (square). The Y-axis scale is normalized by getting the percentage of ETP in each Classification from the total number of clean ETPs in the corresponding year.

		2003		2004		2005		2006		2007		2008		2009		2010	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
I (good)	2003	*35*	*62*	10	20	3	4	1	1	1	4	2	2	0	0	0	0
	2004			*33*	*68*	4	11	4	9	2	3	2	2	0	0	0	0
	2005					*41*	*66*	10	21	4	5	3	4	1	1	1	1
	2006							*61*	*111*	7	13	1	1	0	0	2	3
	2007									*37*	*83*	12	22	4	6	6	12
	2008											*38*	*74*	7	12	2	4
	2009													*25*	*30*	3	4
	2010															*16*	*28*
	Total	35	62	43	88	48	81	76	142	51	108	58	105	37	49	30	52
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
II (bad)	2003	*33*	91	18	45	11	19	8	13	6	8	2	4	1	2	1	1
	2004			*35*	*77*	8	24	4	9	4	13	4	15	1	2	2	3
	2005					*29*	*60*	10	26	9	31	7	15	7	19	5	20
	2006							*25*	*61*	10	32	3	19	2	11	5	15
	2007									*16*	*31*	2	3	1	2	0	0
	2008											*22*	*42*	7	15	1	3
	2009													*11*	*11*	3	7
	2010															*10*	*11*
	Total	33	91	53	122	48	103	47	109	45	115	40	98	30	62	27	60
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
III (bad–good)	2003	*8*	*31*	5	12	5	6	3	3	2	2	1	1	0	0	0	0
	2004			*4*	*11*	2	3	2	7	2	6	2	5	1	1	0	0
	2005					*3*	*28*	2	12	1	4	1	1	0	0	0	0
	2006							*9*	*34*	4	14	2	11	0	0	2	5
	2007									*3*	*11*	3	8	2	4	0	0
	2008											*2*	*3*	0	0	1	1
	2009													*3*	*16*	1	1
	2010															*0*	*0*
	Total	8	31	9	23	10	37	16	56	12	37	11	29	6	21	4	7
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
IV (good–bad)	2003	*0*	*0*	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2004			*3*	*5*	3	10	2	4	0	0	0	0	0	0	1	5
	2005					*1*	*1*	1	1	0	0	1	1	1	3	1	2
	2006							*1*	*2*	1	3	0	0	0	0	1	1
	2007									*0*	*0*	0	0	0	0	0	0
	2008											*0*	*0*	0	0	0	0
	2009													*1*	*2*	1	1
	2010															*0*	*0*
	Total	0	0	3	5	4	11	4	7	1	3	1	1	2	5	4	9
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
V (oscillating)	2003	*3*	*11*	3	10	2	10	3	10	3	11	1	1	1	6	2	5
	2004			*2*	*1*	2	6	2	5	1	3	2	2	1	3	1	1
	2005					*0*	*0*	0	0	0	0	0	0	0	0	0	0
	2006							*2*	*15*	2	8	1	3	1	2	1	4
	2007									*1*	*9*	1	4	1	5	0	0
	2008											*0*	*0*	0	0	0	0
	2009													*0*	*0*	0	0
	2010															*0*	*0*
	Total	3	11	5	20	4	16	7	30	7	31	5	10	4	16	4	10

Table 4.1: For the given first release year y_1 and an additional release year y the table shows the number of ETPs (E) first released in y_1 and also released in y, and the total number of versions (V) for these ETPs. If $y_1 = y$ we show the number of all ETPs released in this year and their versions.

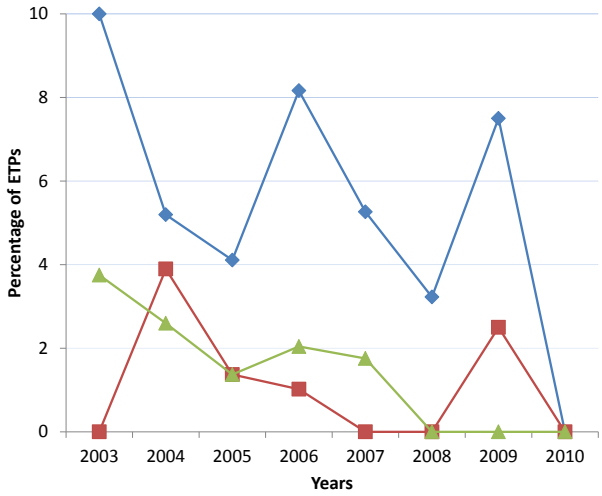


Figure 4.2: Percentages of ETPs in Classifications III (diamond), IV (square) and V (triangle). The Y-axis scale is normalized by getting the percentage of ETP in each Classification from the total number of clean ETPs in the corresponding year.

Figure 4.1 and 4.2 shows the graphs of the different classifications.

First, adding the number of ETPs along the diagonals in Table 4.1 we observe that 286 ETPs (55.8%) belong to Classification I, i.e., do not depend on non-APIs, while 224 (44.2%) ETPs belong to Classifications II–V, i.e., there is at least one version for each ETP that depends on at least one *bad interface*. There is, therefore, a significant number of ETPs depending on "bad" interfaces. Second, Figure 4.1 shows an increasing trend followed by stabilization for the percent of ETPs in Classification I and no clear trend for the percent of ETPs in Classification II. One of the possible reasons is that Eclipse is making it harder for developers of ETPs in Classification II. Third, we observe that many more ETPs continuously depend on *bad interfaces* (Classification II) than those that removed dependencies on *bad interfaces* at some point of time (Classification IV). This indicates that the elimination of non-APIs in the evolving ETP is very limited. Finally, comparing the numbers in Classifications III and IV we observe that there are more ETPs that start depending on non-APIs (Classification III) compared to those that eliminate the *bad interfaces* (Classification IV). A possible reason for the use of non-APIs could be that the functionality required is available solely via these *bad interfaces*.

In the metrics analysis discussed in the next section, we quantitatively discuss Classifications I and II ETPs. Since the number of ETPs in Classifications III–V are few, we qualitatively discuss these ETPs in Chapter 5 with the help of source compatibility analysis that we introduce in Section 5.5 of the same chapter.

4.4 Metrics Analysis: Classification I ETPs vs. Classification II ETPs

In this section we are interested in identifying the differences in characteristics between Classification I ETPs and Classification II ETPs. During the study in Chapter 2, with the limited sample of carefully selected ETPs, we have informally observed that ETPs

that depended on at least one *bad interface* are larger systems compared to those that depended on *good interfaces* only. In this section, we formally verify the observation on a larger scale.

Hence, we consider two metrics related to the size of a plug-in:

- *NOF*: the number of Java files,
- *NOF-D*: the number of Java files that have at least one `import` statement related to ECP classes or interfaces.

Furthermore, we want to understand whether Classifications I ETPs and II ETPs differ in the amount of ECP functionality used. We consider, therefore, two metrics related to the amount of ECP functionality imported by a plug-in:

- *D-Tot*: the number of `import` statements related to ECP classes and interfaces,
- *D-Uniq*: the number of unique `import` statements related to ECP classes and interfaces.

We conducted our study using two data-sets of ETPs *Data-set I* and *Data-set II*:

- *Data-set I*: For each year considered *Data-set I* includes one version of every Classification I or II ETP that has been first released in that year (cf. diagonal cells in Table 4.1)
- *Data-set II*: Similarly, for each year considered *Data-set II* includes one version of every Classification I or II ETP that has been first released in that year or earlier (cf. total cells in Table 4.1). Since we may have more than one version for an ETP released in one year, we select the last version in that year.

4.4.1 Metrics distribution

The distribution of all metrics studied is similar for both data sets: skewed to the right and with outliers. The outliers in the data are real since there are very large ETPs that have numerous dependencies on ECP interfaces. Figure 4.3 shows an example of metric distribution histograms: distribution of the D-Uniq values for Data-set II. Per metrics and per data-set we present 16 histograms corresponding to eight years (2003–2010) and two classifications (I and II). In the histograms in Figure 4.3 we observe that the concentration of the D-Uniq values in Classification I is on the lower side of x-axis for all the years, and the distribution is more spread in Classification II. The rest of the metrics show a similar trend and can be found in Appendix B.

4.4.2 Hypothesis testing

To verify validity of the observations made in Section 4.4.1, we perform statistical hypothesis testing. We formulate the null hypotheses, $H_0^{m,y,d}$ and the alternative hypotheses, $H_a^{m,y,d}$ for each of metrics $m \in \{\text{NOF}, \text{NOF-D}, \text{D-Tot}, \text{D-Uniq}\}$, year y, $2003 \leq y \leq 2010$, and data-set $d \in \{\text{I}, \text{II}\}$. The null hypotheses state that on the data-set d the values of m for Classification I and II ETPs released in year y originate from the same distribution, The alternative hypotheses state that values of the metrics m for Classification II are

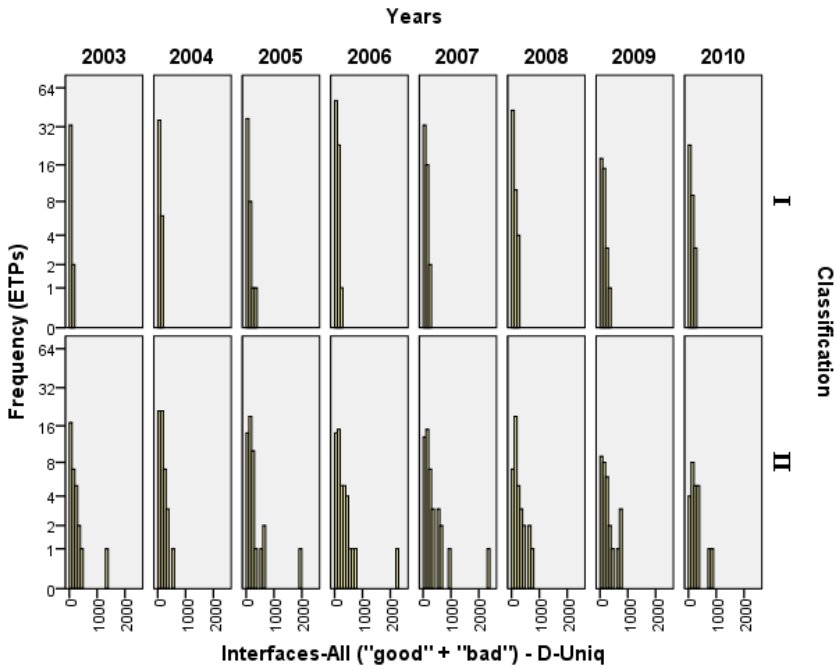


Figure 4.3: Distributions of D-Uniq: The ETPs in Classification II have higher dependency on ECP than those of Classification I, but their dependency on "bad" interfaces remains low.

	#CI	#CII	NOF	NOF-D	D-Tot	D-Uniq
Data-set I						
2003	35	33	0.001	0	0	0
2004	33	35	0.011	0.004	0	0
2005	40	29	0.004	0	0	0
2006	61	25	*0.424*	0.029	0.017	0.013
2007	37	16	*0.068*	0.018	0.005	0.007
2008	38	22	0.010	0.003	0.001	0
2009	25	11	*0.614*	*0.556*	*0.527*	*0.527*
2010	16	10	0.027	0.007	0	0
Data-set II						
2003	35	33	0.001	0	0	0
2004	42	53	0.002	0	0	0
2005	47	48	0.001	0	0	0
2006	76	47	0.002	0	0	0
2007	51	45	0.002	0	0	0
2008	57	40	0	0	0	0
2009	37	30	0.027	0.008	0.009	0.002
2010	30	27	0.011	0	0	0

Table 4.2: p-values for Classification I (CI) and II (CII) in Data-sets I and II. Zeros indicate values too small to be precisely determined by SPSS. Values exceeding 0.05 are typeset in italics.

higher than for Classification I. The choice for the "higher" the alternative hypothesis is based on a pilot study we carried out.

Since the number of ETPs for Classification I and II are relatively low, especially in the recent years, we chose a less stringent nonparametric test, *Mann-Whitney U*, as opposed to the *two-independent-sample t* test that depends on the assumption of *normality* and relatively high number of data points [82]. In total we have to carry out sixty-four Mann-Whitney tests corresponding to four metrics, two data-sets and eight years (2003–2010).

Table 4.2 shows the p-values for the sixty-four Mann-Whitney tests. Assuming a common threshold of 0.05, for Data-set I we can reject the null-hypotheses for most year/metric combinations, except for those indicated in italics in Table 4.2. Thus, for most year/metrics combinations in Data-set I we accept the corresponding alternative hypotheses. For Data-set II the null-hypotheses can be rejected for all years and all metrics. Hence, for Data-set II we can confidently claim that the metrics values for ETPs in Classification II are higher than for Classification I, i.e., ETPs in Classification II have more Java files, more files dependent on ECPs, more dependencies on ECPs and more unique dependencies on ECPs.

Comparing the results obtained for Data-set I and Data-set II we observe that the results are similar: null hypotheses can be rejected for most years and metrics both for Data-set I and for Data-set II with 2009 and NOF being exceptions. In general, higher p-values on Data-set I can be attributed to low numbers of the data points in this data-set as opposed to Data-set II. Specifically, inability to reject $H_0^{m,2009,I}$ can be explained by the fact that eleven Classification II ETPs only were released in 2009 and included in Data-set I.

One possible reason for Classification II ETPs being larger than Classification I, may be that the functionality required by ETPs is absent from *good interfaces*. We also

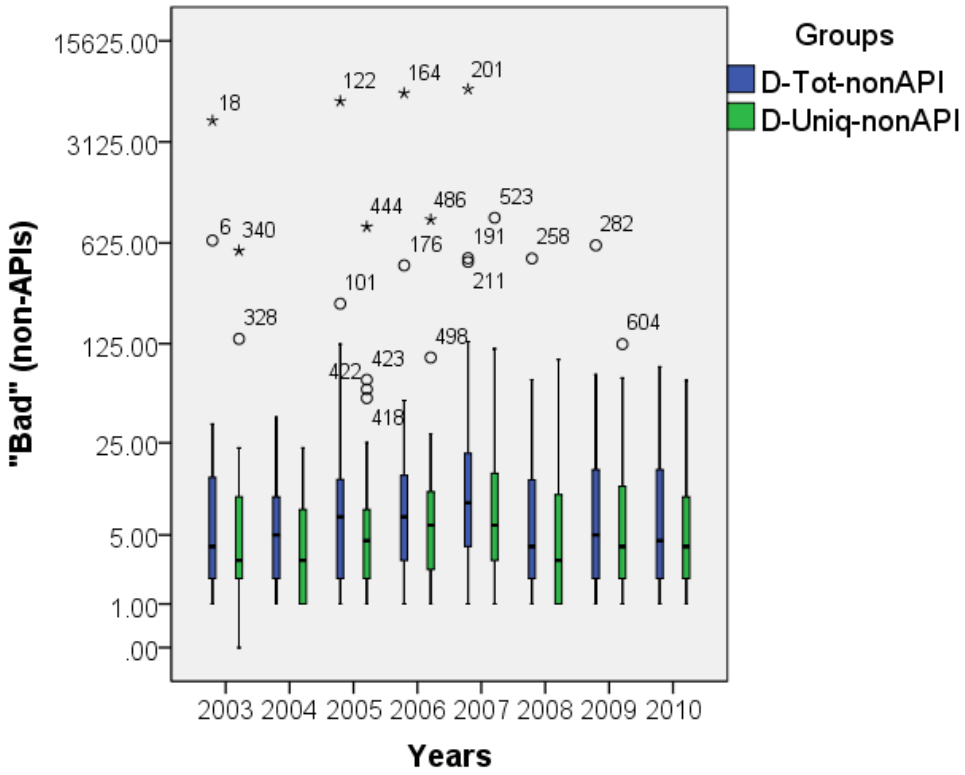


Figure 4.4: Distributions of D-Uniq and D-Tot for "bad" interfaces: The ETPs in Classification II have higher dependency on ECP than those of Classification I, but their dependency on "bad" interfaces remains low.

conjecture that although developers might be aware that the *bad interfaces* are volatile and unsupported, they may prefer to suffer the consequences of using these *bad interfaces* than building their own APIs. We verify this conjecture in Chapter 7.

4.5 Metrics analysis: Non-API usage

In this section we focus on Classification II and consider numbers of *bad interfaces* used by the ETPs. Figure 4.4 presents ETPs of Data-set II and visualizes *D-Tot-nonAPI*, the number of `import` statements related to non-APIs, and *D-Uniq-nonAPI*, the number of unique `import` statements related to *bad interfaces*. Comparing Figure 4.3 and Figure 4.4 we observe that although the ETPs in Classification II have a high dependency on ECP interfaces, medians in the box-plots reveal that the ETPs have a low dependency on *bad interfaces*.

To formalize this observation, we conduct a statistical hypothesis testing whether *D-Tot-nonAPI* and *D-Tot*, as well as *D-Uniq-nonAPI* and *D-Uniq* represent different populations. We formulate, therefore, the following null and alternative hypotheses:

- $H_0^{\text{Tot},y,d}$: metric values of *D-Tot* and *D-Tot-nonAPI* obtained for ETPs in data-set d and year y represent the same population;

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
# ETPs	28	15	14	12	11	9	8	7	6	5	4	3	2
# non-APIs	1	1	1	1	1	2	4	5	9	9	25	64	200

Table 4.3: Summary of the common non-APIs used by at least two of the 181 ETP-non-APIs. For example, in column 6, each of the 2 non-APIs is commonly used by 9 of the 181 ETP-non-APIs and in column 10, each of the 9 non-APIs is commonly used by 5 of the 181 ETP-non-APIs. The non-APIs considered in this table still exist in Eclipse SDK 3.7.

	Data-set I		Data-set II	
	Tot	Uniq	Tot	Uniq
2003	345	118.5	345	118.5
2004	281.92	111.07	308.5	121.5
2005	517.72	155	449.03	148.5
2006	306	108.5	594.14	176
2007	490.5	123.75	777.5	172.5
2008	320.5	128	597	178.37
2009	377.75	92.25	754.5	194.5
2010	221	114.5	683.53	192.3

Table 4.4: Hodges-Lehmann estimation of the median of the difference between the number of (unique) interfaces and the number of (unique) non-APIs.

- $H_a^{\text{Tot},y,d}$: metric values of *D-Tot* obtained for ETPs in data-set d and year y are higher than those obtained for *D-Tot-nonAPI*;
- $H_0^{\text{Uniq},y,d}$: metric values of *D-Uniq* and *D-Uniq-nonAPI* obtained for ETPs in data-set d and year y represent the same population;
- $H_a^{\text{Uniq},y,d}$: metric values of *D-Uniq* obtained for ETPs in data-set d and year y are higher than those obtained for *D-Uniq-nonAPI*.

Our choice for "greater" as the directed alternative stems from the fact that *bad interface* dependencies, counted by *D-Tot-nonAPI* and *D-Uniq-nonAPI* are a special kind of dependencies, while *D-Tot* and *D-Uniq* include both API and non-API dependencies. By the same argument the metric values are related, i.e., we have to conduct a paired two-sample test. We start by conducting a series of Shapiro-Wilk tests to check whether metric values are distributed normally. Depending on the outcome of these tests we should either use the well-known t test for two dependent samples (if the metric values are distributed normally) or its non-parametric counterpart, the paired two-sample Wilcoxon test.

The p-values obtained in the series of Shapiro-Wilk tests never exceeded 0.0003, i.e., normality hypothesis can be confidently rejected. Therefore, we conducted a series of paired two-sample Wilcoxon tests. Based on the p-values obtained we reject $H_0^{\text{Tot},y,d}$ and accept $H_a^{\text{Tot},y,d}$ for all years y and both data-sets d: the p-values never exceeded 0.001. Similarly, based on the p-values obtained we reject $H_0^{\text{Uniq},y,d}$ and accept $H_a^{\text{Uniq},y,d}$ for all years y and both data-sets d: the p-values never exceeded 0.001.

To provide better insights in the differences between *D-Tot* and *D-Tot-nonAPI* as well as between *D-Uniq* and *D-Uniq-nonAPI* we estimate the median of the difference

between the corresponding metric values.[1] Table 4.4 summarizes the Hodges-Lehmann estimator values [52,91]. The estimator values support the observation made by comparing Figure 4.3 and Figure 4.4: although the ETPs in Classification II have a high dependency on ECP interfaces, medians in the box-plots reveal that the ETPs have a low dependency on *bad interfaces*.

Furthermore, in a preliminary investigation, we also found that most ETPs use *bad interfaces* directly without using *wrappers*. In Chapter 5 we report that *bad interfaces* are the main cause of incompatibilities in new releases of the SDK framework because these bad interfaces frequently change in the new SDK releases. Moreover, in Chapter 7 we discovered that the reason why developers us *bad interfaces* is because there are no *good interfaces* with the necessary functionality. In this chapter, we have discovered that on average not so many *bad interfaces* are being used and the number of D-Tot is greater than D-Uniq, i.e., some *bad interfaces* are being frequently used in the ETPs' source code. To reduce the amount of work in fixing the incompatibilities of the EPTs in new SDK releases, ETP developers should avoid using the *bad interfaces* directly but instead use *wrappers*. With the help of a wrapper, fixing the incompatibilities caused by a changed *bad interfaces* that is frequently called in the ETP's source code will only be done in the wrapper.

4.6 Commonly used non-APIs

In this section we continue our discussion of Classification II ETPs and the *bad interfaces* used by these ETPs. We analyze the used *bad interfaces* by ETPs in order to get a quantitative insight of what kind of *bad interfaces* are highly used and what their distribution looks like. This would help reveal to the Eclipse SDK developers possible *bad interfaces* that would be good candidates to become *good interfaces* in the future.

We looked at the 181 ETPs in Classification II. Since we have a number of versions for each ETP, we decided to choose the last version for each ETP. A total of 1,717 unique non-APIs were extracted from the ETPs. 1,525 of the 1,717 non-APIs still exist in Eclipse SDK 3.7, the latest major release. We wrote a script that searches through the non-APIs for all ETPs and returns the ETPs that use a given non-API. Figure 4.5 and Table 4.3 show the distribution of the frequency of use of the non-APIs. The distribution is positively skewed: there are *many non-APIs used by few ETPs* and there are *few non-APIs used by many ETPs*. Furthermore, the ETPs in the *many non-APIs used by few ETPs* are just a small subset of the total 181 ETPs. This indicates that the ETPs use a diverse set of *bad interfaces*. The most popular non-APIs are presented in Table 4.5 while the complete list of non-APIs and their frequencies can be found in Appendix B.

By observing Table 4.5 one might conjecture that the most popular non-APIs are related to Eclipse Java development tools (JDT). This conjecture is confirmed by Figure 4.6. Figure 4.6a shows the distribution of the number of non-APIs used by at least two ETPs across different Eclipse projects. Similarly, Figure 4.6b shows the distribution of the use, i.e., the total number of import statements referring to non-APIs, across different Eclipse projects. Both figures suggest that the lion share indeed belongs to the JDT project. However, not less than ten different Eclipse projects deliver non-APIs used by at least two ETPs.

[1] We stress that the median of the difference is not the same as the difference in medians.

non-APIs	# ETPs
org.eclipse.jdt.internal.ui.JavaPlugin	28
org.eclipse.jdt.internal.core.JavaProject	15
org.eclipse.ui.internal.ide.IDEWorkbenchPlugin	14
org.eclipse.jdt.internal.corext.util.JavaModelUtil	12
org.eclipse.jdt.internal.ui.JavaPluginImages	11
org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor	9
org.eclipse.jdt.internal.core.PackageFragment	9
org.eclipse.jdt.internal.ui.wizards.TypedElementSelectionValidator	8
org.eclipse.core.internal.resources.Workspace	8
org.eclipse.jdt.internal.ui.util.ExceptionHandler	8
org.eclipse.jdt.internal.core.SourceType	8

Table 4.5: A sample of commonly used non-APIs by the ETPs ranked according to the highest frequency to lowest frequency. The column *# ETPs* shows the number of ETPs of the 181 ETPs that use the corresponding non-API.

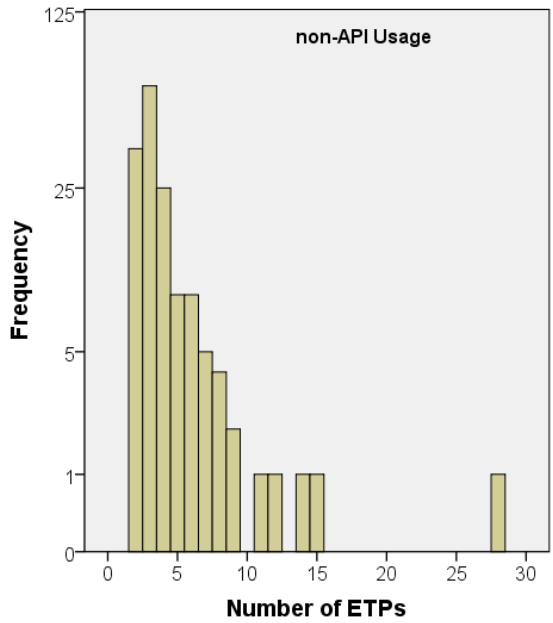
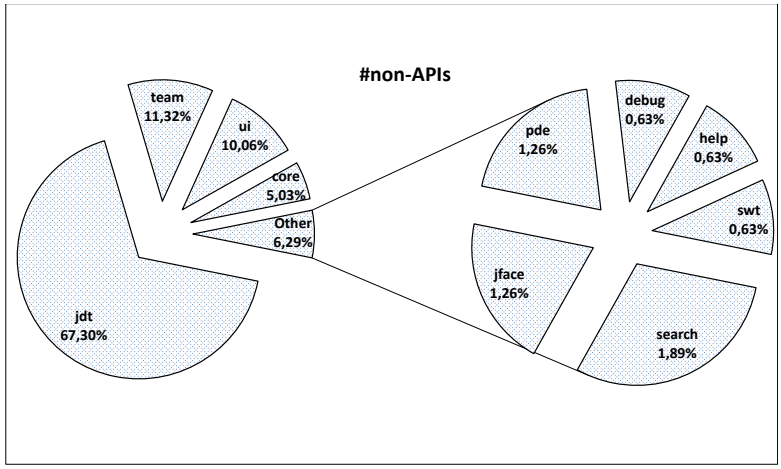
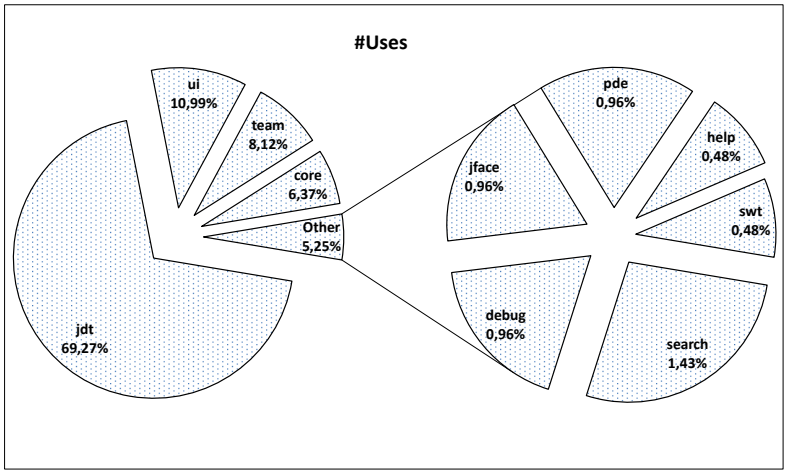


Figure 4.5: Distribution of frequency of non-API use. Non-APIs used by only one ETP have not been included.



(a) The number of non-APIs used by at least two ETPs across different Eclipse projects.



(b) The total number of import statements referring to non-APIs (used by at least two ETPs), across different Eclipse projects.

Figure 4.6: Distribution of non-API usage per project.

4.7 Threats to Validity

As any other empirical study, our analysis may have been affected by validity threats. In our analysis, *construct validity* may be threatened by the metrics analysis in Sections 4.4 and 4.5. Indeed, presence of `org.eclipse.*` imports can result in incorrect counts in D-Tot and D-Uniq. However, for ETPs released in 2006 and earlier, less than 3% of the total imports have this form. The figure is even lower for ETPs released in 2007 or later. Additional threat pertains to unused imports. In our fact extraction unused imports are not excluded in the metrics. We tried to mitigate *internal validity* threats during the data collection by removing the incomplete ETPs and ETPs with no ECP-depencies. For *external validity*, it is possible that our results may not be generalizable since we only considered ETPs from Sourceforge. Sourceforge is, however, a big and well-known repository, and, therefore, our ETPs' collection can be considered representative.

4.8 Related Work

Mileva et al. study popularity of APIs and state that, analyzing popularity of software projects is a relatively new research field [76]. Holmes and Walker [53] present a prototype tool calculating a popularity measure for every API in a framework. As opposed to our study that spans a period of eight years, Holmes and Walker focus on one specific version of Eclipse. Mileva et al. [76] investigate API popularity on data collected from 200 open-source projects. The authors developed a tool prototype that analyzes the collected information and plots API element usage trends. As opposed to our focus on Eclipse, they present a general study of APIs usage in any given project.

The study that is more closely related to ours, is by Lämmel, Pek and Starek [63]. The authors demonstrate a scalable approach to AST-based API-usage analysis for a large-scale corpus of open-source projects. Their investigation reports on usage of 77 APIs from different domains extracted from built projects, reference projects and unbuilt projects. In comparison to our study, our API usage analysis considers APIs from the same domain, namely, Eclipse APIs. Since we only consider usage by only looking at the imports in a project, we did not need to build the studied projects. The study of Grechanik et al. explore API usage in Java programs on the large scale [48]. Finally, the study of Lämmel et al. presents a new approach of understanding reuse characteristics of composite frameworks such as .NET and Java Standard Edition [62]. The characteristics of reuse defines by the authors include metrics of potential reuse (such as percentage of socializable types), categories related to reuse (such as open and close namespaces), and metrics of actual reuse (such as percentage of specialized types).

This work builds on and extends the previous work on the evolution of ETPs [22]. In the previous study, we investigated the evolution of dependencies on ECPs of 21 carefully selected ETPs. We, however, did not distinguish between "good" APIs and "bad" non-APIs. The current study is one of the follow-up studies we pointed out in [22].

4.9 Conclusions and Future Work

In this paper, we have investigated Eclipse SDK API usage by means of a case study of Eclipse third-party plug-ins. Our conclusion is based on empirical results for data collected on 512 Eclipse third-party plug-ins altogether having a total 1,873 versions. We discovered that about 44% of the 512 Eclipse third-party plug-ins depend on "bad"

non-APIs and also discovered that developers continue using "bad" non-APIs in the new versions of the third-party plug-ins. The subsequent empirical study of 467 plug-ins also showed that plug-ins that use or extend at least one *bad interface* are comparatively larger and also use more functionality from Eclipse than those that use or extend only "good" APIs. We also found out that ETP-non-APIs use a diverse set of "bad" APIs. This information provides Eclipse SDK developers with feedback on the current use of APIs and non-APIs in ETPs as opposed to the expected use. Furthermore, it reveals the characteristics of these ETPs that use the *good interfaces* and *bad interfaces*. The information can be used as a starting point to answer questions like: why these situation occur and how they can be mitigated in future *good interface* releases.

Furthermore, in a preliminary investigation, we also found that most ETPs use *bad interfaces* directly without using *wrappers*. In Chapter 5 we report that *bad interfaces* are the main cause of incompatibilities in new releases of the SDK framework. Moreover, in Chapter 7 we discovered that the reason why developers us *bad interfaces* is because there are no *good interfaces* with the necessary functionality. In this chapter, we have discovered that on average not so many *bad interfaces* are being used and some *bad interfaces* are being frequently used in the ETP's source code. To reduce the amount of work in fixing the incompatibilities of the EPTs in new SDK releases, ETP developers should avoid using the *bad interfaces* directly but instead use *wrappers*. With the help of a *wrapper*, fixing the incompatibilities caused by a changed *bad interfaces* that is frequently called in the ETP's source code will only be done in the wrapper.

We have also identified possible ways in which the study should be extended. First, one can investigate possible reasons why developers depend on *bad interfaces*. This question is considered in Chapter 7. In Section 4.4.2 we have already conjectured that absence of the required functionality from the APIs might be a possible reason for developers depend to choose for non-APIs. This conjecture is also investigated in Chapter 7. We also compare the failure rate of ETPs that depend on APIs and those that depend on at least one non-API when ported to new ECP releases (Chapter 5).

Chapter 5

Survival of ETPs

*Today numerous software systems are being developed on top of frameworks. Eclipse SDK is such a framework that has been in existence for over a decade. Like many other evolving software systems, the Eclipse SDK has both stable and supported APIs (*good interfaces*) and unstable, discouraged and unsupported non-APIs (*bad interfaces*). In this study, we investigate the survival of Eclipse third party plug-ins (ETPs) based on whether they use bad interfaces or not. We analyzed a total of* 512 *ETPs altogether having* 1,873 *versions.*

A number of observations are drawn from the study: First, we discovered that, the majority of ETPs do not produce new versions beyond the first year of release. Second, as stated by Eclipse, we confirm that indeed APIs are stable over subsequent Eclipse releases that do not involve API-breaking changes. We further confirm that non-APIs are indeed unstable. This observation on the stability of the Eclipse interfaces is based on how these interfaces affect compatibility of the ETPs in new Eclipse SDK releases. ETPs that depend on only good interfaces almost never fail in the subsequent Eclipse SDK releases and ETPs that use bad interfaces have a very high failure rate in new SDK releases. Furthermore, we observed that, ETPs that depend more on old non-APIs and less on newly introduced non-APIs, have a very high forward source compatibility success rate. Third, we observed that the reason why the non-APIs are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new SDK release. Finally, when eliminating the use of problematic non-APIs in the ETPs source code, we have observed that developers perform one of the following things: i) build their own API that has same functionality as the non-API, ii) find similar functionality offered by an API in the SDK, iii) completely eliminate the entities in the ETP source code that uses the functionality from the non-API and iv) when a non-API matures into an API, developers replace the functionality of the non-API with the new API.

5.1 Introduction

Today, many software developers build systems on top of frameworks [103] (e.g., currently Eclipse marketplace[1] reports over 1.4 million of Eclipse solutions). This approach has many advantages, such as reuse of the functionality provided [80] and increasing productivity [60]. However, these benefits are accompanied by co-evolutionary challenges that come along with using frameworks [20]: as the framework evolves, it makes changes to its APIs and these changes may cause the applications that use them to fail [33, 81, 112]. Framework-based applications are, therefore, subject to two dimensions of survival: 1) survival related to releasing new versions of the application itself, and 2) survival related to incompatibilities of the software system in new framework releases[2].

In this work we focus on a popular framework, Eclipse, and study survival of Eclipse third-party plug-ins (ETPs). As opposed to the previous work [16, 30, 31, 96, 100], we investigate survival in terms incompatibilities of the ETPs in the new framework releases. To measure source compatibility we count the number of Eclipse SDK releases an ETP can successfully compile with.

Understanding the survival related to incompatibilities of the ETPs in new framework releases is essential for the users of the ETPs and their maintainers. Users of the ETPs need to understand whether the ETP is likely to operate when the Eclipse SDK is updated to a new release, and whether the ETP will produce new versions. Maintainers of the ETPs should be clearly aware of the impact of the changes in the SDK on their plug-in. This understanding is complicated by the fact that while some SDK interfaces, *APIs*, are stable and supported, other SDK interfaces, *non-APIs*, are subject to arbitrary change or removal without notice [5, 37, 38]. ETP maintainers are strongly discouraged from adopting any of the non-APIs [38], and indeed, it has been shown that many non-APIs are among interfaces that are most likely to introduce a post-release failure [94]. However, despite this, in Chapter 4 we have observed that the use of non-APIs is not uncommon: 44.2% of the ETPs on SourceForge have at least one version that depends on at least one non-API [23]. Based on this observation, we formulate the following question that will be addressed in this chapter.

RQ3: *How does the compatibility of ETPs that depend solely on Eclipse APIs compare with that of ETPs that depend on at least one Eclipse non-API in new Eclipse SDK releases?*

The remainder of the chapter is organized as follows: In Section 5.2 we introduce Eclipse plug-ins and their interfaces and in Section 5.3 we explain the data collection process. In Section 5.4 we discuss the version release rates of the ETPs. In Section 5.5 we discuss the methodology we used to analyse the survival of the ETPs. In Section 5.6, we discuss the quantitative analysis of the survival of the ETPs. In Section 5.7 we discuss the qualitative analysis of the survival of ETPs. In Section 5.8, we discuss the threats to validity. In Section 5.9, we discuss the related work. And finally, In Section 5.10 we present the conclusions and future work.

[1] http://marketplace.eclipse.org

[2] Here and elsewhere our notion of survival is *not* related to a branch of statistics known as survival analysis [42], applied to study of software [92, 97]

5.2 Eclipse Plug-in Architecture

Eclipse SDK is an extensible platform that provides a core of services for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by wrapping their tools in pluggable components, called *Eclipse plug-ins*, which conform to the Eclipse plug-in contract. Plug-ins are bundles of code and/or data that contribute functions to a software system. Functions can be contributed, e.g., in the form of code libraries, platform extensions or documentation.

The plug-ins in the Eclipse framework can be categorized into three main groups: Eclipse core plug-ins, Eclipse extension plug-ins, and Eclipse Third-party plug-ins.

- *Eclipse core plug-ins (ECPs)*: These are plug-ins present in and shipped as part of the Eclipse SDK. In this thesis, the ECPs and Eclipse SDK are interchangeable. The ECPs provide core functionality upon which all plug-in extensions are built. The ECPs also provide the runtime environment in which other plug-ins are loaded, integrated, and executed. The fully qualified names of ECP packages starts with `org.eclipse`. Examples of ECPs include Java development tools (JDT), Standard Widget Toolkit (SWT) and Platform runtime and resource management (Core) [45].

- *Eclipse extension plug-ins (EEPs)*: These are plug-ins built with the main goal of extending the Eclipse SDK. Most EEPs are large, generic, applications frameworks with tool plug-ins to build other specialized applications. Like the ECPs, fully qualified names of EEP packages start with `org.eclipse`, but as opposed to the ECPs, the EEPs are not shipped as part of the Eclipse SDK. Popular EEPs include J2EE Standard Tools, Eclipse Modeling Framework and PHP Development Tools [45].

- *Eclipse Third-party plug-ins (ETPs)*: These are the remaining plug-ins. The size of these plug-ins ranges from large application frameworks to small specialised applications. Unlike ECPs and EEPs, the package names of ETPs do not have a prefix `org.eclipse`. All the ETPs use at least some functionality provided by ECPs but also may use functionality provided by EEPs. The ETPs are also sometimes referred as Eclipse products/solutions and sometimes Eclipse extensions.

The Eclipse framework has two main types of interfaces, i.e., visible features that it provides to ETPs that reuse its functionality: non-APIs and APIs [5, 37, 38].

- ***Eclipse non-APIs ("bad"):*** The non-APIs, which we also term as *bad interfaces*, are internal implementation artifacts that are found in a package with the substring "internal" in a the fully qualified package name according to Eclipse naming convention [37]. The internal implementations include public Java classes or interfaces, or a public or protected method, or field in such a class or interface. Users are strongly discouraged from adopting any of the non-APIs since they may be unstable [38]. Eclipse clearly states that clients who think they must use these non-APIs do it at their own risk as non-APIs are subject to arbitrary change or removal without notice. Eclipse does not usually provide documentation and support to the non-APIs.

- ***Eclipse APIs ("good"):*** The APIs, which we also term as *good interfaces*, are the public Java classes or interfaces that can be found in packages that do not contain the segment "internal" in the fully qualified package name, or a public or protected method, or field in such a class or interface. Eclipse states that, the APIs

are considered to be stable and therefore can be used by any application developer without any risk. Furthermore, Eclipse also provides documentation and support for these APIs.

5.3 Data Set

How we collected the data that we use this chapter is explained in Chapter 3. In this chapter, we only analyse the *clean* ETPs, i.e., the ETPs that remained after eliminating noise (see Chapter 3). The ETPs were classified based on their dependency on the Eclipse SDK (ECPs) interfaces. We define *dependencies-on-ECPs* as *import statements* from ECPs starting with the prefix `org.eclipse` in the source code of an ETP. The ETPs were classified based on evolution of their dependencies on "good" interfaces and "bad" interfaces:

I ETPs with all versions dependent solely on APIs—*good ETPs*;

II ETPs with all versions depending on a non-API—*bad ETPs*;

III ETPs with earlier versions dependent solely on APIs and latter versions depending on a non-API—*good–bad ETPs*;

IV ETPs with earlier versions dependent on a non-API and latter versions depending solely on APIs—*bad–good ETPs*;

V ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API—*oscillating ETPs*.

Results of the classification are shown in Table 5.1. The table is divided into four groups: *I (good)*—Classification I ETPs (*good ETPs*), II (bad)—Classification II ETPs (*bad ETPs*), III (bad–good)—Classification III ETPs (bad–good ETPs), IV (good–bad)—Classification IV ETPs (good–bad ETPs), and V (oscillating)—Classification V ETPs (oscillating ETPs).

For Classification I ETPs (*good ETPs*) in Table 5.1, the cell entry (2004,2004), typeset in italics, contains a pair (33 68) indicating that there are a total of 33 ETPs that were first released in the year 2004 on SourceForge altogether having a total of 68 versions. The pair (4 11) in the cell (2004,2005) means that there were 4 ETPs of the 33 that had new versions released in the year 2005 with a total of 11 versions in that year. We observe that the trend on the evolution in the *total* number of ETPs is non-monotone, for example, (2003,2007)=1, (2003,2008)=2, (2003,2009)=0. This indicates that, while an ETP may have version(s) in a given year, it does not release any version(s) in the subsequent year(s) and resumes releasing later.

5.4 Release of new ETP Versions (*good ETPs* vs *bad ETPs*)

In this section we discuss the release rates of new versions of ETPs between the good and *bad ETPs*. Since the number of ETPs and versions in Classification III—V are small, we will analyse the qualitatively in Section 5.7.

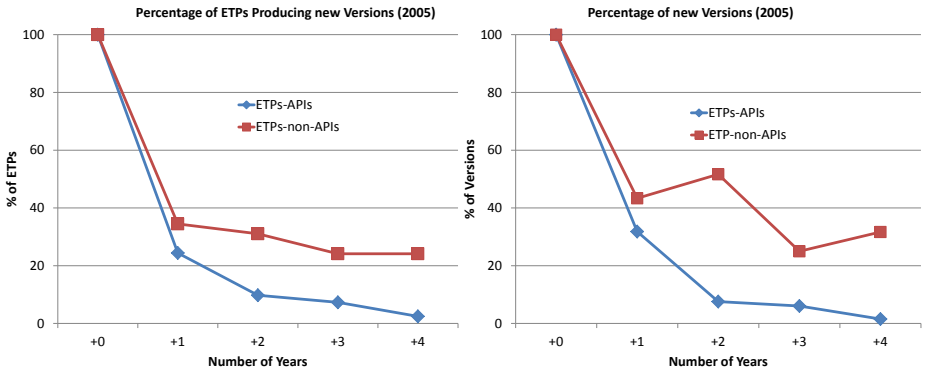
		2003		2004		2005		2006		2007		2008		2009		2010	
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
I (good)	2003	*35*	*62*	10	20	3	4	1	1	1	4	2	2	0	0	0	0
	2004			*33*	*68*	4	11	4	9	2	3	2	2	0	0	0	0
	2005					*41*	*66*	10	21	4	5	3	4	1	1	1	1
	2006							*61*	*111*	7	13	1	1	0	0	2	3
	2007									*37*	*83*	12	22	4	6	6	12
	2008											*38*	*74*	7	12	2	4
	2009													*25*	*30*	3	4
	2010															*16*	*28*
	Total	35	62	43	88	48	81	76	142	51	108	58	105	37	49	30	52
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
II (bad)	2003	*33*	91	18	45	11	19	8	13	6	8	2	4	1	2	1	1
	2004			*35*	*77*	8	24	4	9	4	13	4	15	1	2	2	3
	2005					*29*	*60*	10	26	9	31	7	15	7	19	5	20
	2006							*25*	*61*	10	32	3	19	2	11	5	15
	2007									*16*	*31*	2	3	1	2	0	0
	2008											*22*	*42*	7	15	1	3
	2009													*11*	*11*	3	7
	2010															*10*	*11*
	Total	33	91	53	122	48	103	47	109	45	115	40	98	30	62	27	60
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
III (bad–good)	2003	*8*	*31*	5	12	5	6	3	3	2	2	1	1	0	0	0	0
	2004			*4*	*11*	2	3	2	7	2	6	2	5	1	1	0	0
	2005					*3*	*28*	2	12	1	4	1	1	0	0	0	0
	2006							*9*	*34*	4	14	2	11	0	0	2	5
	2007									*3*	*11*	3	8	2	4	0	0
	2008											*2*	*3*	0	0	1	1
	2009													*3*	*16*	1	1
	2010															*0*	*0*
	Total	8	31	9	23	10	37	16	56	12	37	11	29	6	21	4	7
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
IV (good–bad)	2003	*0*	*0*	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2004			*3*	*5*	3	10	2	4	0	0	0	0	0	0	1	5
	2005					*1*	*1*	1	1	0	0	1	1	1	3	1	2
	2006							*1*	*2*	1	3	0	0	0	0	1	1
	2007									*0*	*0*	0	0	0	0	0	0
	2008											*0*	*0*	0	0	0	0
	2009													*1*	*2*	1	1
	2010															*0*	*0*
	Total	0	0	3	5	4	11	4	7	1	3	1	1	2	5	4	9
		E	V	E	V	E	V	E	V	E	V	E	V	E	V	E	V
V (oscillating)	2003	*3*	*11*	3	10	2	10	3	10	3	11	1	1	1	6	2	5
	2004			*2*	*1*	2	6	2	5	1	3	2	2	1	3	1	1
	2005					*0*	*0*	0	0	0	0	0	0	0	0	0	0
	2006							*2*	*15*	2	8	1	3	1	2	1	4
	2007									*1*	*9*	1	4	1	5	0	0
	2008											*0*	*0*	0	0	0	0
	2009													*0*	*0*	0	0
	2010															*0*	*0*
	Total	3	11	5	20	4	16	7	30	7	31	5	10	4	16	4	10

Table 5.1: For the given first release year y_1 and an additional release year y the table shows the number of ETPs (E) first released in y_1 and also released in y, and the total number of versions (V) for these ETPs. If $y_1 = y$ we show the number of all ETPs released in this year and their versions.

			+0	+1	+2	+3	+4
2005	ETPs	good ETPs	41	10	4	3	1
		bad ETPs	29	10	9	7	7
	Versions	good ETPs	66	21	5	4	1
		bad ETPs	60	26	31	15	19
2006	ETPs	good ETPs	61	7	1	0	2
		bad ETPs	25	10	3	2	5
	Versions	good ETPs	111	13	1	0	3
		bad ETPs	61	32	19	11	15

Table 5.2: Number of *good ETPs*, *bad ETPs* and their versions per year after the initial release for two release years 2005 and 2006.

We illustrate the release rate using an example of two ETP release years , i.e., 2005 and 2006. Table 5.2 presents an extract of the ETP and version release rates for two ETP release years 2004 and 2005. The first row of the table indicates the number of years after the initial release year of the ETPs. For example, +0 means the initial release year, +1 means one year after the initial release. The entry 41 in the second row of the table indicates that there were 41 *good ETPs* released for the first time in 2005. The entry 10 in the second row of the table indicates that 10 of the 41 *good ETPs* released in 2005 had new versions in the year 2006.



(a) (b)

Figure 5.1: Percentage of ETPs that produced new versions (a) and percentage of new versions produced (b).

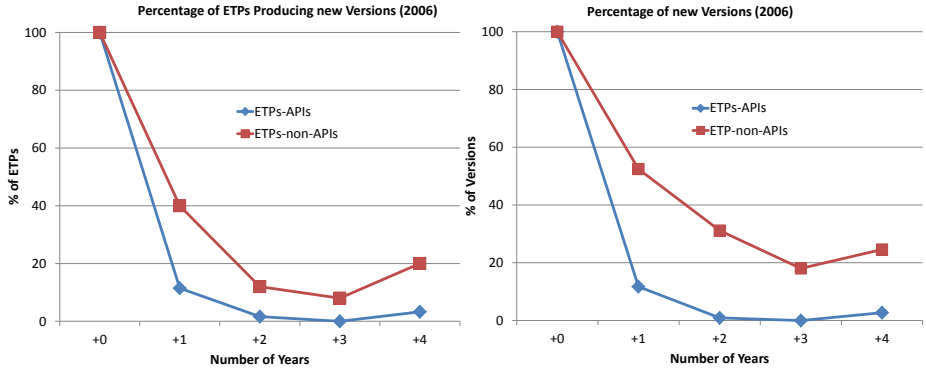


Figure 5.2: Percentage of ETPs that producing new versions (a) and percentage of new versions produced (b).

Figures 5.1 and 5.2 show a plot of the percentage equivalent of ETP and version release rates for the results in Table 5.2. The number of new versions is normalized with respect to the initial release numbers.

We observe that the plots for both Figure 5.1 and for 5.2 exhibit a clear decreasing trend. The decrease is sharpest from the initial release year to the subsequent year. This indicates that a large number of ETPs are not maintained immediately after the first year of their release. The observation supports the earlier findings [16, 88] that SourceForge projects have a low chance of evolving. Study of ETPs hosted at other repositories is considered as a future work. Furthermore, in both Figure 5.1 and 5.2, we observe a sharper decrease in the numbers of ETPs releasing new versions and versions of *good ETPs* than in *bad ETP*. The rest of the years show a similar trend.

One possible reason for lower values in *good ETPs* may be that developers of *good ETPs* are not forced to release new versions to keep up with changes in Eclipse SDK APIs. As opposed to *good ETPs*, *bad ETPs* can be expected to fail more often when ported to new releases of Eclipse SDK compared to *good ETPs* due to the use of unsupported and unstable non-APIs. We therefore conjecture that *good ETPs* have a higher *forward source compatibility success rate* when ported to newer releases of Eclipse SDK compared to *bad ETPs*. We verify this conjecture in Section 5.5.

5.5 Source Compatibility Between the ETPs and Eclipse

In this section, we discuss the survival related to incompatibilities between the ETPs and Eclipse SDK. To measure survival of an ETP developed on top of a given major Eclipse SDK release we determine the number of subsequent major Eclipse SDK releases that the ETP is compatible with (forward compatibility). Eclipse distinguishes between a number of compatibility notions [37, 41]. *API Binary Compatibility* requires that pre-existing binaries of the ETP link and run with new releases of the Eclipse SDK without recompiling. *API Source Compatibility* requires that the source code of the ETP needs to be recompiled to keep working with new releases of the Eclipse SDK but no changes have to be made in

the sources.

Any given change in the component API on which a client depends, may break none of the two compatibilities, one of the two compatibilities or both compatibilities. The scenarios below exemplify the two compatibility notions:

1. *Both Binary and Source compatible:* Changing a method body in a way that continues to behave the same.

2. *Binary compatible and Source incompatible:* Adding new method overloads. Since overload resolution is determined at compile time, adding new methods will not affect already-compiled binaries. If the client recompiles, it is possible that binaries may bind to the new overloads.

3. *Binary incompatible and Source compatible:* In this case, clients just need to recompile their sources to keep working. The compiler will respond to the change in a corrective way. For example, consider removing a method overload. At a binary level, the method the clients are bound to is removed and so things fail. But if you recompile, the compiler may bind to another overload that is semantically equivalent, and so things keep working without having to change any source.

4. *Both Binary and Source incompatible:* A breaking change. This requires clients to update their sources and recompile. For example, removing a method.

We analyze source compatibility related to scenarios 1, 2 and 4 above. Scenario 3 is predominantly related to binary compatibility. Furthermore, the ETP may also be subject to runtime incompatibilities [40, 112]. As the first empirical investigation on API compatibility, in study we only focus on source compatibility of the ETPs. In a followup study, one can build on our study and focus on binary and runtime compatibility.

In the analysis, we study both backward source compatibility and forward source compatibility. Below we define the two source compatibilities.

- *Backward source compatibility*: This means that source files written to use a given SDK release, will continue to compile and run against older SDK releases.

- *Forward source compatibility*: This means that source files written to use a given SDK release, will continue to compile and run against newer SDK releases.

5.5.1 Dependency Structure of an ETP

ETPs commonly depend on multiple software components such as ECPs, EEPs, external libraries and other ETPs (cf. Figure 5.3). We distinguish three types of dependencies that an ETP may have. Compulsory *direct dependency* links an ETP to at least one ECP. Recall that in Section 5.3 we have excluded from consideration ETPs that do not depend on ECPs. Optional *direct dependency* is present if an ETP depends on an external library, an EEP or even another ETP. Finally, we talk about an optional *indirect dependency* if an ETP depends on an EEP or another ETP, and these components also depend on an API from ECP. This API is said to be an indirect dependency of the ETP being studied. The study of source compatibility of an ETP is challenging because of the complex structure of the dependencies.

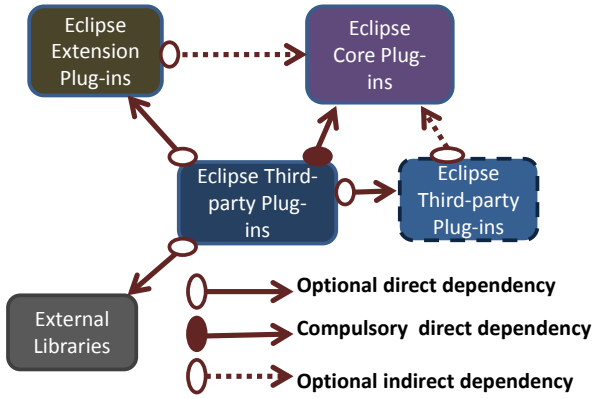


Figure 5.3: Dependency Structure of an ETP

5.5.2 Source Compatibility Check

To study source compatibility, we need to compile ETPs with different releases of Eclipse SDK. Initially, we were interested in comparing the success rate of *good ETPs* and *bad ETPs* when compiled with new Eclipse SDK releases, i.e., Eclipse SDK releases following the Eclipse SDK release on top of which the ETP has been built. This would require determining on top of which Eclipse SDK release a given ETP has been built. Unfortunately, less than 5% of the collected ETPs explicitly stated this information either by mentioning it in the SourceForge description or by recording it in the meta-data section of the manifest file of the ETP. The release year of the ETP can be used to determine *terminus ante quem*[3] of the Eclipse SDK release, but cannot be seen as the exact date. Indeed, there may be a time gap between beginning of the development process phase and choice of the SDK, and the end of the development process phase and publication of the ETP at SourceForge. Moreover, we have observed that some programmers prefer to develop plug-ins on top of earlier releases rather than on top of the most recent one, e.g., if those earlier releases are being perceived as being more stable.

Hence, we decided to check the source compatibility of the ETPs with *all* Eclipse SDK major releases.

To check for source compatibility of a version of an ETP with a given ECP release, we employ the methodology described in [18]. At compile time, we augment the *build path* for compiling a dependent ETP with the jar files of all the components containing the APIs used in the ETP's source code. If the ETP has direct dependencies on *other components* (i.e., EEP, an external library, or another ETP), the jar files of these components are also included in the ETP's *build path*. The information about the components *required by* the ETP can be found in the ETP's manifest file. However, information about the appropriate versions of the components is usually missing.

A promising approach to automatic identification of the component versions is *software bertillionage* [34]. To perform the identification bertillionage requires the jar used in compiling the ETP and corpus of all versions of the jar. Unfortunately the ETPs we collected were not bundled with the Eclipse SDK jar files used to compile the ETP, rendering bertillionage inapplicable. Thus, version identification is essentially a manual

[3](Lat.) the latest possible date of an event or an object.

process.

Version identification would require a prohibitive effort if all versions of an ETP were to be considered: indeed, each version of the ETP might require a completely different set of versions of the components. Thus, we focus only on one version of the ETP in each release year. Given the ETP p and a release year y, the manual version identification process follows the steps presented in Fig. 5.4.

After initialization, we identify appropriate versions of the *other components* (Step 2). If no such versions of the *other components* exist we consider the previous version of p, and if there is none, we exclude p from consideration. Once the appropriate versions of the *other components* have been identified, we check for source compatibility of the current version of p with each of the Eclipse SDK releases (Step 4). We stress that when checking the compatibility with another Eclipse SDK release, the versions of *other components* are kept unchanged in the build path. The rationale for this decision is twofold: 1) we are only interested in the relationship between ETP and ECP interfaces and 2) since the identification of the errors is done manually, only errors related to incompatibilities between the ETP under study and the ECP interfaces will appear in the Eclipse console.

We illustrate the version identification process with an example of an ETP, googlipse 0.5.4. In addition to the ECP dependencies, googlipse depends on two EEPs, J2EE Standard Tools (JST) and Web Standard Tools (WST). Since googlipse 0.5.4 was released in 2007, the versions of the jar files added to the *build path* of googlipse are Eclipse SDK 3.3, WST R-2.0 and JST R-2.0, all released in 2007. When the project is built, 29 errors are reported in the Eclipse console. When we trace the errors in the source code of googlipse, we find that the errors result from an unresolved class dependency from an ECP. This indicates that googlipse 0.5.4 is incompatible with the Eclipse SDK 3.3. The jar files from Eclipse SDK 3.3 are removed and replaced by the jar files of Eclipse SDK 3.2. The SDK replacement removes all errors from the Eclipse console. The same procedure would have been followed if any of EEP dependencies had caused errors. Hence, the compatible versions of the jar files required by googlipse 0.5.4 are found in WST R-2.0 and JST R-2.0. These versions of the jar files are used to determine source code compatibility of googlipse 0.5.4 with different SDK releases.

In addition to swapping the components an ETP depends on, we also had to select the appropriate Java JDK that is compatible with the ETP. Since the ETPs' development dates back as far as the year 2003, the Java JDKs that were used on the different ETPs range from 1.3 to 1.6.

We call an ETP source compatible with a given Eclipse SDK release if no compilation errors are reported, and source incompatible if at least one error related to the *compulsory dependencies* is reported.

5.6 Quantitative analysis: *good ETPs* and *bad ETPs*

In this section, we quantitatively analyse the compatibility of the *good ETPs* vs *bad ETPs*.

5.6.1 Results

Table 5.3 and Figure 5.5 present the results of the source compatibility experiments. The numbers of *good ETPs* and *bad ETPs* considered in the experiments are those in the *Total* rows in the groups of *I (good)* and *II (bad)* of Table 5.1. For example, consider the data from Table 5.3 for *good ETPs* released in 2003. 32 *good ETPs* are source compatible (**SC**)

Input: ETP p, release year y

1. Initialization:
   (a) Select the latest version of p in a given release year y.
   (b) For every *other component* add its latest version released in y to the build path of p.
   (c) Add the SDK released in y to the build path of p.
2. Compile the current version of p with respect to the current version of the SDK (including current versions of the *other components*)
   (a) If compilation was successful, go to Step 4;
   (b) Identify the source of compilation errors:
      i. Errors due to a direct dependency on an *other component*: replace in the build path the versions of the components involved by the preceding ones.
      ii. Errors due to a direct dependency on the SDK: replace the SDK by the one released in the preceding year.
      iii. Go to Step 2.
3. If there is another version of p released in y,
   (a) take the preceding version of p, go to Step 2.
   (b) otherwise, exclude p from consideration.
4. For Eclipse SDK release r (1.0 to 3.7) repeat
   (a) Compile p.
   (b) If compilation was successful, report "success(p,r)".
   (c) Identify the source of compilation errors:
      i. Direct dependency on the SDK: report "failure(p,r)".
      ii. Indirect dependency only:
         A. and there is another version of p released in y, take the preceding version of p, go to Step 2.
         B. otherwise, exclude p from consideration.

Figure 5.4: Manual version identification: tracing errors to their source (Steps 2b and 4c) is a manual process.

		SDK releases											SC	SIC
		1.0	2.0	2.1	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7		
		2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011		
ETP-APIs released in	2003	4	19	*31*	27	27	27	27	27	27	27	27	**32**	3
	2004	2	6	12	*39*	39	39	38	38	38	38	38	**40**	3
	2005	0	1	2	34	*41*	41	41	41	41	41	41	**41**	7
	2006	0	1	2	29	58	*67*	67	67	67	67	67	**67**	9
	2007	0	0	1	13	22	41	*46*	46	46	46	46	**47**	4
	2008	1	1	2	10	16	34	50	*55*	55	55	55	**55**	3
	2009	0	0	0	3	5	12	30	34	*35*	35	35	**35**	2
	2010	0	0	0	2	6	11	22	27	28	*28*	28	**28**	2
bad ETPs released in	2003	2	9	*27*	15	8	7	6	6	6	6	6	**28**	5
	2004	1	4	10	*43*	28	21	21	20	19	19	19	**44**	9
	2005	1	1	2	21	*44*	23	22	22	22	22	21	**45**	3
	2006	0	0	1	7	27	*40*	25	21	19	18	18	**43**	4
	2007	0	0	1	6	16	28	*32*	24	21	20	20	**38**	7
	2008	0	0	0	1	7	19	27	*33*	30	30	28	**36**	4
	2009	0	0	0	0	1	7	19	23	*25*	23	23	**25**	5
	2010	0	0	0	1	5	7	10	18	21	*19*	19	**23**	1

Table 5.3: The number of good ETPs (above) and bad ETPs (below), source compatible with the Eclipse SDK releases. The diagonal cells (shaded gray) show the number of ETPs that are source compatible with the Eclipse SDK released in the same year as the ETPs. The upper and lower triangles correspond to forward and backward source compatibility, respectively.

at least once with all the components they depend on, 4 of the 32 are source compatible with Eclipse SDK 1.0, 19 of the 32 are source compatible with Eclipse SDK 2.0 and 31 of 32 are source compatible with Eclipse SDK 2.1.

For each ETP release year, the bars in Figure 5.5 corresponding to the Eclipse SDK releases (shown in the *legend* of the graphs) show the percentage of source compatible ETPs with respect to the total number of source compatible ETPs (column SC in Table 5.3). For the sake of illustration we choose two ETP release dates, 2004 and 2006; bar charts corresponding to other release years have similar shapes.

Starting with the Eclipse SDK release corresponding to the release year of the ETPs (starred bar), for each year, we notice that most *good ETPs* are forward source compatible but not backward source compatible. In contrast, *bad ETPs* are neither forward source compatible, nor backward source compatible with the Eclipse SDK.

5.6.2 Discussion: *good ETPs* and *bad ETPs*

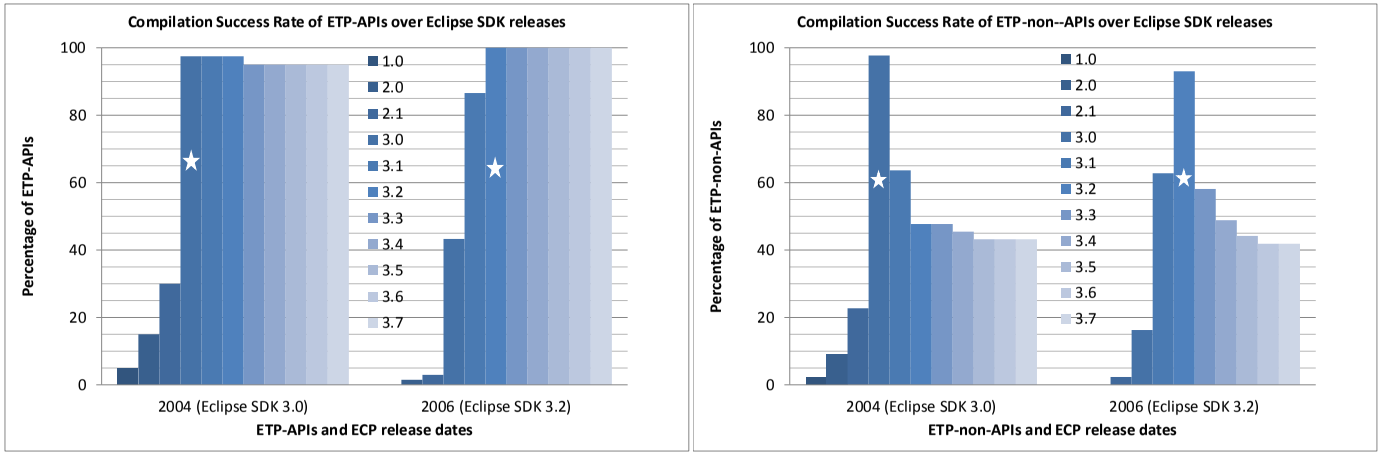
In this section we discuss the results of source compatibility of the *good ETPs* and *bad ETPs*.

5.6.2.1 Good ETPs

According to the *Provisional API guidelines* [5], ETPs that follow the guidelines are not supposed to fail in the subsequent Eclipse SDK releases. In addition, the *Version Numbering Document* [11], describing the guidelines of how to evolve Eclipse SDK versions in the subsequent releases states that Eclipse SDK version numbers are composed of two integers named major.minor: the major segment indicates *breaking change* in the API, the minor segment indicates *non-breaking change* in the API ("externally visible" changes). A breaking change violates both the ETP's binary and source compatibility (cf. Section 5.5). For our study this means that *good ETPs* should not fail in the subsequent Eclipse SDK releases that do not involve breaking changes. To check whether this guideline is indeed being adhered to we take a closer look at Table 5.3.

By inspecting Table 5.3, we observe that four *good ETPs* released in 2003 failed from Eclipse release-3.0 to release-3.7 (i.e., there were 31 source compatible ETPs in cell (2003,21) and 27 source compatible ETPs in cell (2010,3.7)). This observation does not contradict the guideline. Indeed, Eclipse SDK 2.1 was released in 2003, i.e., the ETPs released in 2003 were based on an Eclipse SDK release not later than 2.1. According to the version numbering, there were breaking changes between Eclipse SDK 2.1 and Eclipse SDK 3.0. Hence following the guideline some *good ETPs* might have become source incompatible, and this indeed happened for four *good ETPs*.

One ETP-API, OracleExplorer, released in 2004 failed from Eclipse SDK 3.3 throughout to 3.7. From Eclipse SDK 3.2 to Eclipse SDK 3.3, the Eclipse version numbering does not indicate breaking changes and, therefore, the failure of OracleExplorer indicates a guideline violation. Since Eclipse SDK 3.3, org.eclipse.ui.part.MultiPageEditorPart, extended by one of the OracleExplorer classes, contains a final method setActiveEditor(IEditorPart targetEditor). This method has the same signature as setActiveEditor(IEditorPart editorPart), one of the methods of OracleExplorer itself. Therefore a conflict is introduced as a final method cannot be overridden.



(a) good ETPs

(b) bad ETPs

Figure 5.5: Backward and forward source compatibility success rate of the ETPs released in 2004 and 2006 with all the releases of Eclipse SDK considered.

5.6.2.2 Bad ETPs

In Table 5.3, looking at the source compatibility success rate of the *bad ETPs*, we observe that the forward source compatibility improves every year for ETPs released in 2007 to 2010. On further investigation of the possible cause of this phenomenon, we discovered that the majority of non-APIs used by these ETPs were introduced in the early Eclipse SDK releases. For example, we found out that the 19 ETPs classified in release year 2010 altogether used a total of 127 non-APIs of which 28% were introduced in Eclipse SDK release-1.0, 30% in Eclipse SDK 2.0 and none of the newly introduced non-APIs in Eclipse SDK 3.5 and 3.6 was used by any of the ETPs. This could mean that older non-APIs are relatively stable, and this is why the ETPs that use them do not fail. This conjecture has been formally studied in Chapter 6.

5.6.2.3 Good ETPs vs Bad ETPs

Comparing Figure 5.5-(a) and Figure 5.5-(b), it becomes apparent that *good ETPs* have a very high source compatibility success rate compared to the *bad ETPs*. This can be expected, since *bad ETPs* use unstable non-APIs and hence are more likely to be affected by changes in Eclipse. For *backward* source compatibility we look at the source compatibility success rates from the Eclipse SDK released in the same year as the ETPs back to Eclipse SDK 1.0. We observe that for both *good ETPs* and *bad ETPs*, the percentage of source compatible ETPs rapidly decreases. We also observe that the decrease rate is much sharper for *bad ETPs* compared to *good ETPs*. For *forward* source compatibility we look at the source compatibility success rates from the Eclipse SDK released in the same year as the ETP up to the most recent Eclipse SDK release considered (3.7). *bad ETPs* have a very high failure rate compared to *good ETPs*.

5.6.3 Quantifying the Survival of ETPs: good ETPs vs bad ETPs

While Section 5.6.2 informally discussed differences in the forward source compatibility success rate between *good ETPs* and *bad ETPs*, in this section we augment the preceding discussion with a formal statistical study. Recall that at the beginning of Section 5.5 we stated a general definition of survival of an ETP developed on top of a given *major Eclipse SDK release* as the number of subsequent major Eclipse SDK releases that it can successfully compile with (forward source compatibility).

Because we lack the information about the specific Eclipse SDK releases that were used when developing the ETPs, in this section we redefine *survival* of an ETP. Since the release year of the ETP can be used to determine *terminus ante quem* of the Eclipse SDK release, we define survival of an ETP as the number of major Eclipse SDK releases that the ETP can successfully compile with, starting with the Eclipse SDK release in the year after the release of the ETP (i.e., the count of Eclipse SDK releases after the release corresponding to the cell in the shaded diagonal in Table 5.3). *Failure* of an ETP is the opposite of *survival*, i.e., the number of major Eclipse SDK releases that an ETP fails to compile with, starting with the Eclipse SDK release in the year after the release of the ETP. In the data-set all ETPs that were source compatible with one Eclipse SDK release and failed in the next Eclipse SDK release also failed with all the Eclipse SDK releases released later.

Table 5.4 is the contingency table of survivals and failures for *good ETPs* and *bad ETPs* based on Table 5.3. Observe that although the number of ETPs that failed are not shown in Table 5.3 it can easily be computed. For example, of the *bad ETPs* released in

	Compilation Response		
	Failure (1)	**Survival (2)**	**Total**
bad ETPs	672	678	1,350
	49.8%	50.2%	100.0%
good ETPs	52	1,512	1,564
	3.3%	96.7%	100.0%
Total	724	2,190	2,914
	24.8%	75.2%	100.0%

Table 5.4: ETPs forward-source-compatibility contingency table based on the total compilation responses

2003 in Table 5.3, a total of 28 ETPs were source compatible. With respect to Eclipse SDK 3.0 we have a survival of 15 ETPs in cell (2003,3.0). The failure with respect to Eclipse SDK 3.0 is of 13 ETPs (difference between the total, 28, and the survival, 15).

We compare the *overall* survival between *good ETPs* and *bad ETPs*. First, we test the independency of whether an ETP is source compatible of it being a good ETP or a bad ETP. Second, we test the impact of non-APIs in ETPs on the forward compatibility success rate. All statistical calculations have been carried using popular statistical software R [87].

Independence. We test the following hypotheses:

- H_0^i: *The compatibility success rate is independent on whether an ETP is an good or bad ETP*;
- H_a^i: *The compatibility success is dependent on whether an ETP is an good or bad ETP.*

For the independence test, the results of both the χ^2 test and Fisher's exact test lead to the p-value $< 2.2 \times 10^{-16}$. Hence, we can guarantee statistical significance on any reasonable threshold, confidently reject H_0^i and claim that the compatibility success is dependent on whether an ETP is a good or bad ETP.

Impact. Based on the Eclipse guideline [37] we expect *good ETPs* have higher forward compatibility success rate:

- H_0^c: *good ETPs and bad ETPs represent two populations with equal median values for forward compatibility success*;
- H_a^c: *good ETPs represents a population with higher forward compatibility success median value than bad ETPs.*

To test the hypotheses, we imposed an ordinal scale on compatibility failure (1) and success (2) as suggested in [10], and performed the Wilcoxon test. The test statistic equals 1546104 and the p-value $< 2.2 \times 10^{-16}$. As above, we can guarantee statistical significance on any reasonable threshold, confidently reject H_0^c and claim that *good ETPs* have higher forward compatibility success than *bad ETPs*.

5.7 Qualitative Analysis: Classification III—V ETPs

As opposed to the quantitative analysis of *good ETPs* and bad ETPs in Sections 5.6, to analyze Classifications III-V ETPs we employ qualitative analysis since the ETPs in these

classifications are too few, i.e., Classification II, 32 ETP, Classification IV, 6 ETPs, and Classification V, 8 ETPs. During the analysis, we investigate possible causes why and how ETPs move from one category to the other, i.e., between "good" and "bad".

The following metrics will be used to analyze the ETPs in Section 5.7.1–5.7.3. We use the metrics *NOF* and *NFD* to give us an indication of the ETP we are analyzing. The *Date* metric to give us an indication of the possible SDK on top of which the version was built. The functionality metrics *NnP* and *Int* to give us an indication of the functionality from Eclipse used by a version of an ETP. The metric *NnP* is our main focus in the analysis. Finally, the *Compatibility* metric help us to trace the non-APIs that are problematic in new SDK releases.

- Ver: Version name of the ETP.
- Date: Date of release on SourceForge of a version of an ETP (represented the format *mm/yy*). With respect to the release dates of the SDK, the metric will give us an indication of the possible SDK on top of which the version was built.
- NnP: Number of unique non-APIs used by a given version of an ETP.
- NOF: Number of `.java` files in a given version of an ETP.
- NFD: Number of `.java` files in a given version of an ETP having dependencies on non-APIs. Since we compile the ETPs, we deleted all the unused imports from the source code.
- Int: The unique number of interfaces (good + bad) from Eclipse used by a given version of an ETP.
- Compatibility: This indicates the source compatibility of a given version on an ETP with the SDK releases (0–incompatibility and 1–compatibility).

5.7.1 Classification III—good–*bad ETPs*

In this section we qualitatively analyze the 32 ETPs in Classification III in Table 5.1. Recall that for all the ETPs in this classification, earlier versions dependent solely on APIs (good) and latter versions depending on a non-API (bad). We investigate possible reasons why ETPs change from the "good" category to the "bad" category.

For each ETP, we extracted and analysed the metrics defined in Section 5.7. Table 5.5 and 5.6 present the metrics for two ETPs *Parfumball* and *Eclipse Corba*, respectively. The remaining 30 ETPs show similar trends.

For the versions of the ETPs that depend on only APIs (*good part*), we observe similar findings as those we observed for Classification I ETPs (*good ETPs*). For the versions of ETPs that have a dependency on non-APIs (*bad part*), we observe similar findings as those we observed in Classification II ETPs (*bad ETPs*).

For the *good part*, we observed that versions of ETPs that are compatible with a given SDK release continue to be compatible with later releases of the SDK. For the *bad part*, we observed that some of the versions of the ETPs are not source compatible in new SDK releases and those that continue to be compatible we observe that they use old non-APIs.

The reason why ETPs move from good to bad could be related to the Lehmans' law of continuous growth [68]. The law states that the functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime. We observe that

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.0.1	11/04	52	0	51	0	1	0	0	0	0	0	0	0
0.0.2	11/04	53	0	68	0	1	0	0	0	0	0	0	0
0.0.3	11/04	59	0	71	0	1	1	1	1	1	1	1	1
0.0.4	06/05	70	1	108	1	1	1	0	0	0	0	0	0

Table 5.5: Parfumball. Compatibility with SDKs, 0–Compatible and 1–Incompatible.

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.2.0	07/04	79	0	446	0	1	1	1	1	1	1	1	1
0.3.0	02/05	83	0	462	0	1	1	1	1	1	1	1	1
0.3.1	04/05	91	0	483	0	1	1	1	1	1	1	1	1
0.3.2	05/05	92	0	448	0	0	1	1	1	1	1	1	1
0.4.0	05/06	104	0	454	0	0	0	1	1	1	1	1	1
0.4.1	05/06	104	0	454	0	0	0	1	1	1	1	1	1
0.5.5	02/07	133	1	613	1	0	0	1	1	1	1	1	1
0.5.6	07/07	144	1	619	2	0	0	0	1	1	1	1	1
0.6.0	01/08	147	1	554	1	0	0	0	1	1	1	1	1
0.6.1	02/08	147	1	608	1	0	0	0	1	1	1	1	1
0.7.0	09/08	164	1	648	1	0	0	0	1	1	1	1	1

Table 5.6: Eclipse Corba

as the ETPs evolve, in accordance to the law of continuous growth, there is an increasing trend in the two metrics *Int* and *NOF* in Tables 5.5 and 5.6, respectively. Since the ETPs consume more functionality from the SDKs as they evolve, we conjecture that some of the functionality they require may be absent from the APIs of the SDK (this conjecture is verified in Chapter 7).

5.7.2 Classification IV—bad–good ETPs

In this section, we analyze the six ETPs in Classification IV in Table 5.1. Recall that for all the ETPs in this classification, earlier versions dependent on a non-API (bad) and latter versions depending solely on APIs (good). In the analysis, we investigate possible reasons why and how ETPs change from "bad" category to the "good" category. We investigate why and how the non-APIs are eliminated from the source code of the ETP and how the removed non-API affects the ETP before and after removal.

Tables 5.7–5.11 present the results of analysis for the ETPs in Classification IV (good to bad)[4]. Since the ETPs are few and also have very few versions, the tables present analysis for all the versions.

5.7.2.1 Strutsbox

Strutsbox is a visual Eclipse plugin toolkit for developing applications with Jakarta Struts Framework. Strutsbox is eight years old having a total of eight versions with its latest

[4]One of the ETPs, ShellEd, had one unused non-API import in its early versions. This ETP now belongs Classification II ETPs. Five of the remaining ETPs are analyzed below.

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.0.1	07/04	200	4	220	4	1	0	0	0	0	0	0	0
1.0.2	01/05	205	4	224	4	1	0	0	0	0	0	0	0
1.0.3	05/05	207	4	279	4	1	1	0	0	0	0	0	0
1.0.4	06/05	207	3	290	3	1	1	0	0	0	0	0	0
1.1.0	08/05	192	3	252	3	1	1	0	0	0	0	0	0
1.1.1	10/05	156	2	247	2	1	1	0	0	0	0	0	0
2.0.0	01/06	146	0	235	0	0	1	1	1	1	1	1	1
2.0.1	02/06	147	0	242	0	0	1	1	1	1	1	1	1

Table 5.7: Strutsbox

version released in 2006.

Table 5.7 presents the analysis results of the ETP Strutsbox. In version 1.0.1 and 1.0.2 in column *Ver*, the class `de.strutsbox.ui.editor.parts.MessageDialogWithToggle` calls a method `org.eclipse.ui.internal.WorkbenchMessages.getString`. From SDK 3.0 to SDK 3.1, the method `org.eclipse.ui.internal.WorkbenchMessages.getString` was removed from Eclipse. As can be observed from Table 5.7, version 1.0.1 and 1.0.2 fail with SDK 3.1 onwards.

In version 1.0.3, the result of the method `org.eclipse.ui.internal.WorkbenchMessages.getString` was replaced by an empty string (`""`). The non-API `org.eclipse.ui.internal.WorkbenchMessages` was removed from version 1.0.3 and as can be observed it is compatible with SDK 3.1.

In version 1.1.0, the class `de.strutsbox.ui.editor.parts.StatusDialog` calls the method `org.eclipse.ui.internal.MessageLine.setErrorStatus`. In version 1.1.1, the class `de.strutsbox.ui.editor.parts.StatusDialog` was deleted from the plug-in. This is the reason the number of non-APIs reduces from 3 to 2.

In version 2.0.0, there was a major restructuring where the packages `de.strutsbox.ui.editor`, `de.strutsbox.ui.wizards` and `de.strutsbox.visualizer.ui`, were merged in version 2.0.1 into one package `de.strutsbox.ui`. All the non-APIs were eliminated in the plug-in during the restructuring. We can observe that the two versions are compatible with the SDK releases from SDK 3.1.

5.7.2.2 Eclipse Coding Tools

Eclipse Coding Tools is an Eclipse plug-in that adds some small refactoring tools, aimed at fixing logging code. The plug-in is eight years old, having 11 versions with the latest version released in 2010.

In Table 5.8 column *Ver*, version 0.3.0 of the ETP accesses the value of the field `org.eclipse.jdt.internal.ui.text.IJavaPartitions.JAVAPARTITIONING`. In SDK 3.1, the non-API `org.eclipse.jdt.internal.ui.text.IJavaPartitions` was removed from Eclipse. We can observe the decrease in the number of non-APIs used by the ETP from 15 to 14. The removal of the non-API is one of the causes of compatibility problems between version 0.3.0 of the ETP with SDK 3.1–3.7. Version 0.4.0 of the ETP replaced `org.eclipse.jdt.internal.ui.text.IJavaPartitions.JAVA_PARTITIONING` with "`___java_partitioning`". "`___java_partitioning`" is the value that is assigned to the accessed field in version 0.3.0 of the ETP (i.e.,

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.2.0	10/04	232	15	138	9	1	0	0	0	0	0	0	0
0.3.0	10/04	231	15	142	9	1	0	0	0	0	0	0	0
0.4.0	03/05	229	14	115	9	1	0	0	0	0	0	0	0
0.5.0	12/05	223	14	202	8	1	1	1	1	0	0	0	0
0.6.0	05/10	164	0	172	0	0	1	1	1	1	1	1	1
1.0.0	05/10	70	0	44	0	0	0	0	1	1	1	1	1
1.0.1	05/10	70	0	44	0	0	0	0	1	1	1	1	1
1.0.2	05/10	69	0	44	0	0	0	0	1	1	1	1	1
1.0.3	05/10	69	0	44	0	0	0	0	1	1	1	1	1

Table 5.8: Eclipse Coding Tools

copied from the non-API in SDK 3.0).

One of the non-APIs `org.eclipse.jdt.internal.ui.dialogs.TypeSelectionDialog` that is used by the ETP was removed in SDK 3.1. In version 0.5.0, all the source code parts that use the removed non-API are replaced by the services of another non-API `org.eclipse.jdt.internal.ui.dialogs.TypeSelectionDialog2`. Version 0.5.0 continued to be compatible with the successive SDK releases until SDK 3.4 because the method `org.eclipse.jdt.internal.corext.util.JavaModelUtil.getFullyQualifiedName`, called by version 0.5.0 of the ETP, was deleted.

In version 0.6.0 all the non-APIs disappeared from the ETP as a result of a major restructuring where the package `awilkins.eclipse.coding.templates` including its sub-packages were removed from the ETP. This package contained the classes that used non-APIs from Eclipse. In version 1.0.0 there was another major restructuring where the package `awilkins.objectmodel` was removed from the ETP, the reason we can observe the decrease in the NOF. Version 1.0.0–1.0.3 use APIs that were introduced in SDK 3.3 the reason they are not compatible with SDK 3.2 and below.

5.7.2.3 EuroMath2

EuroMath2 is an Eclipse plug-in that provides a platform for editors editing various XML files with multiple namespaces and also able to manage editors with WYSIWYG capability. The plug-in is eight year old having the a total of 7 versions. The latest version was released in the year 2006.

As can be see in Table 5.9, the ETP had one dependency on a non-API from Eclipse in versions 1.1.9 to 1.2.1. This non-API did not cause incompatibility with the different SDKs. From version 1.3.0 onwards, the class with the non-API was deleted from the ETP. The sharp drop of the NOF from version 1.3.3 to 1.4.0 is not interesting for this study since these versions do not have dependency on non-APIs.

5.7.2.4 Emonic

Emonic (Eclipse-Mono-Integration) is a Eclipse-Plugin for C#. It provides color highlighting, outline, word-completion and build mechanism via Ant or Nant. The plug-in is 6 years old having a total of six versions. The latest version of the plug-in was released 2010.

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.1.9	09/04	141	1	695	3	1	1	1	1	1	1	1	1
1.2.0	12/04	154	1	775	3	1	1	1	1	1	1	1	1
1.2.1	04/05	132	1	623	1	1	1	1	1	1	1	1	1
1.3.0	10/05	154	0	630	0	1	1	1	1	1	1	1	1
1.3.1	11/05	153	0	630	0	0	1	1	1	1	1	1	1
1.4.0a	08/06	150	0	349	0	0	0	1	1	1	1	1	1
1.4.0	09/06	156	0	344	0	0	0	1	1	1	1	1	1

Table 5.9: EuroMath2

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.1.0	01/06	177	4	56	3	1	1	1	0	0	0	0	0
0.1.1	08/06	163	2	54	2	1	1	1	0	0	0	0	0
0.1.2	03/07	183	4	64	4	1	1	1	0	0	0	0	0
0.1.3	10/07	188	2	66	1	1	1	1	1	1	1	1	1
0.3.0	10/07	315	0	128	0	0	0	1	1	1	1	1	1
0.4.0	01/10	414	0	325	0	0	0	1	1	1	1	1	1

Table 5.10: Emonic

From Table 5.10 column *Ver*, we can observe a decrease in the number of non-APIs from version 0.1.0 to 0.1.1 and again an increase from version 0.1.1 to 0.1.2. In version 0.1.1 the method `org.emonic.base.editors.CSharpEditor.editorContextMenuAboutToShow` that calls the non-API method `org.eclipse.ui.internal.ViewerActionBuilder.readViewerContributions` in version 0.1.0 was deleted. The same method with the same implementation was later reintroduced in version 0.1.2. The package `org.emonic.base.actions`, in version 0.1.0 that calls the non-API method `org.eclipse.jdt.internal.ui.packageview.PackageExplorerPart.getTreeViewer` was deleted in version 0.1.1. The same package was reintroduced in version 0.1.2.

From Table 5.10 column *Ver*, we can see a decrease on non-APIs from version 0.1.2 to 0.1.3. Version 0.1.2 calls the non-API constructor `org.eclipse.ui.internal.dialogs.ViewSorter`. The non-API `org.eclipse.ui.internal.dialogs.ViewSorter` was deleted in SDK 3.3. This caused version 0.1.2 to fail in SDK 3.3 as indicated in Table 5.10. The failure was fixed in version 0.1.3 by getting rid of the services of the non-API `org.eclipse.ui.internal.dialogs.ViewSorter` and building its own API `org.emonic.base.views.OutlineViewerSorter`. In version 0.1.3, the ETP replaced the services of the non-API `org.eclipse.jdt.internal.ui.packageview.PackageExplorerPart` with the services of the API `org.eclipse.jdt.ui.IPackagesViewPart`.

From version 0.1.3 to 0.3.0, two non-APIs are removed the ETP. Version 0.1.3 calls two methods from the non-API `org.eclipse.ui.internal.Workbench` as a workaround for Eclipse bug 75440 stated as a comment in the ETP's source code. Bug 75440 was reported on 10/2004 and fixed on 04/2005. In version 0.3.0, the ETP no longer uses the services in the non-API `org.eclipse.ui.internal.Workbench`. Furthermore, version

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.3.6	07/05	168	1	142	2	1	1	1	1	1	1	1	1
1.3.7	10/09	177	1	152	1	0	1	1	1	1	1	1	1
1.3.8	07/10	177	0	152	0	0	0	1	1	1	1	1	1

Table 5.11: Eclipse Metrics

0.1.3 uses the services of the non-API `org.eclipse.ui.internal.ViewerActionBuilder` in the method `org.emonic.base.editors.CSharpEditor.editorContextMenuAboutToShow`. The same method in version 0.3.0 uses the services from API `org.eclipse.jface.action.Separator` but has different implementation.

5.7.2.5 Eclipse Metrics

Eclipse Metrics is an Eclipse plug-in that provides metrics calculation and a dependency analyzer that detects cycles in package and type dependencies. The ETP has three versions on SourceForge al released between 2008 and 2009.

In Table 5.11, the ETP was calling the non-API method `org.eclipse.jdt.internal.core.Openable.getUnderlyingResource` in the first two versions and in the last version replaced the non-API method with the API method `org.eclipse.jdt.core.IJavaElement.getUnderlyingResource`.

5.7.2.6 Discussion

From the analysis in Section 5.7.2.1—5.7.2.5 we can learn a number of lessons. First, when developers stop using the functionality of a given non-API, we have noticed three things:

1. Developers build their own API that have same functionality as the non-API (copy & paste). This occurred three times in the analysis presented. This way they avoid using the unstable non-API between SDK releases. For example, in the *Emonic* ETP the developers replaced functionality of the non-API that was problematic in version 0.1.2 with their own build API in version 0.1.3. From an interview with some ETP developers, re-implemented (copy & paste) code of the non-APIs can be difficult to maintain. Furthermore, the developers told us that copied & pasted code misses out from benefiting for improvements in the non-APIs in new SDK releases.

2. Developers find similar functionality offered by an API in the SDK. This occurred two times in our analysis. For example, in the *Eclipse metrics* ETP, the developer replaced the functionality of the non-API used in the version 1.3.6 and 1.3.7 with functionality of an API in version 1.3.8.

3. Developers completely eliminate the entities in the ETP source code that uses the functionality from the non-API. This occurred four times in our analysis. For example, in the ETP *EuroMath2* a class that used the non-API one version was deleted in a new version.

Second, we have observed that the reason why the non-APIs are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new SDK release.

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.0.3	08/05	165	3	68	1	0	1	0	0	0	0	0	0
1.1.0	11/05	166	0	69	0	0	0	1	1	1	1	1	1
2.1.1	08/08	166	2	69	7	0	0	0	1	1	1	1	1
2.1.2	08/08	45	0	16	0	0	0	1	1	1	1	1	1
2.1.4	01/09	164	2	71	2	0	0	0	1	1	1	1	1

Table 5.12: ClearCase

5.7.3 Classification V—Oscillating ETPs

In this section we analyze the eight ETPs in Classification V in Table 5.1. Recall that these are ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API. Most of the ETPs in this classification have very many versions. We investigate why the ETPs alternate between the good and bad categories and possible caused of the alternation.

For ETP that alternate from bad–good–bad, we will investigate why they eliminate non-APIs and then reintroduce them. For the ETPs that alternate from good–bad–good, like in Section 5.7.2, we will investigate why they eliminate non-APIs.

Since the the ETPs in this Classification have many versions, the tables as well as in the discussions in this section will present versions where we observe elimination and introduction of non-APIs (reasons for introduction of non-APIs are similar to the ones already mentions in Section 5.7.1–Classification III—good–bad ETPs). Tables 5.12–5.17 present the results of the analysis of six of the eight ETPs in Classification V (oscillating ETPs). One of the ETPs did not have dependencies on Eclipse[5] and in ETP *Eclipse ResourceBundle Editor* we did not observe elimination of non-APIs.

5.7.3.1 ClearCase

ClearCase Eclipse plug-in allow Eclipse users to have access to ClearCase functionality in the Eclipse workbench. The ETP has got 22 versions releases on SourceForge between 2003 and 2010 when we collected this data. Table 5.12 only shows versions where we observed elimination of non-APIs. ClearCase belongs to the bad–good–bad category.

From Table 5.12 we observe the disappearance of the 3 non-APIs from version 1.0.3 to 1.1.0. We also observe that version 1.0.3 is compatible with only SDK 3.1 and version 1.1.0 is compatible with SDK 3.2–3.7. From source code inspection, we observed that in SDK 3.1 a non-API package `org.eclipse.core.internal.resources.mapping` was introduced. Version 1.0.3 started using three classes from this non-API package. In SDK 3.2, the three classes used by version 1.0.3 were graduated to API package `org.eclipse.core.resources.mapping`. Version 1.1.0 now uses these three classes that graduated.

In version 2.1.2 the package `net.sourceforge.eclipseccase.ui`, that exists in the rest of the versions, is missing. The developer did not commit it to the repository. From version 2.1.4 to the latest version 2.2.6 the non-API reappeared since the package `net.sourceforge.eclipseccase.ui` reappeared in the source code.

[5]The ETP *gted* had an unused non-API in one of the intermediate versions. This ETP should be categorized as a good ETP

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.2.0	10/03	57	2	12	1	0	0	0	0	0	0	0	0
1.2.1	12/03	55	0	12	0	1	1	1	1	1	1	1	1
3.2.0	02/06	66	2	30	1	0	1	1	1	1	1	1	1
3.2.1	02/06	65	0	57	0	0	1	1	1	1	1	1	1

Table 5.13: Eclipse Platform Extensions

5.7.3.2 Eclipse Platform Extensions

This is a set of Eclipse plug-ins that provide additional functionality to the Eclipse IDE. The ETP has got 18 versions released on SourceForge between 2003 to 2010. Table 5.13 versions where we observed elimination of non-APIs. The ETP belongs to the bad–good–bad category.

From Table 5.13 we can observe a drop in the number of non-APIs from version 1.2.0 to 1.2.1. Version 1.2.0 uses the non-APIs `org.eclipse.ui.internal.IPreferenceConstants` and `org.eclipse.ui.internal.WorkbenchPlugin`. These non-APIs cause the version to fail in SDK 3.0-3.7 as can be observed from Table 5.13. In version 1.2.1 the non-APIs were eliminated and replaced by two APIs. The implementation of the entities that uses the new APIs in version 1.2.1 is different from the implementation of the same entities that uses the non-APIs in version 1.2.0.

The next three versions from 1.2.0 did not have dependencies on non-APIs. In version 3.2.0, two new non-APIs, `org.eclipse.ui.internal.HeapStatus` and `org.eclipse.ui.internal.util.PrefUtil` were introduced in the source code. From version 3.2.1 to the latest version 3.6.1, the code that uses the two non-APIs was dropped from the source code and the non-APIs were deleted.

5.7.3.3 Eclipse ResourceBundle Editor

This is an Eclipse plug-in for editing Java resource bundles. The ETP lists 26 versions, i.e., six version, 0.5.0 to 0.6.0 supported in SDK 2.x (comment on ETPs' SourceForge page) and 20 versions, 0.1.0 to 0.7.7, supported in SDK 3.x (comment on ETPs' SourceForge page). The versions were released between 2004 to 2007.

Indeed we have confirmed from compiling the six versions supported in 2.x that they are only compatible with SDK 2.0 and 2.1. Version 0.6.0 of the six versions has got a dependency on a non-API.

We have also confirmed that the next 20 versions supported in 3.x are compatible with SDK 3.0 to 3.7 and not with earlier SDKs. 11 of the 20 versions, i.e., 0.1.0 to 0.5.4, do not have a dependency on a non-API. Version 0.6.0 has a dependency on a non-API and the next eight versions, i.e., 0.7.0 to 0.7.7 have dependencies on tow non-APIs.

We have discovered that the six versions supported in SDK 2.x have the same version names and release dates as six of the version supported in SDK 3.x. From our observation, since the first version of the ETP was released in December 2004 and SDK 3.0 was released in June 2004, the ETP was initially designed for SDK 3.x. Later the developers downgraded the six versions, 0.5.0 to 0.6.0, to work with SDK 2.x. This is the reason the ETP is categorised as oscillating. This means that the ETP belongs to the good–bad–good category.

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
1.1.0	10/04	24	1	10	1	1	1	0	0	0	0	0	0
1.1.1	11/05	23	0	10	0	1	1	1	1	1	1	1	1
1.1.2	11/05	28	2	10	2	1	1	1	1	0	0	0	0
1.2.2	09/06	28	2	4	1	0	1	1	1	0	0	0	0
1.3.0	05/08	21	0	3	0	0	0	0	1	1	1	1	1

Table 5.14: JUnitRunner

5.7.3.4 JUnitRunner

This is an Eclipse plug-in that allows the user run/debug Junit test method using context popup menu. The ETP has got nine versions released between 2004 to 2009. We analyze only the versions where we have observed disappearance of non-APIs. The ETP belongs to the bad–good–bad category.

In Table 5.14, we can observe the drop in the number of non-APIs from version 1.1.0 to 1.1.1. Version 1.1.0 calls the non-API method `org.eclipse.jdt.internal.launching.JavaLaunchConfigurationUtils.getMainType`. In SDK 3.2 the non-API `org.eclipse.jdt.internal.launching.JavaLaunchConfigurationUtils` disappeared. This can be observed in Table 5.14 that version 1.1.0 fails from SDK 3.2–3.7. In version 1.1.1, a call to the non-API method was changed to a call to an API method `org.eclipse.jdt.core.ICompilationUnit.getJavaProject`. As can be observed, version 1.1.1 is compatible with all the SDK releases. The next four versions from 1.1.2 to 1.2.2 have a dependency on two non-APIs, `org.eclipse.jdt.internal.junit.launcher.JUnitBaseLaunchConfiguration` and `org.eclipse.jdt.internal.junit.launcher.JUnitLaunchConfiguration`. The ETP accesses two fields `org.eclipse.jdt.internal.junit.launcher.JUnitBaseLaunchConfiguration.TESTNAME_ATTR` and `org.eclipse.jdt.internal.junit.launcher.JUnitLaunchConfiguration.ID_JUNIT_APPLICATION` from the two non-APIs. The two non-APIs used the the ETP were deleted in SDK 3.4. This is the cause of the incompatibilities from SDK 3.4 to 3.7. In version 1.3.0, the methods where the two non-API fields are accessed were deleted from the source code. We can see that version 1.3.0 is compatible with SDK 3.3 to 3.7.

5.7.3.5 Eclipse Verilog editor

This is an Eclipse plug-in that provides Verilog (IEEE-1364) and VHDL language specific code viewer, contents outline, code assist, etc. The ETP has got 22 versions released on SourceForge in 2004 to 2010. The ETP belongs to the good–bad–good category.

No version has a dependency on non-APIs except two intermediate versions 0.5.1 and 0.5.2. As can be observed from Table 5.15, there is one non-API in version 0.5.2 and this non-API disappears in version 0.6.0. Version 0.5.2 calls the non-API method `org.eclipse.ui.internal.ide.actions.BuildUtilities.saveEditors`. In version 0.6.0 the call to the non-API method was replaced to a call a `private` method `checkAndSaveEditors` that was newly introduced in version 0.6.0.

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.5.2	06/07	191	1	74	1	0	1	1	1	1	1	1	1
0.6.0	09/07	227	0	295	0	0	0	0	1	1	1	1	1

Table 5.15: Eclipse Verilog editor

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.3.0	01/06	69	1	22	1	0	1	1	1	1	1	1	1
0.3.1	01/06	65	0	22	0	0	1	1	1	1	1	1	1
1.2.0	12/08	216	15	97	2	0	0	0	0	1	1	0	0

Table 5.16: MoreUnit

5.7.3.6 MoreUnit

MoreUnit is an Eclipse plugin that assists developers in writing more unit tests. MoreUnit ETP has released 21 version on SourceFoge from 2006 to 2010. Table 5.16 only lists versions where we observed elimination of non-APIs. The ETP belongs to the bad–good–bad category.

From Table 5.16 column *Ver*, the first version, 0.2.0 had no dependency on non-APIs. Version 0.3.0 used one non-API `org.eclipse.jdt.internal.core.JavaElement`. However, the non-API used in version 0.3.0 did not cause incompatibilities as can be observed in Table 5.16. The code that used the non-API in version 0.3.0 was commented out in version 0.3.1 and the non-API removed from the source code. The next 10 versions 0.3.1–1.1.4 did not use any non-API and the last versions, 1.2.0–2.2.0 used 15 and more non-APIs. In version 1.2.0 we can observe increased functionality used from Eclipse from the number of total interfaces in column *Int*. Like in Section 5.7.1, the possible reason for the use of non-APIs from version 1.2.0 to 2.2.0 could be that the functionality the developer requires is absent from the good part of Eclipse.

5.7.3.7 PovClipse

PovClipse is an Eclipse editor plugin for Povray (Persistence of Vision Raytracer) scene and include files. The ETP has got 14 versions released on SourceForge from 2006 to 2007. Table 5.17 only lists version of where we observed elimination of non-APIs. The ETP belongs to the good–bad–good category.

The first version (absent from the table) did not use any non-APIs. The next four versions, 0.4.0–0.6.0 used two non-APIs, `org.eclipse.ui.internal.WorkbenchPlugin` and `org.eclipse.ui.internal.about.AboutBundleData`. The eight most

						Compatibility with SDKs							
Ver	Date	Int	NnP	NOF	NFD	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
0.6.0	11/06	222	2	325	1	0	1	1	1	1	1	1	1
0.6.1	12/06	249	0	341	0	0	1	1	1	1	1	1	1

Table 5.17: PovClipse

recent versions from 0.6.1–1.1.0 did not have a dependency on a non-API. The functionality of the non-APIs in version 0.4.0–0.6.0 was replaced by functionality of APIs `org.eclipse.update.configuration.IConfiguredSite`, `org.eclipse.update.configuration.IInstallConfiguration` and `org.eclipse.update.configuration.ILocalSite` with a different implementation of these functionalities.

5.7.3.8 Discussion

From the analysis in Sections 5.7.3.1 to 5.7.3.7, we observe a number things: First we have observed similar findings to those reported in Section 5.7.2.6 for the ETP in Classification IV (bad–*good ETPs*). Second, we have observed that some non-APIs (classes and interfaces) do graduate to APIs in a new SDK release, i.e., they are removed from the package with substring `internal` in the fully qualified package name to one without the substring `internal`. For example we discovered this while analyzing the ETP *Clearcase* in Section 5.7.3.1, three non-APIs in the package `org.eclipse.core.internal.resources.mapping` graduated to package `org.eclipse.core.resources.mapping`. When this scenario happens, developers just delete the substring `internal` from the fully qualified class or interface name and the code work.

From the above analysis, the reasons for alternation of ETPs between categories of good and bad are as follows: For ETPs that alternate from good–bad–good, it is because the developers eliminate the problematic non-APIs from the source code. For ETPs that alternate from bad-good-bad, developers eliminate problematic non-APIs from the source code and later they introduce new non-APIs possibly because the functionality they require can only be found in those non-APIs.

5.8 Threats to Validity

As any other empirical study, our analysis may have been affected by validity threats. We categorize the possible threats into construct, internal and external validity.

Construct validity seeks agreement between a theoretical concept and a specific measuring device or procedure. In our analysis, construct validity may be threatened by operationalizing the construct "survival" into our measurement instruments, i.e., release counts and compilation errors. We opt for an "external" operationalization of survival, as it is perceived by the users of the plug-in as opposed to "internal" operationalization of survival as perceived by the plug-in developers (e.g. the level of activity in code development or mailing lists [88]). Internal operationalizations focus on the process leading to (non-)survival of a plug-in, while external operationalizations focus on the result, i.e., (non-)survival itself. Furthermore, construct validity may be threatened by the grouping of the ETPs into *good ETPs* and *bad ETPs*. During the grouping, we relied on the Eclipse naming convention [38] rather than the API guidelines [37]. Noise in our study might have been introduced if software systems deviate from the convention but stick to the guideline. When checking for source compatibility we have considered only one version of an ETP per year and as such our notion of compatibility success might have been too restrictive. Furthermore, for the decrease in the release of new versions of ETPs, it is possible that developers migrated to repositories that have become more popular, e.g., github.

Internal validity is related to validity of the conclusion within the experimental context of the ETP collection considered above. We have paid special attention to the appropriate

use of statistical machinery, e.g. in Section 5.6.3. The ETP survival results could be influenced by confounding factors that we did not take of. For example, the domain of the ETPs could influence the compatibility of the ETPs in new SDK releases.

External validity is the validity of generalizations based on this study. Our results may not be generalizable beyond the specific collection of ETPs since we only considered ETPs from SourceForge, and, therefore, necessarily only open-source ETPs. To ensure that our results can be generalized one has to replicate the study above with respect to ETPs from other open-source repositories as well as commercial ETPs. Going beyond Eclipse we realize that the same study needs to be carried out on a different plug-in framework.

5.9 Related Work

This work presented in this chapter complements our in Chapters 2 and 4. In Chapter 2, we investigated the constrained evolution of 21 carefully selected ETPs on Lehman's software evolution laws. Specifically, we investigated the evolution of ETPs with respect to the Eclipse SDK interfaces they use. During the study in Chapter 2, we did not distinguish the ETPs from the *good ETPs* and the *bad ETPs*. In Chapter 4, we investigated the Eclipse API usage by 512 ETPs, and proposed the distinction between *good ETPs* and *bad ETPs*. The current study is one of the follow-up studies proposed in Chapter 4 where survival of the *good ETPs* and the *bad ETPs* in terms of the source compatibility of these ETPs in the forthcoming SDK releases..

Besides our own related work, the current study is related to two categories of existing studies: release of new versions of applications and effect of API changes on survivability of framework-based applications.

5.9.1 Release of new versions of applications

Rainer and Gale [88] studied Sourceforge.net and concluded that the majority of the projects hosted there should be considered as failures, which, according to them, is the absence of activity in code development, mailing lists and bug reporting. Weiss [108] defined success of a project in correlation with its popularity on the web. English and Schweik [95] classifies project into success categories according to the number of its releases and the time between these releases. Majority of projects hosted at SourceForge have been observed to be abandoned or in an early stage [95, 108]. As opposed to [88, 95, 108], we studied projects from a specific domain, Eclipse plug-ins. Similarly to [95, 108] we observe that majority of projects get abandoned on the early stages and refine this observation by relating abandonment to presence or absence of non-APIs.

In [16] authors examined large repositories of open source projects: Sourceforge.net, KDE, GNOME, Rubyforge and Savannah. They have observed that repositories like SourceForge, Savannah and Rubyforge with more "relaxed" preconditions in hosting a project, contain projects with low activity. The authors showed that when a project enters a more "controlled" repository, like the KDE or GNOME, it has better chances to become successful, with success defined as growth in size and attention gained from developers.

5.9.2 Effect of API changes on survivability of framework-based applications

Dig and Johnson [40] state that to better understand the requirements for API migration tools of evolving frameworks, one needs to understand API changes. To that end, the authors studied API changes in new versions of one proprietary and three open-source frameworks and one library. In all the studied systems, the authors discovered that over 80% of the API-breaking changes are structural, behavior-preserving transformations (refactorings). The implications of the authors' findings confirm that refactoring plays an important role in the evolution of components. Migration tools should focus on support to integrate into applications those refactorings performed in the framework. In comparison to our study, we investigate the impact of API changes in the framework on applications that depend on them.

Other studies related to our work are based on tool support that guides application developers in adapting to API changes in the evolving framework. Nguyen et al. [81] present a tool, LIBSYNC, that guides developers in adapting API usage code by learning complex API usage adaptation patterns from other clients that already migrated to a new library version as well as from the API usages within the library's test code. The tool can identify changes to API declarations by comparing two library versions, can extract associated API usage skeletons before and after library migration, and can compare the extracted API usage skeletons to recover API usage adaptation patterns. Dagenais and Robillard [33] present a tool, SemDiff, that suggests adaptations to client programs by analyzing how a framework adapts to its own changes. Using a case study of Eclipse JDT framework and three client programs, SemDiff recommends relevant adaptive changes and detects non-trivial changes. Wu et al. [112] present AURA that combines call dependency and text similarity analyzes to identify change rules for one-replaced-by-many and many-replaced-by-one methods in a framework. Unlike this line of research, we currently investigate the effects of API changes on survival of framework-based applications. Developing a tool based on the current findings is considered as future work.

5.10 Conclusion and Future Work

In this study, we investigated notion of survival of Eclipse third-party plug-ins (ETPs) related to source compatibility of the plug-ins with Eclipse SDK releases. Understanding this notion of survival is essential both for the users of the ETPs and for their maintainers, it is complicated by the fact that while some SDK interfaces, *APIs*, are stable and supported, other SDK interfaces, *non-APIs*, are unsupported and are subject to arbitrary change or removal without notice. Therefore, we compared the trend followed by ETPs that depend on *only* stable and supported Eclipse APIs (*good ETPs*) and that followed by ETPs that depend on at least one of the unstable, discouraged and unsupported Eclipse non-APIs (*bad ETPs*). In the study, we made the following observations:

1. The majority of ETPs on SourceForge do not produce new versions beyond the first year of release. We also observed that the rate at which new versions are released for *bad ETPs* is higher than that of *good ETPs*.

2. When looking at source compatibility of the ETPs, we observed good ETPs almost never fail in the subsequent Eclipse SDK releases unless these releases involve API-breaking changes.

3. We observed that recently released *bad ETPs* depend more on old non-APIs and less on newly introduced non-APIs. These *bad ETPs* have a very high forward source compatibility success rate.

4. We observed that *good ETPs* have a relatively strong tendency to be forward source compatible compared to *bad ETPs*.

5. As stated by Eclipse, we confirm that APIs are stable over subsequent Eclipse releases that do not involve API-breaking changes. We further confirm that non-APIs are indeed unstable.

6. We also observed that the reason why the non-APIs are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to new SDK release.

7. When eliminating the use of problematic non-APIs in the ETPs source code, we have observed that developers perform one of the following things: i) build their own API that has same functionality as the non-API, ii) find similar functionality offered by an API in the SDK, iii) completely eliminate the entities in the ETP source code that uses the functionality from the non-API and iv) when a non-API matures into an API, developers replace the functionality of the non-API with the new API.

There are several implications of our findings. First, this information provides Eclipse SDK developers with feedback on the current use of APIs and non-APIs in ETPs as opposed to the expected use. The feedback can be used for planning improved services in new releases of SDKs. Second, developers should avoid using the unstable non-APIs as much as possible since they are the major cause of incompatibilities in new SDK releases. Third, developers who must use bad interfaces should use the old ones since we observed that they are more stable. Fourth, ETP developers should avoid re-implemented (copy & paste) code of the non-APIs since it can be difficult to maintain misses and the (copy & paste) code misses out from benefiting from improvements in the non-APIs in new releases of the SDK. Finally, in Chapter 4 we discovered that a large number of developers use non-APIs and in this chapter, we observed that non-APIs influence survival of ETPs. Hence, it is equally important for the framework developers to document both the non-API changes and the API changes.

The results presented so far are interesting and have given us a number of directions for the follow-up work: First, we plan to carry out a survey so as to get first hand information from ETP developers on the factors that impact survival of the ETPs. This survey was carried out and the results are addressed in Chapter 7. Second, one can build a refactoring tool that will aid developers who use non-APIs. One might further refine our distinction between APIs and non-APIs based on usage of specific ECP interfaces. To express the degree of dependence of an ETP on an interface one can apply econometric inequality indices [99, 106] or the Squale model [78]. Finally, one can replicate our work using ETPs from other repositories such as github.

Chapter 6

Compatibility Prediction of ETPs in New Eclipse Releases

Incompatibility between applications developed on top of frameworks with new versions of the frameworks can be a big nightmare to both developers and users of the applications. Understanding the factors that cause incompatibilities is a step to solving them. One way of solving the incompatibilities is to analyze and identify parts of the reusable code of the framework that are prone to change. In this study we carried out an empirical investigation on 11 Eclipse SDK releases (1.0 to 3.7) and 288 Eclipse third-party plug-ins (ETPs) with two main goals: First, to determine the relationship between the age of Eclipse non-APIs (internal implementations) used by an ETP and the compatibility of the ETP. We found that third-party plug-in that use only old non-APIs have a high chance of compatibility success in new SDK releases compared to those that use at least one newly introduced non-API. Second, our goal was to build and test a predictive model for the compatibility of an ETP, supported in a given SDK release in a newer SDK release. Our findings produced 23 statistically significant prediction models having high values of the strength of the relationship between the predictors and the prediction (logistic regression R^2 of up to 0.810). In addition, the results from model testing indicate high values of up to 100% of precision and recall and up to 98% of accuracy of the predictions. Finally, despite the fact that SDK releases with API breaking changes*, i.e., 1.0, 2.0 and 3.0, are only concerned with APIs, our findings reveal that non-APIs introduced in these releases have a significant impact on the compatibility of the ETPs that use them.*

6.1 Introduction

Applications developed on top of frameworks are becoming increasingly popular these days and users of these applications are constantly on the rise [103] (e.g., currently Eclipse marketplace[1] reports over 1,300 Eclipse solutions developed and there are over 1,600,000 installations of these solutions installed directly from Eclipse). Frameworks are constantly

[1] http://marketplace.eclipse.org

evolving to improve on the quality of the functionality they provide to their clients. The applications that want to benefit from the better quality provided by the new release of the framework are subject to compatibility problems [33, 112]. Understanding the factors that cause incompatibilities is a step to solving them. One direction is to identify parts of the reusable code of the framework used by the applications that are prone to change [39].

Eclipse non-APIs are subject to arbitrary change without notice [5, 37]. Eclipse developers discourage the use of these non-APIs and further state that developers can use them at their own risk. However, despite being discouraged, in Chapter 4 we have observed that the use of non-APIs is not uncommon: 44.2% of the Eclipse third-party plug-ins (ETPs) on SourceForge have at least one version that depends on at least one non-API. Moreover, in Chapter 5 we found that the use of non-APIs heavily affects the compatibility of the ETPs with new Eclipse SDK releases.

In this study, by means of source code analysis, we investigate two main goals: 1) the relationship between the age of non-APIs (i.e., the Eclipse SDK release in which the non-API was introduced) used by an ETP and the compatibility of the ETP in new SDK releases. The findings of this investigation are useful to framework-based application developer to estimate the maturity of the non-APIs he/she is using based how they have affected other developers. 2) to build a predictive model for the compatibility of an ETP supported in a given SDK release in a new SDK release. The predictive model can be used by Eclipse-based application developers to assess the chances of their applications' compatibility in a newer SDK release as they prepare for migration towards these releases. Furthermore, it can be used by users of Eclipse-based applications before they update the SDK release where their application is installed to a newer SDK release. The predictions will help the users to predict the compatibility of their applications in the a new release from where they intend to get new updates that will in turn help them avoid incompatibility problems. To perform the investigation, we extracted metrics related to age of the non-APIs used by the ETPs from the source code of 11 major SDK releases (1.0 to 3.7) and the number of non-APIs used by each ETP from the source code of the 288 ETPs. Furthermore, we also test the source of the ETPs to identify their compatibility in new SDK releases.

The remainder of the chapter is organized as follows: In Section 6.2 we introduce the Eclipse SDK and its applications. In Section 6.3 we discuss how we collected the data that we use. In Section 6.4 we discuss compatibility of the ETPs. In Section 6.5 we discuss the relationship between age of non-APIs used by the ETPs and their compatibility. In Section 6.6 present model building and validation. In Section 6.7 we discuss threats to validity. Section 6.8 we discuss related work. Finally in Section 6.9 we discuss conclusions and future work.

6.2 Eclipse Plug-in Architecture

Eclipse SDK is an extensible platform that provides a core of services for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by wrapping their tools in pluggable components, called *Eclipse plug-ins*, which conform to the Eclipse's plug-in contract. Plug-ins are bundles of code and/or data that contribute functions to a software system. Functions can be contributed, e.g., in the form of code libraries, platform extensions or documentation.

The plug-ins in the Eclipse framework can be categorized three main groups: Eclipse core plug-ins, Eclipse extension plug-ins, and Eclipse Third-party plug-ins.

- *Eclipse core plug-ins (ECPs)*: These are plug-ins present in and shipped as part of the Eclipse SDK. In this thesis, the ECPs and Eclipse SDK are interchangeable. The ECPs provide core functionality upon which all plug-in extensions are built. The ECPs also provide the runtime environment in which other plug-ins are loaded, integrated, and executed. The fully qualified names of ECP packages starts with `org.eclipse`. Examples of ECPs include Java development tools (JDT), Standard Widget Toolkit (SWT) and Platform runtime and resource management (Core) [45].
- *Eclipse extension plug-ins (EEPs)*: These are plug-ins built with the main goal of extending the Eclipse SDK. Most EEPs are large, generic, applications frameworks with tool plug-ins to build other specialized applications. Like the ECPs, fully qualified names of EEP packages start with `org.eclipse`, but as opposed to the ECPs, the EEPs are not shipped as part of the Eclipse SDK. Popular EEPs include J2EE Standard Tools, Eclipse Modeling Framework and PHP Development Tools [45].
- *Eclipse Third-party plug-ins (ETPs)*: These are the remaining plug-ins. The size of these plug-ins ranges from large application frameworks to small specialised applications. Unlike ECPs and EEPs, the package names of ETPs do not have a prefix `org.eclipse`. All the ETPs use at least some functionality provided by ECPs but also may use functionality provided by EEPs. The ETPs are also sometimes referred as Eclipse products/solutions and sometimes Eclipse extensions.

The Eclipse has two main types of interfaces, i.e., visible features that can be reused, it provides to ETPs that reuse its functionality: non-APIs and APIs [5, 37, 38].

- ***Eclipse non-APIs ("bad"):*** The non-APIs, which we also term as *bad interfaces*, are internal implementation artifacts that are found in a package with the substring "internal" in a the fully qualified package name according to Eclipse naming convention [37]. The internal implementations include public Java classes or interfaces, or a public or protected method, or field in such a class or interface. Users are strongly discouraged from adopting any of the non-APIs since they may be unstable [38]. Eclipse clearly states that clients who think they must use these non-APIs do it at their own risk as non-APIs are subject to arbitrary change or removal without notice. Eclipse does not usually provide documentation and support to the non-APIs.
- ***Eclipse APIs ("good"):*** The APIs, which we also term as *good interfaces*, are the public Java classes or interfaces that can be found in packages that do not contain the segment "internal" in the fully qualified package name, or a public or protected method, or field in such a class or interface. Eclipse states that, the APIs are considered to be stable and therefore can be used by any application developer without any risk. Furthermore, Eclipse also provides documentation and support for these APIs.

In this chapter, we are interested in analyzing the non-APIs (classes and interfaces) used by the ETPs.

6.3 Data Set

This study complements our previous studies in Chapter 4 and 5. In Chapter 3, we give a detailed description of how the we conducted data collection from one of the popular

open source repositories (SourceForge) and also present how we cleaned the data. In Chapter 4 we presented how we grouped the ETPs into five of the following classifications based on the usage of non-APIs:

I ETPs with all versions dependent solely on APIs—*good ETPs*;

II ETPs with all versions depending on a non-API—*bad ETPs*;

III ETPs with earlier versions dependent solely on APIs and latter versions depending on a non-API—*good–bad ETPs*;

IV ETPs with earlier versions dependent on a non-API and latter versions depending solely on APIs—*bad–good ETPs*;

V ETPs oscillating between versions that depend solely on APIs and versions that depend on a non-API—*oscillating ETPs*.

6.4 Compatibility of the ETPs

This investigation is a follow-up of the study we presented in Chapter 5. In Chapter 5, we defined an ETPs' compatibility using the definition of *API source compatibility*, i.e., *API source compatibility* requires that the source code of the ETP needs to be recompiled to keep working with new releases of the SDK but no changes have to be made in the sources. Similarily to [26], in the current study *API binary compatibility* and *API runtime compatibility* are considered in our future work.

In Chapter 5, we determined the API source compatibility of each version of the ETP with the different SDK releases (1.0 to 3.7). The API source compatibility of a version on an ETP and a given SDK release was determined by compiling the ETP with the SDK release. The `jar` files of the SDK release we want to test for source compatibility with the version of the ETP are included in the `BUILDPATH` of the version of the ETP. The version of the ETP is compatible with the SDK release in the `BUILDPATH` if zero compile time errors are observed in the Eclipse console. To determine compatibility of the same version of the ETP with another SDK release, we swapped the `jar` files in the `BUILDPATH` of the version of the ETP of the former SDK release with the latter.

In Chapter 5 we found that versions of ETPs that depend on only APIs (ETP-APIs) were always compatible in later SDK releases that do not involve *API breaking changes* [11] and most of the versions of the ETPs that depend on at least one non-API (ETP-non-APIs) were incompatible. For this reason, in the current study we decided to further our investigation only on compatibility of ETP-non-APIs in new SDK releases.

In Chapter 5, we presented both forward and backward source compatibility. Only the results from forward source compatibility are interesting for the current study. The results in Chapter 5 revealed that the majority of the ETPs having the same release year as a given SDK release, were source compatible with that SDK release. In the current study, we have chosen the SDK releases, having the majority of compatible ETPs, to be the baselines of the ETPs' forward source compatibility. A few of the ETPs that were incompatible with the SDK release in the same release year were found to be compatible with the SDK released in the previous year or a year before the previous. In the study in this chapter, these ETPs were categorized in the latest SDK release they compile with as their baseline for forward source compatibility.

			Compatibility of ETPs in new Eclipse SDK releases							
	# ETPs		**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**	**3.7**
2.1	29	S	18	9	8	8	8	8	8	8
		F	11	20	21	21	21	21	21	21
3.0	48	S		28	22	22	21	20	19	19
		F		20	26	26	27	28	29	29
3.1	34	S			15	14	14	14	14	14
		F			19	20	20	20	20	20
3.2	40	S				24	19	17	16	16
		F				16	21	23	24	24
3.3	38	S					25	22	21	20
		F					13	16	17	18
3.4	36	S						31	30	28
		F						5	6	8
3.5	33	S							24	23
		F							9	10
3.6	30	S								28
		F								2

Table 6.1: Compatibility of the ETPs developed on top of a given Eclipse release in new Eclipse release(s). F–Compatibility failure and S–Compatibility success.

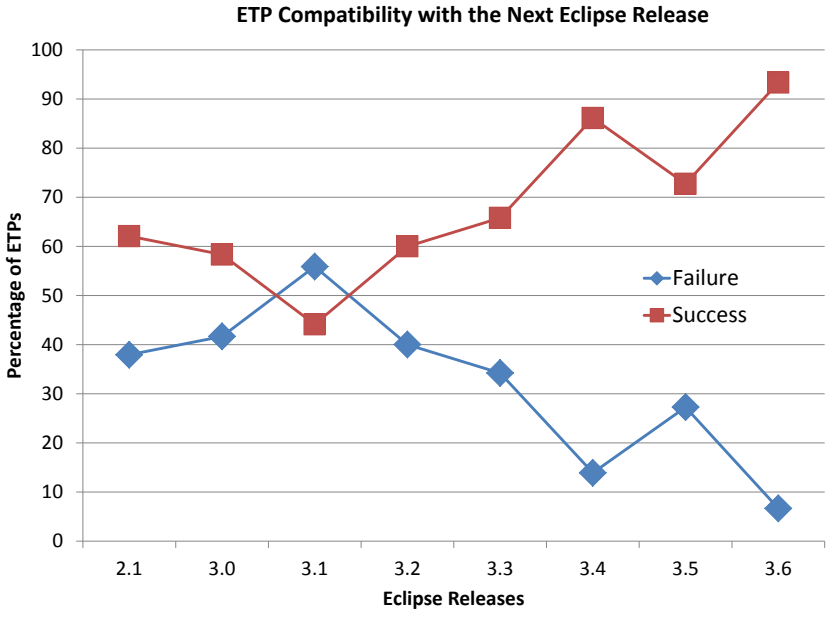


Figure 6.1: Next Eclipse release (diagonal Table 6.1): Compatibility trends for ETPs developed on top of the SDK releases (x-axis) with the next SDK release

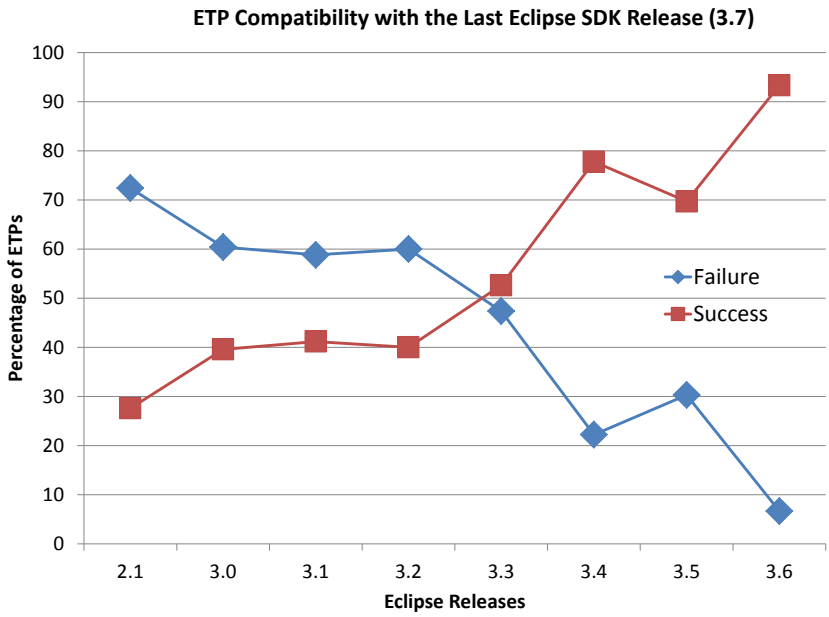


Figure 6.2: Last Eclipse release (colum 3.7 Table 6.1): Compatibility trends for ETPs developed on top of the SDK releases (x-axis) with the most recent SDK release–3.7

Table 6.1 shows the results of the compatibility experiments for the current study. The first column indicates the forward compatibility SDK release baselines. The ETPs in the column # ETPs are all source compatible with the SDK releases in first column. Column 3.0-3.7 show the number of ETPs that successfully compile (S) and fail to compile (F) with the corresponding SDK releases.

Figures 6.1 and 6.2 show the results of compatibility failure and success trends of the ETP supported in the different SDK releases (2.1-3.6) with the next SDK release and the most recent SDK release, respectively. The plots are normalized results of corresponding entries in Table 6.1 (F and S are a percentage of the number on compatible ETPs–column *# ETPs*). We can observe that for both figures there is a general increase in the compatibility success of the ETPs. In Chapter 5, we informally observed that the increase in compatibility success can be attributed to newly released ETPs using "old" non-APIs. In the next section, we will present a quantitative analysis supporting this observation.

6.5 Age of non-APIs vs Compatibility

In this section, we investigate the relationship between the age of the non-APIs used by an ETP and the compatibility of the ETP in a new SDK release. We analyze the compatibility of the ETP by looking at the ETPs' compatibility success or failure in the new SDK release. For age of the non-APIs used by the ETP, we search the SDK release in which the non-API was introduced along the evolution of the SDK.

					# non-APIs from Eclipse SDK Releases								
		Compatibility	**# ETPs**	**Aggregation**	**1.0**	**2.0**	**2.1**	**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**
ETPs Supported in Eclipse Releases	3.2	Success	16	Median	2	0	0	0	0	0			
				Mean	2.2	0.0	0.1	0.3	0.1	0			
				Max	8	3	1	2	1	0			
				Min	1	0	0	0	0	0			
		Failure	24	Median	2.5	1	0	2	0	0			
				Mean	5.8	4.5	0.5	2.3	1.4	0.5			
				Max	32	20	2	8	11	4			
				Min	0	0	0	0	0	0			
	3.5	Success	23	Median	1	0	0	0	0	0	0	0	0
				Mean	2	2.3	0.1	1.5	0.9	0.1	0.1	0.1	0
				Max	10	15	1	8	6	1	2	2	0
				Min	0	0	0	0	0	0	0	0	0
		Failure	10	Median	1.5	0	0	0	0	0	0	0	0
				Mean	2.3	1.8	0.3	0.9	0.5	0.3	0.3	0.1	0
				Max	16	17	4	12	17	6	5	3	0
				Min	0	0	0	0	0	0	0	0	0

Table 6.2: The descriptive statistics of non-APIs in ETPs supported in Eclipse SDK 3.2 and 3.5. The groups (Failure and Success) are with respect to compatibility of the ETPs with Eclipse 3.7.

6.5.1 Hypotheses

We formulate the hypothesis based on the informal observations in Chapter 5 where the use of old non-APIs was found to be more stable.

- $H1_0$: *The age of a non-API used in an ETP supported in a given SDK release, has no impact on the compatibility of the ETP with new SDK releases.*
- $H1_a$: *The older a non-API used in an ETP supported in a given SDK release is, the less likely it will cause compatibility problems of the ETP with new SDK releases.*

6.5.2 Fact Extraction

We carried out the following steps in extracting the facts used in our analysis using a number of scripts:

1. We extracted public non-APIs (classes and interfaces) from each of the SDK releases (1.0 to 3.6) using the Abstract Syntax Tree (AST) of Eclipse JDT.
2. For the non-APIs in each of the collections (1.0 to 3.6), we made new collections which had only non-APIs newly introduced in each of the SDK releases.
3. For each of the ETPs supported in the different SDK releases, we extracted non-APIs used by the ETP.
4. For each non-API in the non-API collection of each ETP, we searched where in the non-API collections of the SDK (step 2 above) is it located.

Table 6.2 shows the distribution of the descriptive statistics of the non-APIs in the ETPs supported in two SDK releases, 3.2 and 3.5. The rest of the descriptive statistics can be found in Appendix C. By definition, an ETP supported in a given SDK release r can only have non-APIs introduced between SDK 1.0 and r, inclusive.

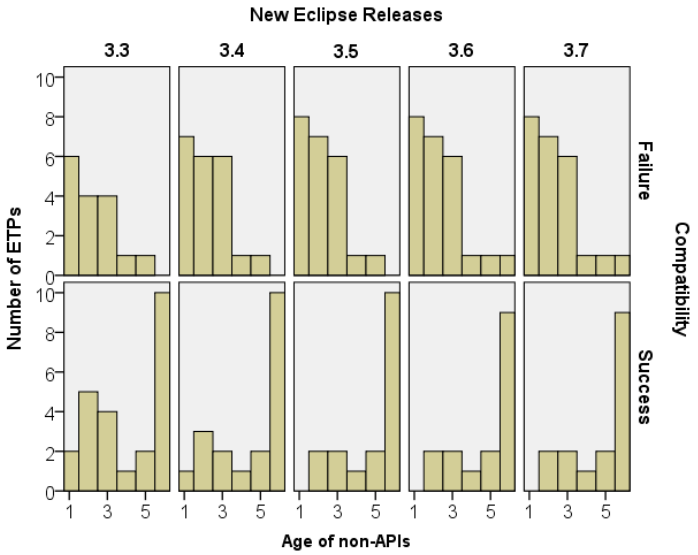


Figure 6.3: ETPs supported in SDK 3.2. x-axis, 1=3.2, 2=3.1, etc.: Histograms showing the source compatibility trends of ETPs supported in SDK 3.2

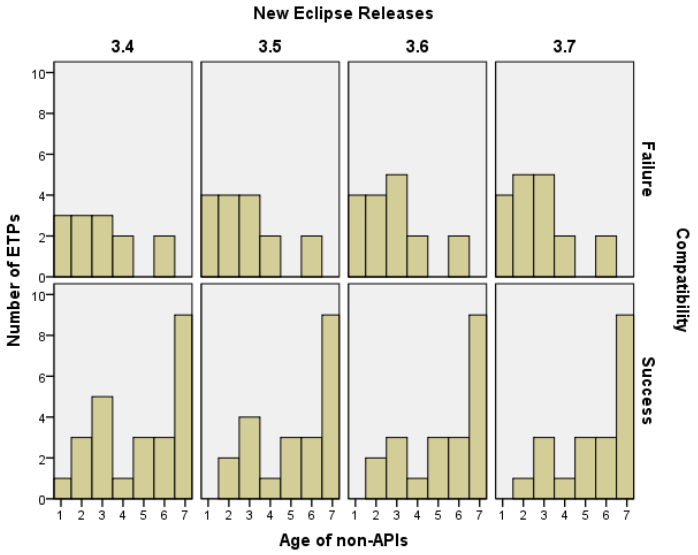


Figure 6.4: ETPs supported in SDK 3.3. x-axis, 1=3.3, 2=3.2, etc.: Histograms showing the source compatibility trends of ETPs supported in SDK 3.3

	non-API age ranking					
	3.2(1)	3.1(2)	3.0(3)	2.1(4)	2.0(5)	1.0(6)
Failure	6	4	4	1	1	0
Success	2	5	4	1	2	10

Table 6.3: Contingency table for *overall age rank* of the non-APIs of the ETPs supported in SDK 3.2 with respect to the compatibility with SDK 3.3.

6.5.3 Data Transformation

To perform the formal statistical analysis of the relationship between the age of the non-APIs used in an ETP and the compatibility of the ETP, we chose to use the ordinal scale to enable us to perform measure of association [10]. We transformed the data into a contingency table that is required for an ordinal measure of association between the variables. Three steps were performed to construct the contingency table. First, we ranked the age of the non-APIs used by the ETP supported in a given SDK release. The age of the non-APIs used by an ETP was ranked in the following way: *age rank 1*–if the non-API was introduced in the SDK release where the ETP is supported, *age rank 2*–if the non-API was introduced in the SDK release preceding the SDK release where the ETP is supported, etc. Second, the overall age of the non-APIs used by the ETP is determined by the youngest used non-API, i.e., overall age in ranking of an ETP is *age rank 2* if the smallest rank in the non-APIs used by the ETP is *age rank 2* (i.e., the ETP does not use non-APIs introduced in the SDK release on which the ETP is supported). The rationale behind the overall age ranking of the non-APIs used by the ETP is based on the stated hypothesis, i.e., the youngest used non-API has the greatest impact on the compatibility of the ETP since it is considered the most unstable. Finally, for each of the new SDK releases the results are grouped according to the compatibility success/failure with respect to the different new SDK releases.

6.5.4 Results

Table 6.3 shows an example of one of the contingency tables of the distribution of the ages of the non-APIs in the ETPs supported in SDK 3.2 with respect to the compatibility with SDK 3.3. The the cells indicate the number of ETPs that succeeded or failed to compile with SDK 3.3 depending on the latest/youngest introduced non-APIs the ETPs use. For example 6 in the cell (Failure, 3.2(1)) indicates that each of the 6 ETPs that failed with SDK 3.3 contained at least one non-API introduced in SDK 3.2 as the youngest non-API. Similarly, 4 in cell (Failure, 3.1(2)) indicates that each of the 4 ETPs that failed with SDK 3.3 contained at least one non-API introduced in SDK 3.1 as the youngest non-API.

Figures 6.3 and 6.4 show the plots of the contingency tables of the ages of the non-APIs for the ETPs supported in SDK 3.2 and 3.3, respectively, with respect to the compatibility with newer SDK releases. The rest of the plots of the other ETPs can be found in Appendix C. For example, the sub-plot–block 3.3 of the left plot in Figure 6.3 represents Table 6.3.

Table 6.4 presents the results of Kendall's Tau-c rank correlation coefficients for the contingency tables of the age of the non-APIs used by the ETPs developed on top of a given SDK release versus the compatibility over new SDK releases. All the correlation coefficients have p-values lower than the threshold of 0.05 except for one with the p-value=0.198 (in bold).

	New SDK releases							
	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
2.1	**.247**	.433	.371	.371	.371	.371	.371	.371
3.0		.389	.293	.293	.345	.398	.450	.450
3.1			.547	.647	.647	.647	.647	.647
3.2				.538	.683	.800	.725	.725
3.3					.476	.665	.693	.759
3.4						.269	.376	.336
3.5							.544	.602

Table 6.4: Kendall's Tau-c correlation coefficients for the contingency tables of the age of the non-APIs used by the ETPs versus the compatibility over new SDK releases.

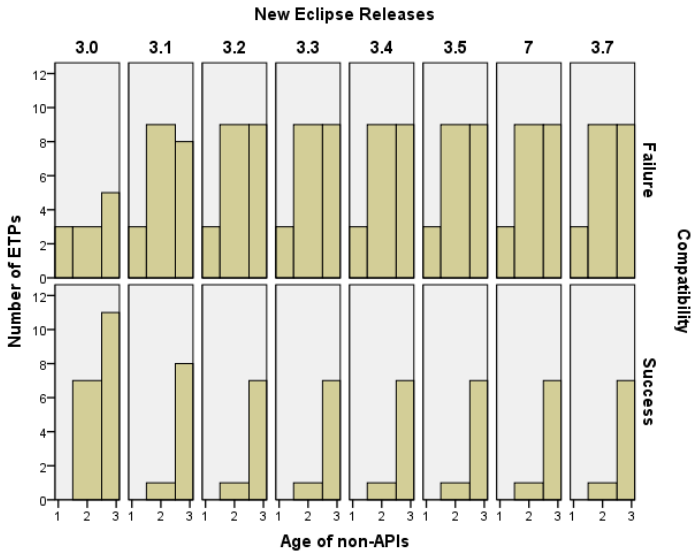


Figure 6.5: ETPs supported in SDK 2.1. x-axis, 1=2.1, 2=2.0, etc.: Histograms showing the source compatibility trends of ETPs supported in SDK 2.1

6.5.5 Discussion

Figures 6.3 and 6.4 show the plots of the results of age versus compatibility of the non-APIs used by the ETPs. From the histograms in Figures 6.3 and 6.4 we can observe that the plots of the ETPs supported in SDK 3.2 and 3.3 that failed in new SDK releases (upper plots), are positively skewed. This indicates that the majority of the ETPs that failed in the new releases use at least one young non-API and very few of the ETPs that fail in new releases use only old non-APIs. The lower plots of Figures 6.3 and 6.4 are negatively skewed indicating that majority of the ETPs that successfully compiled with new SDK releases mostly use very old non-APIs and very few that succeed use a "young" non-API. The rest of the plots in Appendix C show similar trends. The histogram for ETPs supported in SDK 2.1 (Figure 6.5) shows a different shape from the rest of the plots. The reason for this behavior is that the non-APIs used by the ETP are all relatively "young". The ETPs supported in SDK 3.6 (Appendix C) have the highest compatibility success rate in the new SDK 3.7. For the ETPs that successfully compiled with SDK 3.7, none of them had non-APIs newly introduced in SDK 3.6 and 3.5.

The Kendall's coefficient of correlation in Table 6.4 are the results of a formal statistical analysis of the age versus compatibility of the non-APIs used by the ETPs. It should be noted that, unlike the *gamma rank correlation* which ignores all pairs of variables that involve ties, high values of Kendall's Tau-c rank correlation (0.9 or higher) are very rare unless there is a more or less perfect separation in the groups being compared [82]. The perfect separation in our situation occurs if the ETPs that succeed depend only on old non-APIs and the ETPs that fail depend on at least one young non-API.

The values of Kendall's Tau-c are all greater than zero and are all significant at a threshold 0.05, except one. This implies that there is an evidence of the tendency of ETPs that use only old non-APIs to be compatible in new SDK releases (positive relationship of *age* vs *compatibility*).

From the results on the correlation, we can also observe that the strength of the relationship of *age* vs *compatibility* between the ETPs supported in SDK 3.1, 3.2, 3.3 and 3.5 and the corresponding new SDK releases is relatively high. The strength of the relationship between ETPs supported in SDK 2.1 and 3.0 is relatively low since all non-APIs were relatively young at that time. The strength of the relationship for the ETPs supported inSDK 3.4 is relatively low due to presence of many outliers. Furthermore, the variation in the values of the correlation from high to low indicates introduction of outliers in the compatibility of ETPs from one new SDK release to the next (e.g., in row 3.2 from .800 to .725). We discuss the outliers in Section 6.6. The results of ETPs supported in SDK 3.6 are not included since they exhibit over-fitting as a result of imbalance in the number of cases in the groups (i.e., only 2–failure and 28–success). Moreover, the results also indicate that some non-APIs that were introduced in early SDK releases continue to be used in later SDK releases without causing incompatibility problems. This could possibly mean that these non-APIs have reached maturity but they have not been graduated into APIs. From the results presented, we reject $H1_0$ and accept $H1_a$ that "a newly introduced non-APIs used by an ETP is likely to cause incompatibilities of the ETP with new SDK releases".

Based on the findings of age of non-APIs vs Compatibility, we formulate the following question that we would like to address in the next section.

RQ4: *Can prediction model based on the usage of the Eclipse interfaces in the ETPs help in predicting compatibility of an ETP in a new Eclipse SDK release?*

6.6 Compatibility predictions

In this section we will build and validate prediction models for estimating the compatibility of an ETP, supported in a given SDK release, in new SDK releases. To perform a compatibility prediction of an ETP in an SDK release two SDK releases are required: the SDK release in which the ETP is supported and the SDK release for which the prediction is required. Furthermore, the prediction models require the number of non-APIs used by an ETP (taking into account the age of the non-APIs). In Chapter 5 we observed that the ETPs that used only APIs did not have compatibility issues in new releases that do not involve *API breaking changes* (i.e., in the SDK version numbering, the SDK releases that do not have a major segment of the release number changed, e.g., 2.0 to 2.1 or 3.0 to 3.7). Therefore, an ETP that only uses APIs will automatically be predicted by our models to be compatible in any new SDK release.

To build and validate the models, we need a data-set for training the models and a separate data-set for testing the built models. We separated the data for training and testing in the following way: for training, for each models corresponding to the ETPs supported in a given SDK release, we use the data cases for the number of the ETPs in the column *# ETPs* in Table 6.1. For testing, we reuse cases of ETPs supported in earlier SDK releases that still compile with later releases. A detailed explanation data-sets used for training and testing the models is presented in Section 6.6.2.1 and Section 6.6.5.1, respectively.

6.6.1 Hypotheses

In this section we state the hypotheses that will assess how good the predictors are.We will first formally explain the variables that we are going to use in the Hypotheses.

Given a *bad ETP*, e_i, supported in a given Eclipse SDK, E_n, then it may use *bad interfaces* introduced in Eclipse SDKs E_1 to E_n , where $n \geq 1$ (i.e., E_1 is SDK 1.0, E_2 is SDK 2.0 etc.). The SDKs in which the bad interfaces were introduced indicate how old these *bad interfaces* used by the ETP, e_i, are. The goal is to build prediction models to predict compatibility of e_i in newer SDKs, i.e., E_{n+1}, E_{n+1} and so on.

- $H2_0^{e_i,E_n,(E_{n+1},E_{n+2},\ldots)}$: For a given ETP, e_i supported in a given Eclipse SDK, E_n, based on the the number and age of the *bad interfaces* e_i uses, it is not possible build good predictors to predict compatibility of e_i in newer SDK releases, E_{n+1}, E_{n+2} and so on.
- $H2_a^{e_i,E_n,(E_{n+1},E_{n+2},\cdots}$: For a given ETP, e_i supported in a given Eclipse SDK, E_n, with respect to the number and age of the *bad interfaces* e_i uses, it is possible to build good predictors to predict compatibility of e_i in newer SDK releases, i.e., E_{n+1}, E_{n+2} and so on.

The theory on the alternative hypothesis $H2_a^{e_i,E_n,(E_{n+1},E_{n+2},\cdots}$ is based on the the results of Section 6.5. In this section we discovered new bad interfaces have less impact of the compatibility compared to old bad interfaces.

6.6.2 Model training

In this section we will discuss how the trained the models that were built.

		3.5		3.6		3.7	
		β	Sig	β	Sig	β	Sig
3.4	1.0	**-0.462**	**.064**	**-0.235**	**.106**	-0.119	.336
	2.0	-0.840	.708	-0.109	.459	-0.086	.490
	2.1	-1.240	.382	0.137	.886	0.343	.649
	3.0	**-0.788**	**.065**	**-0.575**	**.035**	**-0.603**	**.008**
	3.1	0.903	.225	0.126	.729	-0.131	.717
	3.2	–	–	-0.009	.983	0.115	.782
	3.3	–	–	–	–	–	–
	3.4	–	–	–	–	–	–
	const	**6.770**	**.023**	**4.091**	**.001**	**2.480**	**.000**

Table 6.5: Prediction variables for ETPs supported in Eclipse 3.4 in new Eclipse releases. Variable 3.2, 3.3 and 3.4 was were omitted in the corresponding models since they had very high standard error (indicates multi-collinearity between the predictors). Multi-collinearity is the undesirable situation where the correlations among the independent variables are strong. Multi-collinearity is undesirable because it introduces overfitting in the predicted model. For instance, the model may fit the data well, even though none of the independent variables has a statistically significant impact on explaining the dependent variable.

6.6.2.1 Methodology

The multivariate logistic regression model (based on equation 6.1 and 6.2) below was used to train the models. The data used to train some of the prediction models is presented in the descriptive statistics in Table 6.2, i.e., the prediction models of the compatibility of the ETPs with the last SDK release (3.7). For a given the ETPs supported in a given SDK release, the predictor variables are number and ages of the non-APIs and the predicted variable is the compatibility (failure or success). For example, using the sample of the data we used in Table 6.2, in building the model to predict compatibility of the ETPs supported in SDK 3.2 with SDK 3.7 (upper part of the table), the predictor variables are the non-APIs used by the ETP introduced in the SDK releases 1.0, 2.0,...,3.2 and the predicted variable compatibility which is a binary, i.e., 0 for failure and 1 for success. The logistic function used in building the prediction models is 6.1 [82]:

$$P(Comp) = \frac{1}{1 + e^{-Z}} \tag{6.1}$$

where Z is the linear combination of predictor variables and $P(Comp)$–is the probability of compatibility.

$$Z = \beta_0 + \beta_1 X_1 + \beta_p X_p + ... + \beta_p X_p \tag{6.2}$$

where Xs are the predictor variables and p is the number of predictor variables.

To perform the analysis, we used the software IBM SPSS Statistics 19. We used the *backward-stepwise elimination* method (Backward: LR) to build the models. Backward-stepwise elimination starts with all the variables in the model, then at each step, variables are evaluated for entry and removal. For all the models, the *stepwise probability* or *p-value* for entry and removal of the variables of 0.05 and 0.1, respectively, was used. *Forward-stepwise selection* was also tried and the results were more or less the same as

		3.6		3.7	
		β	Sig	β	Sig
3.5	1.0	-0.139	.393	**-0.927**	**.067**
	2.0	-0.141	.243	**-0.528**	**.077**
	2.1	**-2.042**	**.068**	0.422	.904
	3.0	-0.050	.930	**0.867**	**.147**
	3.1	0.056	.883	-0.564	.373
	3.2	**-3.018**	**.028**	**-3.583**	**.088**
	3.3	1.274	.370	1.701	.554
	3.4	-0.973	.316	-1.309	.381
	3.5	–	–	–	–
	const	**3.293**	**.005**	**6.025**	**.020**

Table 6.6: Prediction variables for ETPs supported in Eclipse 3.5 in new Eclipse releases. None of the non-APIs present in the ETPs was introduced in Eclipse 3.5. This variable was omitted in training the models.

		Predicted					
		New SDK releases					
		3.5		3.6		3.7	
Observed		S	F	S	F	S	F
3.4	S	30	1	29	1	26	2
	F	2	3	3	3	5	3
3.5	S			24	0	23	0
	F			4	5	1	9

Table 6.7: Classification results form model training. S–Compatibility, F–Incompatibility

	3.5			3.6			3.7		
	A	P	R	A	P	R	A	P	R
3.4	92	97	94	89	97	91	81	93	84
3.5				88	100	86	97	100	96

Table 6.8: Error analysis for model training. A–Accuracy, P–Precision and R–Recall

the *backward-stepwise elimination* method and the *Enter* method a little bit worse. Using equation (6.1) in the models, the predicted value (observed prediction), are in the range of 0 to 1. For all the models, we applied a threshold of 0.5 on the predicted value, i.e., the model considers an ETP to be incompatible if the predicted value is less than 0.5, and compatible, otherwise.

6.6.2.2 Results

Table 6.5 and 6.6 present the results of the linear combinations (equation 6.2) of the predictor variables for the ETPs supported in SDK 3.4 and 3.5 with respect to the ETPs' compatibility in the corresponding newer SDK releases. The rest of the models can be found in Appendix C. After the stepwise elimination analysis, the predictor variables in bold remained in the model, i.e., these variables have significant impact on the outcome of the prediction. The variables not in bold are not considered to have a significant impact on the outcome of the prediction. Table 6.9 presents the results of *Nagelkerke* R^2 for all the corresponding models. *Nagelkerke* R^2 indicates the strength of the relationship between the predictors and the prediction in the model.

For example, in Table 6.6, the variables 2.1, 3.2 and *const* remained in the model as the most significant predictors on the outcome of the prediction in from SDK 3.5 to 3.6. Therefore, prediction model of ETPs supported in SDK 3.5 for their compatibility in SDK 3.6 is shown in Equation 6.3 below. To make a prediction for an ETP supported in SDK 3.5 for its compatibility in SDK 3.6 one should substitute only the number of non-APIs used by the ETP introduced in SDK 2.1 and 3.2. The corresponding value of *Nagelkerke* $R^2 = 0.691$, indicating strength of the relationship between the predictors and the prediction in the model from SDK 3.5 to 3.6 is shown in cell (3.5, 3.6) in Table 6.9. Table 6.5 shows the corresponding results of prediction for ETP supported in SDK 3.4 in newer SDK releases.

$$P(Comp) = \frac{1}{1 + e^{-3.293+2.042*SDK_{2.1}+3.018*SDK_{3.2}}} \tag{6.3}$$

Model to predict compatibility of ETPs supported in SDK 3.5 in the new SDK 3.6.

The classification and error analysis training for the models we trained for the ETPs supported in SDK 3.4 and 3.5 are presented in Table 6.7 and Table 6.8, respectively. The rest of results for the classification and error analysis for all the models we trained can be found in Appendix C. Table 6.7 shows the number of correctly and incorrectly classified cases. For example, the main cell (3.4,3.5) shows the prediction results for the model between SDK 3.4 and 3.5. The sub-cell (S,S)=30 indicates the number of correctly predicted compatible ETPs, sub-cell (S,F)=1 indicates the number of incorrectly predicted compatible ETPs, sub-cell (F,S)=2 indicated the number of incorrectly predicted incompatible ETPs and sub-cell (F,F)=3 indicated the number of correctly predicted incompatible ETPs.

Table 6.8 shows the results of the computation of *accuracy*, *precision* and *recall* [13] corresponding to the results presented in Table 6.7. *Accuracy* measures the percentage of all decisions that were correct decisions, i.e., computed from the numbers in Table 6.7, i.e., ((S,S) + (F,F))/((S,S)+(S,F)+(F,F)+(F,S)). *Precision* measures the percentage of the assigned categories that were correct, i.e., (S,S)/((S,S)+(S,F)). *Recall* measures the percentage of the correct categories that were assigned, i.e., (S,S)/((S,S)+(F,S)).

	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
2.1	.462	.471	.386	.386	.386	.386	.386	.386
3.0		.362	.356	.356	.377	.456	.477	.477
3.1			.449	**.810**	**.810**	**.810**	**.810**	**.810**
3.2				**.517**	**.582**	**.571**	**.512**	**.512**
3.3					**.545**	**.657**	**.629**	**.769**
3.4						**.703**	**.503**	.383
3.5							**.691**	**.803**

Table 6.9: Nagelkerke R^2 values showing the strength of the relationship between the predictors and the prediction for all the models that we generated. The numbers in bold indicate high values of R^2.

6.6.2.3 Discussion

In this section we will discuss the results of model training. A number of observations that can be drawn from the results. First, we observe high values of the strength of the relationship between the predictors and the prediction in the model indicated by values of Nagelkerke, R^2 in Table 6.9. As stated in [82], the values of R^2 for the logistic regression model are typically much smaller than what is usually observed in R^2 for linear regression model. In a related study by English et al. [43], the authors obtained an R^2=0.371 and they state that logistic regression of $R^2 > 0.3$ is considered good and therefore they consider their model to be a good predictor. Zhou and Leung in [116] found a value of R^2=0.382 and 0.409 in the two models they presented. Gyimothy et al. [50] found an even much lower value of R^2=0.175 and also state that their model can make useful predictions. In comparison to the related work, we observe very high values of R^2. Like Gyimothy et al. [50] and English et al. [43], we also used the univariate logistic regression but it performed worse than the multivariate regression. From Table 6.9 we observe that the R^2 values of ETPs supported in SDK 2.1 and 3.0 are relatively low compared to those in later SDKs. The reason is that the non-APIs used in 2.1 and 3.0 are relatively "young" compared to the non-APIs used in ETPs in later SDK releases.

For testing goodness of fit in our models, we also looked at the Hosmer-Lemeshow statistic. All the Hosmer-Lemeshow statistics for our models were not significant at a threshold of 0.05, in fact, they were always higher than 0.25. The desired outcome of non-significance indicates that the model prediction does not significantly differ from the observed. We also cross-checked the correlation between the predictor variables. A few of the variables were highly correlated that caused over fitting in the models, these variables without values in Tables 6.5 and 6.6. These variables also had a very high standard error (greater than 2.5) when included in the over fitted model. We excluded these variables in the models as can be seen in the models we presented in Section 6.6.2.2.

6.6.3 Model outliers

Like many empirical studies, our data was subject to outliers. In all the models, the average number of outliers ranges between zero and two. The highest number of outliers of four was observed in only one of the models relating SDK 3.2 and 3.3. We observed three types of outliers: 1) ETPs that depend only on very old non-APIs and failed to compile in new releases. The outliers of ETPs with very old non-APIs are of two types: those that graduate from non-API to APIs and those that change their interface and

	Predictor Variables								
	1.0	2.0	2.1	3.0	3.1	3.2	3.3	3.4	3.5
Significant	25	24	2	20	3	4	0	0	0
Total	35	35	35	27	20	14	9	5	2
Percentage	71	69	6	74	15	29	0	0	0

Table 6.10: The frequency of the predictor variables in the models. Significant–number of times a predictor variable was considered significant, Total–number of times a predictor was used to build the model.

still stay non-APIs. Failure caused by only graduation of non-API was found in only one of the ETPs we analyzed. This indicates that graduation of non-APIs is very limited. Preliminary investigation on the graduation of non-APIs to APIs during the evolution of SDK showed similar results. In a follow-up one can investigate graduation on non-APIs in the evolution of Eclipse. 2) ETPs that had dependency on newly introduced non-APIs but still compiled with new SDK releases. One possible reason for this scenario could be that the non-API is actually old but was just renamed or moved. Our methodology identifies renamed and moved classes as newly introduced.

6.6.4 Predictor frequency

In this section we present and discuss the most frequently selected significant predictors in the built models.

Table 6.10 presents the a summary results of frequency of the most significant predictors selected. Detailed results can be seen in Appendix C. With the exception of predictor 3.5, all other predictors have non-APIs (cf. Appendix C). We observe that the most frequent significant predictors are 1.0, 2.0 and 3.0. SDK releases 1.0, 2.0, and 3.0 are releases considered to have *API breaking changes* [11]. Despite the fact that the *API breaking changes* have got nothing to do with non-APIs, our study reveals that non-APIs from SDK releases 1.0, 2.0 and 3.0 have a significant impact on the compatibility of the ETPs.

6.6.5 Model validation

In this section we test the models we built in Section 6.6.2.2. Below we describe the methodology we used to collect the data for testing the models.

6.6.5.1 Methodology

For a given model under test, we collected cases of data that was used to build models for ETPs supported in earlier SDK releases. For example, testing the models between ETPs supported in SDK 3.1 with new SDK releases, we use cases of ETPs supported in SDK 3.0 and SDK 2.1 that compiled with SDK 3.1. The rationale is that if an ETP compiles with respect to an SDK release, then it is supported in that SDK release. Since we also know the compatibility results of these ETPs with newer SDK releases after 3.1, we consider these compatibility results as the *expected results*.

For computing the *observed results*, the collected cases are substituted into the model between SDK 3.1 and the SDK we are trying to predict (equation 6.1). Since the cases imported from ETPs developed for earlier SDKs have no entries (non-APIs) in later releases, we put zeros in those missing entries. For example, the cases imported from

		Predicted					
		New SDK releases					
		3.5		3.6		3.7	
Observed		S	F	S	F	S	F
3.4	S	82	0	79	0	79	0
	F	5	0	6	2	6	2
3.5	S			102	6	96	9
	F			4	0	4	3

Table 6.11: Classification results form model testing. S–Compatibility, F–Incompatibility.

	3.5			3.6			3.7		
	A	P	R	A	P	R	A	P	R
3.4	94	100	94	93	100	93	93	100	93
3.5				91	94	96	88	91	96

Table 6.12: Error analysis for model testing. A–Accuracy, P–Precision, R–Recall.

SDK 3.0 to 3.1 have no non-APIs introduced in SDK 3.1 as the ETPs existed earlier than SDK 3.1. We put zeros in the entry 3.1 which also implies that each of the non-APIs in the imported cases will be one year older in SDK 3.1 compared to SDK 3.0.

6.6.5.2 Results

Table 6.11 and 6.12 present a sample of classification results and error analysis, respectively, from model validation for the ETPs supported in SDK 3.4 and 3.5. The explanation of the entries is similar to that of Table 6.7 and 6.8 from the results of model training. The rest of the results can be found in Appendix C.

6.6.5.3 Model Discussion

Like in model training, the results of model testing reveal high values of accuracy, precision and recall. Furthermore, out of curiosity, we validated the models by including all predictors, ignoring the respective significance of the predictors. We observed that there is no big difference when all the predictor variables are included in the model. The results from model testing for the comparison when only significant predictors and when all the predictors are included in the models are shown in Appendix C.

From the classification results in Table 6.11 (and the rest in Appendix C), we can observe that we have very few data cases in the incompatibility group compared to those in the compatibility group. This is due to the fact that most of the ETPs that compiled with the next SDK release after the SDK in which the ETPs are supported also compiled with the rest of the new SDK releases until SDK 3.7. Furthermore, we also observe that the percentage of incorrectly predicted incompatible ETPs is high in some cases. This is due to the fact that some of these incompatible ETPs were observed as outliers during the model training phase.

6.6.6 Overall discussion

We have trained 35 compatibility predictive models and tested 27 of the 35 prediction models. Eight of the 35 models related to ETPs supported in SDK 2.1 could not be tested since we did not have data to test them. Recall that in Section 6.6.5.1 we stated that the data for testing a given SDK model was gotten from data for ETPs supported in earlier SDKs. This is the reason we do not have data to test ETPs supported in SDK 2.1.

From the discussion of the results of model training and testing in Section 6.6.2 and 6.6.5, respectively, we can reject $H2_0$ and accept $H2_a$ that "for a given ETP supported in a given SDK release, based on the number and age of *bad interfaces* the ETP uses, it is possible to build good predictors to predict compatibility of the ETP in newer SDK releases".

We observe that our results generate a different model for most of the predictions. There are two main reasons related to the observed phenomenon: First, models generated across ETPs supported in the different SDKs with respect to new SDK releases, apparently, they are all from different domains, i.e., the non-APIs they depend on have different maturity levels. Non-APIs introduced in a given SDK release represent different ages or stability/maturity levels in the different categories of ETPs. An non-API introduced in SDK 1.0 represents different stability/maturity levels when used in an ETP supported in SDK 2.1 compared to SDK 3.5. By SDK 3.5 the non-API undergone numerous evolutions. Second, models generated by ETPs supported in a given SDK release (ETPs in the same domain) with respect new SDK releases are also different. The reason to this difference is that the new SDK releases represent different systems. The non-APIs used by the ETPs provide different functionality during evolution of Eclipse. This can be seen by the increase in the number of incompatibilities with every new release. For situations where the all the non-APIs used by the ETPs did not change between releases, we observe that we have the same models, e.g., from SDK 3.6 to 3.7 in for ETPs supported in SDK 3.2 (Appendix C).

6.7 Threats to Validity

As any other empirical study, our findings may subject to validity threats. We categorize the possible threats into construct, internal and external validity.

Construct validity focuses on how accurately the metrics utilized measure the phenomena of interest. The methodology used to measure the age of the non-APIs used by the ETPs, is subject to construct validity. Because we wanted to use a quick way of determining the age of non-APIs, the applied methodology considers renamed or moved non-APIs as newly introduced in the SDK releases. In a follow-up study we want to use code cloning to determine the moved and renamed non-APIs [35].

Internal validity is related to validity of the conclusion within the experimental context of the ETP collection considered above. We have paid special attention to the appropriate use of statistical machinery in all the results we have presented.

External validity is the validity of generalizations based on this study. Much as we only consider Open Source ETPs from SourceForge, we could say that our results are generalizable to all the Eclipse solutions since we do not measure ETP specific artifacts. The artifacts we measure are based on the framework. However, our results may not be generalizable beyond the Eclipse-based solutions. Going beyond Eclipse we realize that the same study needs to be carried out on a different plug-in framework.

6.8 Related Work

This work complements our previous work in Chapters 2–5. In Chapters 2 we investigated the constrained evolution of 21 carefully selected ETPs on Lehman's software evolution laws. Specifically, we investigated the evolution of ETPs' dependencies on Eclipse interfaces in the new releases of the ETPs. In Chapter 4 we investigated the Eclipse interface usage by 512 ETPs, and proposed the distinction between *good ETPs* and *bad ETPs*. In Chapter 5, we investigated the survival of ETPs in SDK releases based on whether they use or do not use bad interfaces from Eclipse SDK. The study in this chapter is one of the follow-up studies proposed in Chapter 5.

Class change proneness: Several studies have applied metrics related to internal factors that impact change proneness in a software system. Di Penta et al. in [39] and [17] investigated the relationship between design pattern roles and class change proneness. Penta et al. in [39] carried out a finer-grain level of design pattern roles. This study reports some classes playing certain design pattern roles (e.g., Adapter, Receiver and Command) are more change prone compared to the classes that do not play these roles. Romano and Pinzger in [90] investigate the relationship between Chidamber and Kemerer metrics [51] and change proneness. The authors report that cohesion exhibits the strongest correlation with the number of code changes. In comparison to our study, we investigate the impact *age* of non-APIs (which is an external factor), used by the ETPs on change proneness of the non-APIs.

Predictive models: A number of studies have reported predictive models. Most of these models are based on predicting defects in a software system. Mende and Koshke in [74] presented two strategies, probabilistic classifier and regression algorithms, to incorporate the treatment of effort into defect prediction models. A number of studies have built defect prediction models based on C&K metrics for example [43, 50, 116]. In comparison to our study, we report compatibility prediction models of framework-based applications in new versions of the framework.

6.9 Conclusions and Future Work

In this paper we analyzed source code to investigated two main goals: 1) the relationship between the age of *bad interfaces* used by an ETP and the compatibility of the ETP in new Eclipse SDK releases. 2) to build and validate a compatibility predictive model for an Eclipse third-party plug-in (ETP), supported in a given SDK release, in a new SDK release. For first goal, we have observed that newly introduced non-APIs used by an ETP are likely to cause incompatibilities of the ETP with new SDK releases. For the second goal, we have trained 35 compatibility predictive models and tested 27 of the 35 predictive models. Eight of the 35 predictive models related to ETPs supported in SDK 2.1 could not be tested since we did not have data to test them. We have shown that the built models can generate good predictions with high accuracy, precision and recall. Furthermore, despite the fact that SDK releases with *API breaking changes*, i.e., 1.0, 2.0 and 3.0, are only concerned with APIs, our findings reveal that non-APIs introduced in these releases have a significant impact on the compatibility of the ETPs that use them in new SDK releases.

The investigation of the stability of the *bad interfaces* is based on an external artifact, i.e., the impact the *bad interfaces* has on software that use it. In a follow-up study we plan to carry out an investigation by measuring the internal artifacts of the *bad interface*, e.g.,

relationship age of the *bad interface* and changes on the signature of the *bad interface*.

In testing the models, based on the methodology we used to collect testing samples, we had very few cases of ETPs that were incompatible. In some models, the percentage of the incorrectly predicted ETPs was high for the few incompatible cases considered. In a follow-up study one can collect more samples to test the models. One can also investigate the possibility of combining predictors to make better predictions (e.g., using a non-linear combination of predictor variables). The models presented can be used to make interpolation compatibility predictions, i.e., predictions can only be made SDK releases that were used to train the models. In a follow-up study one can build extrapolation prediction models, i.e., making predictions on an SDK we have not used in building the model. Finally, one can come up a domain specific tool based on the models that can be used to make predictions with the help of a tool like RASCAL [58].

Chapter 7

Analyzing the Eclipse API Usage: Putting the Developer in the Loop

Eclipse guidelines distinguish between two types of interfaces provided to third-party developers, i.e., APIs and non-APIs. APIs are stable and supported, while non-APIs are unstable, unsupported and discouraged as they are subject to arbitrary change or removal without notice. In our previous work, we found that despite the discouragement of Eclipse, the use of non-APIs in Eclipse third-party plug-ins (ETPs) is not uncommon. Furthermore, we found that these non-APIs are the main cause of ETP incompatibilities in forthcoming releases of the Eclipse.

In the current work we conducted a survey aiming at understanding why do the ETP developers use non-APIs. We have observed that developers with a level of education of up to master degree have a tendency not to read product manuals/guidelines. Furthermore, while for less experienced developers instability of the non-APIs overshadows their benefits, more experienced developers prefer to enjoy the benefits of non-APIs despite the instability problem. Finally, we have observed that there are no significant differences between Open Source and commercial Eclipse products in terms of awareness of Eclipse guidelines and interfaces, Eclipse product size and updating of Eclipse product in the new SDK releases.

7.1 Introduction

Software engineering researchers constantly carry out studies with the aim of obtain "convincing evidence" that may improve software projects. Oram and Wilson [84] state that despite the decades of software engineering research, so far they has seen relatively few examples of convincing evidence that have actually led to changes in how people run software projects. Oram and Wilson conjecture that the possible reason for relatively few examples of convincing evidence could be a result of a context problem: researchers have been generating evidence about different topics than the ones the developers care about. To understand the actual developers' needs Oram and Wilson [84] recommend the researchers to include the developer in the loop.

Typical examples of excluding a developer from the loop are in our previous studies related to Eclipse interfaces usage in Chapters 4–5 also reported in [24, 26]. In the mentioned previous studies we had limitations in answering certain questions. Eclipse distinguishes between two types of interfaces: API that are stable, supported and non-APIs that are unstable, unsupported and discouraged as they are subject to arbitrary removal without notice. In Chapter 4, we discovered that 44% of Eclipse third-party plug-ins (ETPs) on SourceForge use non-APIs. We also discovered that the ETPs that depend on at least one non-API are larger than those that depend only on APIs [24]. Furthermore, we also discovered that the Eclipse non-APIs cause ETPs to fail when ported to new releases of Eclipse SDK [26]. However, we could not answer why the ETPs that depend on at least one non-API are larger than those that depend only on APIs and why the developers use problematic non-APIs. Furthermore, in our related studies in Chapters 2–6 generalisation of our results to closed-source software was considered as a threat to external validity as our investigation was based on only open-source software products.

Therefore, we decided to include the developer in the loop by carrying out a survey on Eclipse interface usage by Eclipse product developers.

The chapter is organized as follows. In Section 7.2 we discuss the study definitions, design, and planning. Results are presented and discussed in Section 7.3. Section 7.4 discusses the threats of the study. Section 7.5 discusses related work and Section 7.6 concludes the chapter.

7.2 Study Definition, Design, and Planning

The survey was conducted during the summer of 2012. The main aim of the survey was to achieve an accurate picture of the state-of-the-practice of Eclipse interface usage by Eclipse product developers. The goals of the survey are twofold. First, we aim at identifying whether application developers are aware of the differences between APIs and non-APIs and related guidelines. Second, we aim at understanding the differences in characteristics of software applications that use APIs and non-APIs.

7.2.1 Research questions

Given the above goals, the survey answers three main research questions below. First, the research questions are based on the following categories encapsulating the different factors that may affect the development and maintenance of Eclipse products (the factors are outlined in Section 7.2.2): 1) Education and experience, 2) General Eclipse product development, 3)Eclipse interfaces and their guidelines, 4) use of non-APIs, and 3) testing of Eclipse product.

RQ1 What is the relationship between factors in the identified categories and how can we explain the observed relationship?

RQ2 What are the differences in characteristics between Eclipse products that use and do not use non-APIs in terms of the factors in the identified categories?

RQ3 What are the differences in characteristics between closed-source and open-source Eclipse products in terms of the factors in the identified categories?

7.2.2 Survey terms and entities of interest

In this section we define the entities of interest that form the basis for design of the survey questionnaire. First, we define the terms used in the survey:

- *Eclipse SDK*: This is an extensible platform that provides a core of services for controlling a set of tools working together to support programming tasks. The SDK provides generic interfaces that can be used to build specialized applications.
- *Eclipse non-APIs*: In addition to Java access levels, Eclipse has another level called internal implementations. According to Eclipse naming convention [37], the non-APIs are artifacts found in a package with the segment `internal` in a fully qualified package name. These artifacts may include public Java classes, interfaces, public or protected methods, or fields in such a class or interface.
- *Eclipse APIs*: As opposed to non-APIs, APIs are found in packages that do not contain the segment `internal`.
- *Eclipse provisional API guidelines*: This is a document provided by Eclipse describing the different interfaces it provides and the evolution of these interfaces from initial embryonic forms to real battle-hardened APIs with guarantees of long term support. The document defines public interfaces that can be used, i.e., APIs and those that should not be used or used at a risk (non-APIs). The document also sets out rules for committers on how to indicate APIs that are still under development and subject to change. The guidelines are also useful for API clients who want to know about the state of a given Eclipse interface they are using [5].
- *Eclipse product*: This is a software system that reuses the functionality provided by Eclipse SDK. The Eclipse product is sometimes referred to as Eclipse solution and sometimes Eclipse third-party plug-in.

We identified five groups to help us investigate the previously formulated the research questions:

- *Education and experience*: The aim of this category of questions was to establish the experience and education levels of the respondents. In particular, we were interested in collecting information on years of education and years of experience as a software developer, Java developer and Eclipse product developer.
- *General Eclipse product development*: The aim of this category of questions was to identify the characteristics of the Eclipse product developers and the human aspects influencing maintenance of the Eclipse product. In particular, we were interested in collecting data about reasons why developers develop their Eclipse products, development team size, weekly hours dedicated for maintenance and development, importance of updating, number of versions (NOV) released, and number of files (NOF) of the latest version of product.
- *Eclipse interfaces and their guidelines*: The aim of this category of questions was to discover if developers are aware of the existence of the Eclipse provisional guidelines and accessibility levels of the interfaces that Eclipse provides. In particular, we wanted to collect data about developers awareness of the "Eclipse Provisional API Guidelines" and if they are aware of the two types of interfaces, APIs and non-APIs.

- *Use of non-APIs*: The aim of the questions in this category was to identify why developers use or do not use non-APIs, avoid using non-APIs and if they deliberately or not avoid using non-APIs.
- *Testing of Eclipse product*: The aim of this category of questions was to identify if developers test their versions with new Eclipse SDK releases/milestones/betas.

7.2.3 Questionnaire design

The questionnaire contains both open-ended and multiple-choice questions. The questions in the survey are organised according to the identified categories in Section 7.2.2. The survey was designed using `limesurvey`,[1] an open-source web-based survey application. Tables 7.1 and 7.8 present the multiple-choice and open-ended questions, respectively.

7.2.4 Survey execution

In order to increase acceptability of the survey by respondents, as well as avoid ambiguous and biased questions in the survey, the survey was reviewed internally and externally. The internal review was done by ten colleagues at the university. For the external review, about three months before the survey was deployed, we invited Eclipse product developers to review the survey. Three developers accepted our invitation and gave us remarks, which we incorporated in the survey.

The survey was announced on Eclipse Twitter account and news page by Wayne Beaton, the director of open-source projects, Eclipse foundation. The data was collected between 15 June and 31 July 2012.

7.3 Results

In this section, we will first present the information about the non-respondents, then present the descriptive statistics and finally discuss the results of the analysis corresponding to the research questions. We shared the survey data of our study for replication or further analyses [1].

7.3.1 Non-respondent analysis

First, we conducted a data reduction step where incomplete and invalid responses were excluded from the analysis. The survey attracted 114 respondents who started to fill in the questionnaire. All the 114 respondents filled the first section—*Education and experience*, of the survey. Only 40 of the 114 continued and filled the second section—*General Eclipse product development*. 36 of the 40 who filled the second section continued to fill the third section–*Eclipse interfaces and guidelines*. 31 of the 36 who filled the third section continued to fill the fourth and fifth sections—*Use of non-APIs* and *Testing of the Eclipse product*, respectively. Only one respondent of the 31 filled in meaningless answers, leaving 30 answers amenable for further analysis.

One possible reason why there was a heavy drop of respondents from 114 to 40 was that in the *General Eclipse product development* section we asked the respondents to name the plug-in they developed. It is possible that the respondents that dropped out did not develop plug-ins or were reluctant about disclosing the information about their plugins.

[1] `http://www.limesurvey.org`

Var #	**Variable**	Scale	**Description**
			Education and Experience of Respondents
i	educa	1–6	How many years of education after secondary school graduation do you have?
ii	expsd	1–6	How many years of experience as a software developer do you have?
iii	expjd	1–6	How many years of experience as a Java developer do you have?
iv	exped	1–6	How many years of experience as an Eclipse product/solution developer do you have?
			General Eclipse Product Development
v	whydev	N/A	Why did you develop your Eclipse product?
vi	licnc	N/A	What is the licence type of your Eclipse product?
vii	store	N/A	If option in (vi) was open-source. Where did you publish the source code of your Eclipse product?
viii	tsize	1–5	What is the average size of your Eclipse product development team?
ix	upimp	1–3	How important is it to update your Eclipse product in relation to new versions of Eclipse SDK APIs?
x	NOV	1–7	How many new versions of Eclipse product have you released since the initial release?
xi	NOF	1–4	What is the size (estimate) of the latest version of your Eclipse product in terms of number of files?
xii	whours	1–5	How many hours/week do you dedicate updating your Eclipse product in relation to Eclipse SDK API usage?
			Awareness of Eclipse Interfaces and their Guidelines
xiii	awAPg	1–2	Are you aware of the “Eclipse Provisional API Guidelines”?
xiv	flAPg	1–4	Do you follow “Eclipse Provisional API Guidelines” when developing your Eclipse product(s)?
xv	awAnNA	1–2	Are you aware that Eclipse provides two types of interfaces (APIs and non-APIs)?
			Use of non-APIs
xvi	usenNA	N/A	Do you use non-APIs?
xvii	avusNA	N/A	If answer was “NO” in (xvi). Do you deliberately avoid using Eclipse non-APIs?
xviii	deusNP	N/A	If answer was “YES” in (xvi). Do you deliberately use the non-APIs?
			Testing of Eclipse Product
xix	1stSDK	N/A	On which Eclipse SDK release did you develop the first version of your Eclipse product?
xx	testP	N/A	Have you tested your Eclipse product on newer eclipse SDK releases, betas or milestones?
xxi	freTst	N/A	If answer was “YES, the on next release and/or on a release after the next or later. Are you frequently testing you Eclipse product on new SDK releases?

Table 7.1: Variable description and corresponding questions. Scale—Ordinal scale

Years	**educa** (i)		**expsd** (ii)		**expjd** (iii)		**exped** (iv)	
	fre	%	fre	%	fre	%	fre	%
<2	1	3.3	0	0.0	0	0.0	1	3.3
≥2 & <4	0	0.0	2	6.7	3	10.0	7	23.3
≥4 & <6	16	53.3	1	3.3	2	6.7	8	26.7
≥6 & <8	6	20.0	2	6.7	7	23.3	7	23.3
≥7 & <10	2	6.7	5	16.7	4	13.3	6	20.0
≥10	5	16.7	20	66.7	14	46.7	1	3.3
Total	30	100	30	100	30	100	30	100

Table 7.2: Years of education and experience of the respondents. fre–Frequency and %–Percentage. The number in brackets after every variable name corresponds to the variable number in Table 7.1.

The other reason why respondents dropped out in *Eclipse interfaces and guidelines* section onwards is because the survey was comprehensive and the respondents got less interested.

7.3.2 Descriptive Statistics

In this section we present the descriptive statistics for the variables corresponding to multiple choice questions. Table 7.1 presents the questions and the respective variable names. Due to question branching, some questions were not filled by all the 30 respondents. In the descriptive statistics, we shall explicitly state the number of respondents that answered the question only when the number is less than 30.

7.3.2.1 Education and Experience of Respondents

Table 7.2 shows the frequency and percentage distributions of the four variables that belong to this category, i.e., *educa*, *expsd*, *expjd*, and *exped*.

7.3.2.2 General Eclipse Product Development

The following list presents the frequency and percentage distributions of the cases of the variables in this category. The item number corresponds to the variable number in Table 7.1.

v. *Why did you develop your Eclipse product?*: 13 (43.3%) of the 30 were employed on *full-time job*, 4 (13.3%) were employed as a *part-time job*, 6 (20.0%) developed plug-ins as a *hobby*, 2 (6.7%) developed the plug-in as part of an *educational use*, and 5 (16.7%) were in the *other* category.

vi. *What is the licence type of your Eclipse product?*: 16 (53.3%) of the 30 selected *open-source* and 14 (46.9%) selected *closed-source*.

vii. *If option in (vi) was open-source. Where did you publish the source code of your Eclipse product?*: 16 respondents answered this question, where 6 (37.5%) of the 16 had SourceForge as their repository, 2 (12.5%) had GoogleCode, 2 (12.5%) had GitHub, 4 (25.0%) had eclipse.org and 2 (12.5%) were in the *other* category.

viii. *What is the average size of your Eclipse product development team?*: 11 (30.8%) of the 30 developed the Eclipse product *solely*, 11 (36.7%) had a team of *2 to 4 persons*, 2 (6.7%) had a team of *5 to 7 persons*, 3 (10.0%) had a team of *8 to 10 persons* and 3 (10.0%) had a team of *more than 10 persons*.

ix. *How important is it to update your Eclipse product in relation to new versions of Eclipse SDK APIs?*: 6 (20.0%) of the 30 updating their Eclipse product in relation to Eclipse SDK APIs was *not important*, 12 (40.0%) updating was *moderately important* and 12 (41.0%) updating was *very important*.

x. *How many new versions of Eclipse product have you released since the initial release?*: 1 (3.3%) of the 30 had not released a version of the Eclipse product since the initial release, 11 (36.7%) had released *2 to 5* versions, 7 (23.3%) had released *6 to 10* versions, 5 (16.7%) had released *11 to 15* versions, 1 (3.3%) had released *16 to 20* versions, 0 (0.0%) had released *21 to 25* versions and 5 (16.7%) had released *more than 25* versions.

xi. *What is the size (estimate) of the latest version of your Eclipse product in terms of number of files (NOF)?*: 7 (23.3%) of the 30 had *less 100 files* for the latest version of the Eclipse product, 10 (33.3%) had *101–500* files, 2 (6.7%) had *501–1000* files and 11 (36.7%) had *more than 1000 files*.

xii. *How many hours/week do you dedicate updating your Eclipse product in relation to Eclipse SDK API usage?*: all the thirty respondents chose the option of *less than 10 hours*.

7.3.2.3 Awareness of Eclipse Interfaces and their Guidelines

The following list presents the frequency and percentage distribution of the cases of the variables that belong this category.

xiii. *Are you aware of the "Eclipse Provisional API Guidelines"?*: 15 of the respondents chose the option they are *this is the first time I have heard of them* of the *Eclipse provisional API guidelines* and the remaining 15 the option *I am already aware*.

xiv. *Do you follow "Eclipse Provisional API Guidelines" when developing your Eclipse product(s)?*: 14 (46.7%) of the 30 responded that *they do not know, because they do not know the guidelines*, 3 (10.0%) responded that they *never* follow the guidelines, 12 (40.0%) responded that they *sometimes* follow the guidelines and 1 (3.3%) responded that they *always* follow the guidelines.

xv. *Are you aware that Eclipse provides two types of interfaces (APIs and non-APIs)?*: 3 (10.0%) of the 30 responded that the survey *just informed* them that Eclipse provides two types of interfaces, APIs and non-APIs while 27 (90%) responded that they were *already aware* of the Eclipse APIs and the non-APIs.

7.3.2.4 Use of non-APIs

The following list presents the frequency and percentage distributions of the cases in the variables that belong to this category

xvi. *Do you use non-APIs?*: 21 (70%) of the 30 responded that they use non-APIs while the remaining 9 (30%) responded that they do not use non-APIs.

	Frequency (Percentage)				
Resp	nxtRL	rlANA	nwMV	nwBV	NO
Selected	20(66.7%)	8(26.7%)	12(40%)	8 (26.7%)	5(16.7%)
Not selected	10(33.3%)	22(73.3%)	18(60%)	22(73.3%)	25(83.3%)
Total	30	30	30	30	30

Table 7.3: Distribution and percentage of the responses on variable *testP*.

xvii. *If answer was "NO" in (xvi). Do you deliberately avoid using Eclipse non-APIs?*: 9 respondents answered this question, where 8 (88.9%) of the 9 responded that they *deliberately avoid using* non-APIs and 1 (11.1%) responded that he/she does *not deliberately avoid using* non-APIs.

xviii. *If answer was "YES" in (xvi). Do you deliberately use the non-APIs?*: 21 respondents answered this question, where 18 (85.7%) of the 21 responded that they *deliberately use* non-APIs and the remaining 3 (14.3%) do not deliberately use non-APIs.

7.3.2.5 Testing of Eclipse Product

The following list presents the frequency and percentage distributions of the cases in the variables that belong to this category.

xix. *On which Eclipse SDK release did you develop the first version of your Eclipse product?*: 1 (3.3%) of the 30 developed the first Eclipse product on SDK 1.0, 4 (13.3%) on SDK 2.0, 1 (6.5%) on SDK 2.1, 7 (23.3%) on SDK 3.0, 2 (6.7%) on SDK 3.1, 4 (13.3%) on SDK 3.2, 2 (6.7%) on SDK 3.3, 3 (10%) on SDK 3.4, 3 (10%) on SDK 3.5, and 1 (3.3%), each, on SDKs 3.6, 3.7, and 4.0m4.

xx. *Have you tested your Eclipse product on newer Eclipse SDK releases, betas or milestones?*: had multiple choice options where one was allowed to select all that applied. We have subdivided the variable into five sub-variables according to the choices. The sub-variables are *nxrRL* for option *Yes, on the next new release*, *rlANA* for option *Yes, on a release after the next or later*, *nwMV* for the option *Yes, on a new milestone release*, *nwBV* for the option *Yes, on a new beta version*, and *NO* for the option *NO*. Table 7.3 show the distribution and percentage of the sub-variables.

xxi. *If answer was "YES, the on next release and/or on a release after the next or later. Are you frequently testing you Eclipse product on new SDK releases?*: 23 respondents answered this question, where 12 (52.2%) of the 23 responded that *YES* they frequently test their Eclipse product on new SDK releases and 11 (47.8%) responded *NO*.

7.3.3 Quantitative Analysis of the Variables

In this section we present the quantitative analysis of the survey. First, we present the observed correlation between the answers of the pairs of questions whose answers can be expresses on an ordinal scale. From Table 7.1, only 12 of the 21 questions have answers that can be expressed on a ordinal scale. Variable names in Table 7.1 represent the questions.

Var	Correlation (Significance)									
	educa (i)	expsd (ii)	expjd (iii)	exped (iv)	tsize (viii)	upimp (ix)	NOV (x)	NOF (xi)	awAPg (xiii)	flAPg (xiv)
educa	–	.23(.11)	.22(.13)	**.32(.04)***	-.04(.42)	.02(.46)	.07(.35)	.04(.41)	**.39(.02)***	.28(.07)
expsd	.23(.11)	–	**.67(.00)****	**.72(.00)*****	.20(.14)	.25(.09)	.03(.43)	.20(.15)	.17(.18)	.21(.14)
expjd	.22(.04)	**.67(.00)****	–	**.72(.00)*****	.00(.48)	.07(.36)	-.06(.37)	.22(.13)	.11(.28)	.15(.22)
exped	**.32(.04)***	**.72(.00)*****	**.72(.00)*****	–	.19(.16)	**.40(.02)****	.13(.24)	**.38(.02)***	**.37(.02)***	**.32(.04)***
tsize	-.04(.42)	.20(.14)	.00(.49)	.19(.16)	–	.04(.41)	.05(.40)	**.62(.00)****	.09(.32)	.19(.15)
upimp	.02(.46)	.25(.09)	.07(.36)	**.40(.02)****	.04(.41)	–	**.37(.02)***	.20(.15)	.15(.22)	.18(.17)
NOV	.07(.42)	.03(.43)	-.06(.37)	.13(.24)	.54(.40)	**.37(.02)***	–	**.46(.01)****	.13(.24)	.14(.23)
NOF	.04(.42)	.20(.15)	.22(.13)	**.38(.02)***	**.62(.00)****	.20(.15)	**.46(.01)****	–	.15(.21)	.21(.13)
awAPg	**.39(.02)***	.17(.18)	.11(.28)	**.37(.02)***	.08(.33)	.15(.22)	.13(.24)	.15(.21)	–	**.88(.00)*****
flAPg	.29(.07)	.21(.14)	.15(.22)	**.32(.04)***	.18(.17)	.19(.15)	.18(.17)	.14(.23)	**.88(.00)*****	–
mean	3.910	5.290	4.690	3.430	2.200	2.260	3.400	2.630	1.510	2.060
std	1.292	1.250	1.409	1.290	1.232	0.741	1.818	1.239	0.507	0.998

Table 7.4: Matrix of observed Spearman's correlation coefficients and their significance. The values in bold indicate correlations that are significant at 95% confidence interval. By the rule-of-thumb, correlations of .70 and higher are considered very strong (***), .40 to .69—strong (**), .30 to .39—moderate (*), .20 to.29—weak, and .01 to .19—No or negligible relationship.

Table 7.4 presents a matrix of observed correlation coefficients and their significance. Variables *whours* and and *awAnNA* are not included in Table 7.4 since they do not correlate with any of the variables. Correlations that are significant at 95% confidence interval are bold-faced.

7.3.3.1 RQ1—Relationship between identified factors

For this research question, we discuss only significant correlations between pairs of variables related to use-of-Eclipse-interfaces. From Table 7.4, we observe the significant correlations between the following pairs of variables:

- *educa vs awAPg (.39*)*: We observe a moderate positive relationship between education and awareness of the existence of the Eclipse provisional API guidelines. This implies that the more educated a developer is, the more likely he/she will try to find guidelines/manuals. However, when we look at the results of years of education in Table 7.2, we observe 29 of the 30 have 4 and more years after secondary school. If we relate the years to level of education, then, ≥4 & <6, is equivalent to masters degree. The results of years of education therefore reveal that 96.7% of the respondents have a high level of education. The results of the relationship are surprising as they reveal that developers with higher level of eduction (beyond masters degree) are the ones who take time to find and read the manuals/guidelines. Why we find it surprising is because the majority of the respondents have attained a high level of education to realize the benefits of reading guidelines. Therefore, we would not expect a significant difference between the respondents in the different levels of education in knowing the existence of the guidelines.

- *educa vs exped (.32*)*: We observe a moderate positive relationship between education and experience as an Eclipse product developer. This shows that developers with many years of experience tend to be more educated. However, we were not able to find a plausible explanation of the observation.

- *exped vs upimp (.40**)*:We observe a strong positive relationship between experience as an Eclipse product developer and importance of updating the Eclipse products with new releases of the Eclipse SDK APIs. This implies that experienced Eclipse product developers tend to value updating their products with new SDKs.

- *exped vs NOF (.38*)*: We observe a moderate positive relationship between experienced product developers and NOF. This shows that experienced developers are involved in developing large Eclipse products.

- *exped vs awAPg (.37*)*: we observe a moderate positive relationship between experience of Eclipse product developers with awareness of the "Eclipse provisional API guidelines". This implies that less experience product developers are not aware of the guidelines.

- *tsize vs NOF (.62**)*: We observe a strong positive relationship between size of the development team and NOF. This shows that larger development teams are involved in large projects.

- *upimp vs NOV (.37*)*: We observe a moderate positive relationship between importance of updating and NOV. This implies that developers who value the importance

of updating their Eclipse products with new SDK interfaces, have released many versions of their products.

- *NOV vs NOF (.46**)*:We observe a strong positive relationship between NOV and NOF. This shows that products with many versions have a tendency to have more files in later releases of the product.
- *awAPg vs flAPg (.88***)*: We observe a very strong positive relationship between the awareness of the existence of the Eclipse provisional guidelines and following the guidelines. This shows that developers who are of the guidelines have a tendency follow them.

From the discussion above in the relationship between the pairs of variables, we make two main observations: First, we observe that experience of an Eclipse product developer correlates with 7 out of 10 variables as can be seen in Table 7.4. This implies that the experience of the Eclipse developer plays an important role in development/maintenance of the Eclipse products. One very important pair of correlation is *exped vs awAPg (.37*)*. Second, in Section 7.3.2 we reported that 50% of the respondents were not aware of the Eclipse API provisional guidelines. We now observe that level of education correlates strongly with awareness of the guidelines. Companies/organizations involved in developing Eclipse product and in general software development, should encourage developers to read product manuals/guidelines.

7.3.3.2 Exploratory Factor Analysis

For research questions RQ2 and RQ3, we employ exploratory factor analysis [82] to help us analyze the differences between the two groups of respondents in each of the research questions, respectively. Each of the research questions can be analyzed on the 10 variables whose answers are represented on a ordinal scale. Exploratory factor analysis helps in grouping the large number of variables into a small number of independent factors, i.e., groups of variables that are correlated with each other, on which each of the two identified group for research questions RQ2 and RQ3, respectively, can be compared. Exploratory factor analysis is a statistical method for investigating common but unobserved sources of influence (factors) in a collection of variables [32].

To perform factor analysis, we used software IBM SPSS Statistics 20. Principal component analysis (PCA) was used to extract the factors from the correlation matrix [82]. Equation 7.1 represents the relationship between the extracted factors and a given variable, where, *var* is a variable, β_1,...,β_N are the coefficients, also called *factor loadings*, for *factor1*,...,*factorN*, and U_{var} is the error. In addition, we used *Varimax* rotation method to provide us with a *simple structure* of the data. In a simple structure, each factor has large loadings in absolute value for only one of the variables.

After extracting the factors, to determine the number of factors to retain, we used the eigenvalue-greater-than-one criterion [82]. Eigenvalues explain the variance of the factors. We retained three factors that had eigenvalues greater than 1. The final percentage of variance explained by the 3 factors retained is 71%.

$$var = \beta_1(factor1) + ... + \beta_N(factorN) + U_{var} \tag{7.1}$$

Table 7.5 presents the communalities of nine of the variables (reasons for excluding variable *upimp* (ix) will be explained later). *Communality* of a variable is the proportion of variance explained by the common factors [82]. Communality ranges from 0 to 1, with

Variable	Initial	Extraction
educa (i)	1.000	.426
expsd (ii)	1.000	.746
expjd (iii)	1.000	**.825**
exped (iv)	1.000	.775
tsize (viii)	1.000	.563
NOV (x)	1.000	.374
NOF (xi)	1.000	**.874**
awAPg (xiii)	1.000	**.919**
flAPg (xiv)	1.000	**.836**

Table 7.5: Communalities using the principle component analysis extraction method

	Factor		
Variable	1	2	3
expjd (iii)	**.904**	.063	.054
expsd (ii)	**.855**	.118	-.028
exped (iv)	**.807**	.265	.233
awAPg (xiii)	.070	**.950**	.110
flAPg (xiv)	.103	**.889**	.188
educa (i)	.328	.552	-.128
NOF (xi)	.170	-.033	**.919**
tsize (viii)	.072	-.044	.746
NOV (x)	-.049	.158	.589

Table 7.6: Rotated factor matrix showing factor loadings on the variables. Extraction method–PCA, Rotation method–Varimax with Kaiser normalization.

0 indicating that the common factors do not explain any of the variance, and 1 indicating that all the variance is explained by the common factors. Communalities of .80 or higher are considered high [107] but uncommon with real data. More common magnitudes are low to moderate communalities of .40 to .70 [29]. Table 7.6 presents the rotated matrix between the factors and the variables after the factors have been sorted by the absolute values of loading.

As opposed to our small sample size of 30 subjects, the rule-of-thumb requires at least 100 subjects to perform factor analysis. Costello and Osborne [29] report that a large percentage of researchers report factor analysis using relatively small samples and that strict rules of sample size of exploratory factor analysis have disappeared. "The stronger the data, the smaller the sample can be for an accurate analysis. *Stronger data* in factor analysis means uniformly high communalities without *cross loading* (variables having high loadings in more than one factor), plus several variables loading strongly on each factor" [29]. Tabachnick and Fidell [101] report that .32 as a good rule-of-thumb for minimum loading of a variable. The authors further report that, researchers need to decide whether to drop the variable with low loading. In our analysis, the communality of the variable *upimp* (ix) was .269, the reason we decided to exclude it. Tabachnick and Fidell [101] further report that a cross loading variable is a variable that loads at .32 or higher on two or more factors. The researcher need to decide whether a cross loading variable should be dropped from the analysis.

From Table 7.5, we can observe that the variable communalities are all above .32 and four of the communalities are high (ones in bold). Furthermore, from Table 7.6 we can observe that only *eduac* (i) variable has cross loading on both *Factor 1* and *Factor 2*. The rest of the variables have uniformly high factor loadings on one factor and very low loadings on the remaining two factors. We decided not to drop the cross loaded variable *eduac* (i) in our analysis since we have few number of variables. From the observations, we can say that our data is relatively strong and therefore our small sample size of 30 subjects does not affect the accuracy of the results.

Looking at the factor loadings of the variables in Table 7.6, we can observe that factor 1 has high loadings (shaded in gray) on variables *expjd* (iii), *expsd* (ii), *exped* (iv) and *educa* (i), factor 2 has high loadings on variables *awAPg* (xiii), *flAPg* (xiv), and *educa* (i), and factor 3 has high loadings on *latvn* (xi), *tsize* (viii), and *verep* (x). Interpreting the factors based on the natural selection of variable loadings we could say that the conceptual meaning of factor 1 is related to *education and experience*, factor 2 is related the *education and API guideline awareness*, and factor 3 is related to *size of the plug-in*.

With the small number of factors that explain the collection of variables, we can now compare the two groups of respondents that use or do not use non-APIs. Recall in Section 7.3.2 we stated that 9 of the 30 respondents do not use non-APIs and the remaining 21 use non-APIs.

During factor extraction, we saved factor scores for each of the subjects using the regression method [82]. We use the factor scores in analysing research questions RQ2 and RQ3. To test for normality, we used *one-sample Kolmogorov-Smirnov* test at 95% confidence interval. We grouped the factor scores independently according the groups identified in RQ2, i.e. those that do not use non-APIs and those that use non-APIs (six normality tests, corresponding to two groups and three factors). Similarly, the factor scores are grouped independently according to the two groups identified in RQ3, i.e., those developing open-source and closed-source products (six normality tests, corresponding to open-source vs. closed-source products and three factors). Since the p values associated with the 12 tests were all above 0.32, we could not reject the normality hypotheses.

7.3.3.3 RQ2—Differences between Eclipse products that use and do not use non-APIs

We will start by stating the hypotheses for this research question:

- H_0: *Developers who use non-APIs and those who do not use non-APIs have the same average values on each of the identified factors.*
- H_a: *There is a difference on the average values of the identified factors between the developers that use and do not use non-APIs.*

Since we could not reject the normality hypotheses, we use the traditional t-test and the traditional 95% confidence level to investigate H_0. The test reveals significant difference between developers who use and do not use non-APIs on factors *Education and experience* and *Size of the plug-in* (p-values equal 0.003 and 0.026, respectively). However, developers who use and do not use non-APIs have no significant difference when it comes to *education and API guideline awareness* factor (p-value 0.702). Therefore, we reject the H_0 for *Education and experience* and *Size of the plug-in* and accept H_a, however, we do not have enough evidence to accept H_a for *education and API guideline awareness*.

Correlation (Significance)						
educa	expsd	expjd	exped	NOF	NOV	tsize
-.0(.78)	**.71(01)**	**.66(.02)**	**.77(00)**	**.68(00)**	.19(.48)	**.47(.08)**

Table 7.7: gamma correlation coefficients between the variables and the two groups of respondents that use and do not use non-APIs.

Furthermore, to identify how different the two groups are, we used the *gamma measure of association* [10] to quantify the relationship between the variables in the significant factors and the two groups. Since the gamma measure of association can only be applied to test the relationship between two variables on an ordinal scale, we impose an ordinal scale, as suggested in [10], on the two of respondents, i.e., those who do not use non-APIs (1) and those who use non-APIs (2).

The results in Table 7.7 reveal that compared to developers who do not use non-APIs, the developers who use non-APIs have more years of experience as software developers, Java developers and Eclipse developers, the latest versions of their Eclipse products have more files and their Eclipse development team is bigger.

In our previous studies in Chapters 4–5, we found that Eclipse products depending on at least one non-API are larger in terms of NOF and also have more versions (higher NOV) compared to Eclipse products that depend only on API. The current study confirms the previous findings on NOF but we do not have sufficient evidence for the differences on NOV for the two groups of Eclipse products. The possible reason for the findings on NOV is that, NOV is dependant on the Eclipse SDK on which the initial version of the product was developed. Recall that in Section 7.3.2 we reported that the first versions of the 30 Eclipse products were developed on 12 Eclipse SDK releases, where SDK 3.0 had the highest number of 7 products. The distribution of the 30 Eclipse products on the 12 SDK reduces the number of data points to statistically test NOV for the two groups of software systems.

The findings of research question RQ2 now reveals an answer for previously unanswered question in Chapter 4: "why ETPs that depend on at least one non-API are larger than those that depend on only APIs?". The reasons for the difference in size between the two groups of ETPs are related to two main human factors: 1) developers of the ETPs that use at least one non-API are more experienced as software/Java/Eclipse products developers, and 2) the Eclipse development team sizes are larger.

7.3.3.4 RQ3—Differences between open-source and closed-source Eclipse products

We first state hypotheses for this research question:

- H_0: *Developers of open-source and closed-source Eclipse products have the same average values on each of the identified factors.*
- H_a: *There is a difference on the average values of the identified factors between the developers of open-source and closed-source Eclipse products.*

Since we could not reject the normality tests on the factor scores of open-source and closed-source, we use the t-tests to perform the hypothesis testing. By performing three t-tests we have observed that at 95% confidence interval, there is no significant

#	**Question**
ix(a)	Please give reasons for your choice of importance of updating your Eclipse product
xiv(a)	If answer in (xiv) was "Sometimes" or "Never" follow guidelines. Why don't you always follow the guidelines?
xvii(a)	If answer was "NO" in (xvi) and answer was "YES" in (xvii). Please give an explanation of why you deliberately avoid the use of Eclipse non-APIs.
xviii(a)	If answer was "YES" in (xvi) and answer was "YES" in (xviii). Please give an explanation of why you deliberately use Eclipse non-APIs.
xxi(a)	If answer was not "NO" in (xx). Please name some of the challenges you have faced in testing your product with newer Eclipse SDK releases.

Table 7.8: Open-ended questions. Follow-up questions in Table 7.1 for the corresponding question numbers without the lower-case letters in brackets.

difference between developers of open-source and closed-source Eclipse products (p-values always exceeded 0.1). We therefore accept the H_0 that developers of open-source and closed-source Eclipse products have the same mean values on each of the identified factors.

The findings of research question RQ3 clear our doubts on the generalizability of the results of our previous studies in Chapters 2–6, that there is no significant difference between open-source and closed-source Eclipse products in terms of the factors we have considered.

7.3.4 Qualitative Analysis

To complement the quantitative analysis presented so far, we sought the opinions of the developers by asking the open-ended questions. Table 7.8 contains the questions that we asked in the survey. We analyze the answers questions qualitatively. Due to space limitations, we only discuss the common responses. The rest of the answers in the survey can be found in Appendix D.

7.3.4.1 Reasons for the developers choice of importance (question ix(a) in Table 7.8)

The answers were given according to the respondents' chosen option on how they value the importance of updating their Eclipse products, i.e., *not important*, *moderately important* or *very important*. Recall that in research question RQ1 (Section 7.3.3), we discussed the relationship between experience of Eclipse developers and importance of updating, i.e., *expev vs upimp (0.4**)*. This means that the respondents whose answers correspond to *very important*, are more likely to be experienced Eclipse product developers.

After examining the common answers grouped according to the respondents value importance of updating their product with new versions of Eclipse SDK, we observed that the reasons are based on the following:

- For those who chose the option *not important* and *moderately important*, the reasons are related to: 1) *use of basic APIs* that do not change very often, 2) *product finished* and no longer maintained, 3) taking care of *product functionality first* before they can update with Eclipse, and 4) new versions of the product have to *keep working with older versions* as well.

- For those who chose the option *very important*, the reasons are related to: 1) users of the product require compatibility with the latest release of the SDK, 2) getting

new features, 3) benefiting from simplified, improved functionality and performance enhancement of the evolved interfaces, 4) keeping up with the Eclipse release train to survive in time.

7.3.4.2 Reasons why developer do not always follow the guidelines - (question xiv(a) in Table 7.8)

The answers were given according to the respondents chosen option on whether they follow Eclipse API provisional guidelines, i.e., *never follow* or *sometimes follow*.

After examining the common answers grouped according to whether respondents never or sometimes follow the guidelines, we observed that the reasons were based on the following:

- For those who chose the option *never follow*, the reason is that they know that they are using unstable interfaces, but they are willing to update the code in new releases of the SDK.
- For those who chose the option *sometimes follow*, the reasons are related to, 1) sometimes they require functionality from non-APIs when they cannot find the functionality in the APIs, and 2) they use some useful non-APIs and try to get Eclipse org make them public.

7.3.4.3 Reasons why developers deliberately avoid the use of non-APIs - (question xvii(a) in Table 7.8)

The respondents that answered the question the gave reasons for deliberately avoiding the non-APIs that were related to, knowing/assuming that the non-APIs are unstable.

Recall that during the discussion of research question RQ2 we found that most of the respondents of this question are less experienced Eclipse product developers and their products are small in terms of NOF. It is possible, for these developers, that the benefits of using non-APIs are overshadowed by the instability of the non-APIs or the the developers do not require extensive functionality yet since their software products are still relatively small.

7.3.4.4 Reasons why developers deliberately use non-APIs - (question xviii(a) in Table 7.8)

The common answers to the question are as follows: 1) There is no API with necessary functionality whereas, re-implementation (copy & paste) of the functionality provided by the non-API would be more difficult to maintain. Re-implemented code misses out on improved quality of the non-API in the new releases of Eclipse. 2) Avoiding reinventing the wheel and avoiding possible pollution of my code (copying in Eclipse public licence (EPL) code, vs linking, can be very problematic). Old non-APIs are unlikely to disappear. 3) We use the non-APIs so as to expose them.

Recall that in the discussion of research question RQ2 we found that most of the respondents of this question are more experienced Eclipse product developers and their products are larger in terms NOF. This could possibly mean that with experience one gets to uncover the benefits of using the non-APIs that can be more important than their instability.

7.3.4.5 Challenges faced by developers in testing their products with newer Eclipse SDK releases (question xxi(a) in Table 7.8)

The challenges that developers face in testing their products in new SDK releases are related to: 1) drastic changes in Eclipse leading to the need to learn a large set of APIs to make the product function properly again (especially e4 APIs are a prime example), 2) a full regression testing required, and 3) manual testing of the product.

7.4 Threats

In this section, we discuss validity and reliability threats that may have affected our study.

7.4.1 Threats to Validity

As any other survey investigation, our findings may be subject to validity threats. We categorize the possible threats into construct, internal, and external validity.

Construct validity threats in our study mainly concern how the measurements were performed in the study. In particular, how questions were asked, the way they could be answered (e.g. open vs. closed questions), the scales used to codify the answers. As we discussed in Section 7.2, as well as our experience in our previous empirical research about Eclipse API usage, we took time to design the questionnaire. Recall that in Section 7.2, the questionnaire was reviewed internally and externally to avoid ambiguous and biased questions. The way the survey was executed poses a threat of respondent selection bias. Since survey was deployed on the Eclipse twitter account, the survey could have missed out Eclipse product developers that do not follow the Eclipse twitter account and those who are not on twitter at all. Construct validity of our findings can also be threaten if a respondent filled in the questionnaire more than once. We tried to minimize this threat during the design of the questionnaire by asking a compulsory question of the name the Eclipse product the respondents are involved in developing. All the 30 respondents were all developed different plug-ins. Since we promised anonymity of the respondents, the plug-in names and contact addresses in the survey data we shared [1] were excluded.

Internal validity threats of our study concern confounding factors that may affect the outcomes of the results and difficult to control. Confounding factors like human memory, knowledge, motivation, personality and the fact that people want to be seen in a good light are possible threats to internal validity with questionnaires [89, 102]. It is impossible to know whether the respondents answer truthfully, or whether other effects like pressure of the projects they are working on affect their response. To limit these threats, we tried to motivate the respondents by explaining the importance of the survey where we also promised to reveal the results of the survey including the data we collected. Furthermore, to minimize this threat, we employed both quantitative and qualitative methods and in some cases we employed both methods at the same time.

External validity threats concerns the extent to which our findings can be generalized. Our findings on Eclipse API usage may be threatened by external validity since they are based on only 30 respondents. However, we tried to minimize this threat by the providing a comprehensive questionnaire comprising five sections of different questions. Furthermore, the threat is minimized by the different groups of respondents, for example open-source and closed-source developers, developers who use and do not use non-APIs.

7.4.2 Threats to Reliability

This section discusses the most relevant threats to reliability using the internal consistency reliability [71]. Internal consistency reliability applies not to one item but a group of items that are thought to measure different aspects of the same concept. Litwin [71] interprets a correlation coefficient greater than 0.7 to be strong between pairs of the groups of items. In our scenario, internal consistency reliability can be observed between variables that measure experience of the developer, i.e., *expsd*, *expjd*, and *exped*, where we have correlation of around 0.7 between pairs of the variables. Another pair of variables measuring the different aspects of the same concepts are *awAPg* and *flAPg* which measure awareness and following of the Eclipse guideline. Between the pair we have a correlation of 0.88. The high values or correlation reveal how the reliability threat is minimized.

7.5 Related Work

In the previous sections, we implicitly discussed how the current work relates to our previous work Chapters 2–6. In general, our previous work is based on empirical analysis of the co-evolution of the Eclipse SDK framework and its third-party plug-ins (ETPs). During the evolution of the framework, we studied how the changes in the Eclipse interfaces used by the ETPs, affect compatibility of the ETPs in forthcoming framework releases. We only used open-source ETPs in the study and the analysis was based on the source code. The current study, based on analysis of the survey, complements our previous studies by including closed-source ETPs and taking into account human aspects.

Besides our own work, the current study is related to the study of Oram and Wilson [84], where the authors analysed the difference between open-source products and Commercial products. The authors examined code quality metrics that were collected from four large industrial-scale operating systems: FreeBSD, Linux, OpenSolaris, and the Windows Research Kernel. No significant across-the-board code quality differences were found between these four systems. Similarly to this work, we also did not find significant differences between open-source and closed-source ETPs.

7.6 Conclusions

This chapter reports on a survey, where 30 Eclipse product developers took part. The aim of the survey was to achieve a clear picture on the state-of-the-practice of the Eclipse interface usage by Eclipse product developers. The lessons learned from the findings of the survey can be used to improve the development and maintenance of Eclipse products.

The results from the analysis reveal a number of findings: First, we observed that the Eclipse product developers' experience plays a very important role in the development and maintenance of the Eclipse products. Experienced Eclipse product developers are more aware of the benefits of updating their Eclipse products with new releases of Eclipse, for example they are aware that evolved Eclipse interfaces are simplified, they have improved functionality and have performance enhancement. Second, we observed that developers with a level of education of up to master degree have a tendency of not reading product manuals/guidelines. Third, we observed that compared to Eclipse products that do not use non-APIs, Eclipse products that use non-APIs are bigger and have larger Eclipse development teams. Furthermore, the developers are more experienced as software developers, as Java developers and as Eclipse product developers. The reasons for using

the non-APIs are that while there could be a possibility that instability of the non-APIs overshadows the benefits that the non-APIs offers to less experienced developers, the experienced developers have uncovered benefits of using non-APIs that are more important than the of instability of the non-APIs. For example, when there is no API with the functionality they require, one alternative is to write their own API from scratch. The other simplest alternative is to re-implement (copy & paste) the code of the non-API. The experienced Eclipse product developers state that re-implemented code is difficult to maintain and also misses out on the improvements of the non-APIs in new versions of Eclipse. Moreover, they are aware that if the non-API has been there for years, it is unlikely to disappear. The developers observation of the stability of the old non-APIs coincides with our finding in [25] that old non-APIs relatively more stable than newly introduced non-APIs. Finally, we have observed that there are no significant differences between open-source and closed-source Eclipse products in terms of awareness of Eclipse guidelines and interfaces, Eclipse product size and updating of Eclipse product in the new SDK releases.

The survey analysis was based on only 30 Eclipse-based application developers that took part. In a follow-up study, one can collect more subjects and investigate some of the aspects that we did not investigate extensively. For example, one may consider challenges faced by application developers in testing their software systems in new SDK releases.

Chapter 8

Conclusions

In this thesis, we have conducted a series of empirical analyzes to investigate the co-evolution of the Eclipse framework and its third-party plug-ins. This chapter summarizes the contributions of this thesis and provides directions for further research.

8.1 Contributions

Today, when constructing software systems, many developers build their systems on top of frameworks. The Eclipse SDK framework is a popular and widely adopted framework that has been evolving for over a decade. The Eclipse SDK framework has both stable and supported APIs (*good interfaces*) and unstable, discouraged and unsupported non-APIs (*bad interfaces*). However, despite being discouraged by Eclipse, from our study we found out that the usage of *bad interfaces* when developing plug-ins is not uncommon. In this thesis, by means of a series of empirical studies, we quantify/qualify the challenges faced by Eclipse third-party plug-in developers when using the interfaces provided by the Eclipse SDK framework. Furthermore, we propose solutions to the identified challenges, such as changes in development strategy to both interface providers and interface users.

We approached the investigation in this thesis by posing a number of research questions. The main research question covered in this thesis is formulated as follows:

RQ: *What are the challenges faced by framework-based application developers in co-evolving their application to the framework they reuse?*

This research question is divided into five more specific research questions, and each of these questions is addressed in one of the chapters of this thesis.

8.1.1 Evolution of ETPs

The first of these specific questions concerns the evolution of the *framework-based* applications with respect to the known *general* evolution of E-Type systems [14, 65, 113] and is formulated as follows.

RQ1: *How does the evolution of ETPs with respect to the interfaces they use from the framework compare to the known general evolution of E-Type systems?*

We addressed RQ1 in Chapter 2. Lehman's laws of software evolution [65] were employed to study the evolution of E-Type systems. We investigated whether the constrained evolution of the Eclipse third-party plug-ins (ETPs) conforms to the laws, i.e., we studied the evolution of the ETPs based on the interfaces they reuse from the Eclipse SDK framework.

We investigated the empirical evidence of 7 of the 8 Lehmans' evolution laws. Our findings confirm the laws of *continuing change*, *increasing complexity*, *self regulation*, and *continuing growth* when metrics related to dependencies between the plug-ins and the Eclipse architecture are considered. We could not validate the *conservation of familiarity* and *conservation of organizational stability* law, and the results for the *declining quality* law were inconclusive. We could not comment on the law of *feedback system* since we did not have the data to measure the law. Our overall observation is that the trends observed for constrained evolution are similar to those presented by earlier researches on the general evolution of software systems [14, 65, 113].

8.1.2 Eclipse API usage: the Good and the bad

During the analysis of RQ1, we did not distinguish between the *good interfaces* and the *bad interfaces*. Next, we were interested in investigating the extent of ETPs dependency on the two types of Eclipse interfaces. We therefore formulated the second research question given below.

RQ2: *To what extent do ETPs depend on the two types of interfaces the Eclipse SDK framework provides?*

Research question RQ2 was addressed in Chapter 4. To answer this question, we conducted an empirical investigation based on a total of 512 Eclipse third-party plug-ins, altogether having a total of 1,873 versions. We discovered that 44% of the 512 analyzed ETPs depends on *bad interfaces* and that developers continue to use *bad interfaces* in the new versions of their plug-ins. The empirical study also shows that plug-ins that use or extend at least one *bad interfaces* are larger and use more functionality from Eclipse than those that only use *good interfaces*. Furthermore, the findings show that the ETPs use a diverse set of *bad interfaces*.

8.1.3 Survival of ETPs

The findings of RQ2 raised further questions that we decided to investigate. We wanted to get additional insights on how the ETPs are affected while using the interfaces as the Eclipse SDK framework evolves. For this reason, the following research question was formulated.

RQ3: *How does the compatibility of ETPs that depend solely on good interfaces compare to that of ETPs that depend on at least one bad interface in new Eclipse SDK releases?*

Research question RQ3 is addressed in Chapter 5. In answering this question, we reused the data that was also used for RQ2. We investigated the survival of the ETPs in

the different Eclipse SDK releases. We defined survival of a version of an ETP compatible with a given SDK release as the number of new SDK releases it can successfully compile with. We made the following observations:

1. We discovered that the majority of ETPs do not produce new versions beyond the first year of release.

2. As stated by Eclipse, we confirm that indeed APIs are stable over subsequent Eclipse releases that do not involve API-breaking changes. We further confirm that non-APIs are indeed unstable. The observation on the stability of the Eclipse interfaces is based on the impact of the compatibility of the ETPs that use these interfaces in new Eclipse SDK releases. ETPs that only depend on *good interfaces* almost never fail in the subsequent Eclipse SDK releases and ETPs that use bad interfaces have a very high failure rate in new SDK releases. Furthermore, we observed that, ETPs that depend more on old *bad interfaces* and less on newly introduced *bad interfaces*, have a very high forward source compatibility success rate.

3. We observed that the reason why the non-APIs are being eliminated from the ETPs' source code is, because (ETP developers believe) these non-APIs will cause incompatibilities when a version of the ETP is ported to a new SDK release.

4. When eliminating the use of problematic non-APIs in the ETPs source code, we have observed that developers perform one of the following things: i) build their own API that has the same functionality as the *bad interfaces*, ii) find similar functionality offered by *good interfaces* in the SDK, iii) completely eliminate the entities in the ETP source code that uses the functionality from the *bad interfaces* and iv) when a non-API matures into *good interfaces*, developers replace the functionality of the *bad interfaces* with the new *good interfaces*.

There are several implications to the findings of RQ3. First, this information provides Eclipse SDK developers with feedback on the current use of APIs and *bad interfaces* in ETPs as opposed to the expected use. The feedback can be used for planning improved services in new releases of SDKs. Second, developers should avoid the unstable *bad interfaces* as much as possible since they are the major cause of incompatibilities in new SDK releases. Third, developers who have to use *bad interfaces* should use those introduced in earlier releases of Eclipse since we observed that they are more stable. Fourth, ETP developers should avoid re-implemented (copy & paste) code of the *bad interfaces* since it can be difficult to maintain and the (copy & paste) code does not benefit from improvements in the *bad interfaces* in new releases of the SDK. Finally, in addressing RQ2 we discovered that a large number of developers use *bad interfaces* and in addressing RQ3, we observed that *bad interfaces* influence the survival of ETPs. Hence, it is equally important for the framework developers to document the changes in both the *bad interfaces* and *good interfaces*.

8.1.4 Compatibility prediction of ETPs in new SDKs

The findings of RQ3 have revealed compatibility problems that both the ETP users and developers face when using these ETPs in forthcoming SDK releases. We wanted to continue our study by proposing a solution to this problem. For this reason, we formulated our next research question.

RQ4: *Can a prediction model based on the usage of the Eclipse interfaces in the ETPs help in predicting the compatibility of an ETP in a new Eclipse SDK release?*

Research question RQ4 was addressed in Chapter 6. While investigating RQ3, we observed that *good interfaces* do not cause compatibility problems in forthcoming SDK releases that do not involve API breaking changes. We also observed that *bad interfaces* are prone to change in forthcoming SDK releases. Therefore, in analysing RQ4 we only focused on *bad interface*. We carried out an empirical investigation on 11 Eclipse SDK releases (1.0 to 3.7) and 288 ETPs.

The study was based on two main goals. First, to determine the relationship between the age of *bad interface* used by an ETP and the compatibility of the ETP. We found that ETP that use only old *bad interfaces* have a high chance of compatibility success in new SDK releases compared to those that use at least one newly introduced *bad interface*. Second, our goal was to build and test a predictive model for the compatibility of an ETP, supported in a given SDK release in a newer SDK release. Our findings produced 35 statistically significant prediction models. The results from model testing indicate high values of up to 100% of precision and recall and up to 98% of accuracy of the predictions. Finally, despite the fact that SDK releases with *API breaking changes*, i.e., 1.0, 2.0 and 3.0, have got nothing to do with *bad interfaces*, our findings reveal that *bad interfaces* introduced in these releases have a significant impact on the compatibility of the ETPs that use them.

8.1.5 Eclipse interface usage by the developers of ETPs

The findings from the Eclipse interface usage in research question RQ1—RQ4 were all based on the source code analysis. These findings left us with some unanswered questions for example, *why developers use and continue to use bad interfaces?*, *Are our findings generalizable to closed source products?* We therefore introduced a different view point in investigating the Eclipse interface usage in research question RQ5, i.e., based on state-of-practice of Eclipse interface usage by the developers. We discuss the findings of RQ1—RQ4 are discussed in comparison to the findings of RQ5.

RQ5: *What is the state-of-practice of Eclipse interface usage by the developers of ETPs?*

Research question RQ5 was addressed in Chapter 7. In addressing RQ5, we conducted a survey on Eclipse interface usage where 30 Eclipse product developers took part. The results from the analysis reveal a number of findings:

1. We observed that the Eclipse product developers' experience plays an important role in the development and maintenance of the Eclipse products. Experienced Eclipse product developers are more aware of the benefits of updating their Eclipse products with new releases of Eclipse. For example they are aware that evolved Eclipse interfaces are simplified, and they have improved functionality and performance enhancement.

2. We also observed that developers with a level of education of up to master degree have a tendency of not reading product manuals/guidelines.

3. We observed that compared to Eclipse products that do not use *bad interfaces*, Eclipse products that use *bad interfaces* are bigger and have larger Eclipse development teams. Furthermore, the developers are more experienced as software developers, as Java developers, and as Eclipse product developers.

4. We also observed that the reasons for using the *bad interfaces* are that while there could be a possibility that instability of the *bad interfaces* overshadows the benefits that the *bad interfaces* offer to less experienced developers, the experienced developers have uncovered benefits of using *bad interfaces* that are more important than the of instability of the *bad interfaces*. For example, when there is no *good interface* with the functionality they require, one alternative is to write their own API from scratch. The other simplest alternative is to re-implement (copy and paste) the code of the *bad interfaces*. The experienced Eclipse product developers state that this re-implemented code is difficult to maintain and also misses improvements of the *bad interfaces* in new versions of Eclipse. Moreover, they are aware that if the *bad interface* has been there for years, it is unlikely to disappear. The developers observation of the stability of the old *bad interfaces* coincides with our finding in research question RQ4 that old *bad interfaces* are relatively more stable than newly introduced *bad interfaces*.

5. Finally, we have observed that there are no significant differences between open-source and commercial Eclipse products in terms of awareness of Eclipse guidelines and interfaces, Eclipse product size and updating of Eclipse products in the new SDK releases.

In the thesis, we carried out empirical studies on the evolution of the Eclipse framework ecosystem. The methodology used to carry out the empirical studies is based on the guidelines proposed by Kitchenham et al. [57]. In the studies, we have used *a scientific method*, i.e., a body of techniques for investigating a phenomena [15]. Complexity of the phenomenon we have studied required application of such methods as regression analysis, exploratory factor analysis, Student's t-test, Mann-Whitney U test, and logistic regression. As part of future work (see Section 8.2) we propose designing a standard methodology for carrying out empirical studies on evolution of software ecosystems. The methodology should build on and extend the methodology used in this thesis.

8.2 Future Work

In this thesis we have addressed the challenges faced by framework-based application developers in evolving their applications. There are a number of ways in which the research presented in this thesis can be extended:

1. The investigation of the stability of the *bad interfaces* (addressed in Chapters 5 and 6) is based on an external artifact, i.e., the impact the *bad interfaces* have on software systems that use them. In a follow-up study one can carry out an investigation by measuring the internal artifacts of the *bad interface*, e.g., relationship between age of the *bad interfaces* and changes in the signature of the *bad interface*.

2. In testing the models, based on the methodology we used to collect testing samples, we had very few cases of ETPs that were incompatible. In some models, the percentage of the incorrectly predicted ETPs was high for the few incompatible cases considered. In a follow-up study one can collect more samples to test the models. One can also investigate the possibility of combining predictors to make better predictions (e.g., using a non-linear combination of predictor variables).

3. Our studies in Chapters 5 and 6 were based on source compatibility. In a follow-up study, one can extend our work by considering binary compatibility and runtime compatibility.

4. The models presented in Chapter 6 can be used to make interpolation compatibility predictions, i.e., predictions can only be made on SDK releases that were used to train the models. In a follow-up study one can build extrapolation prediction models, i.e., making predictions on an SDK that was not used in building the model.

5. A tool based on the prediction models can be built, which can be used by framework-based application developers or users. Users can use the tool to predict compatibility of the application they are using in case they want to get updates of a newer SDK release. Developers can use the tool to give them pointers in their source code where very unstable interfaces are used.

6. In Chapter 7 the survey analysis was based on only 30 Eclipse-based application developers that took part. In a follow-up study, one can collect more subjects and investigate some of the aspects that we did not investigate extensively. For example, one can/should consider challenges faced by application developers in testing their software systems in new SDK releases.

7. As stated in Chapter 1, the Eclipse framework is a large and complex software system used by thousands of application developers. The current Eclipse framework release, Eclipse Juno, has got 72 project teams by 445 open source committers on 55 million lines of code, and the participation of 40+ Eclipse member companies [7]. The framework promises benefits such as reduced costs, higher quality, and rapid application development. However, the framework is not free of costs. Before the framework can be exploited, application developers have to spend time reading, locating, and comprehending code so as to use the functionality the framework provides. In the survey we conducted (survey data can be found[1]), we found out that most developers find the functionality from Eclipse they are interested in, manually (25 of the 30 manually locate the functionality and 5 of the 30 use a semi-automated tool). To this end, one can develop a plug-in that can be integrated in the Eclipse IDE to aid developers in finding the functionality they are interested in.

8. In Chapter 7 we reported that developers use bad interfaces because there are no good interfaces with the necessary functionality they require. Since the Eclipse framework is a large complex open-source software system with a large number of committers, it is possible that good interfaces located in a different subproject could be offering similar functionality as the bad interfaces. To this end, one can carry out an investigation to find out good interfaces that have similar functionality as the bad interfaces and recommend them to developers. The problem of identifying good interfaces having similar functionality as the bad interfaces is related to type 4 clone detection [72].

9. Our work can further be replicated on other framework ecosystems to compare the findings with our findings in this thesis. To study another software framework ecosystem, one would first understand the naming convention of the interface during the evolution of these frameworks. For example, in NetBeans [104] the naming

[1] http://www.win.tue.nl/~jbusinge/CSMR13/index.html

convention of the interfaces include: `friend` that is used for features accessible to specific components in the framework, `devel` (under development) is a name for a contract that is expected to become a stable API, `stable` interfaces are those that have received a final state and the maintainers are ready to support it forever and never change them incompatibly, and `official` are stable ones and also packaged into one of NetBeans official namespaces.

10. Finally, one can/should consider designing a standard methodology for carrying out empirical studies on evolution of software ecosystems by building on and extending the methodology used in this thesis.

To summarize, the research conducted in this thesis aims at quantifying/qualifying the challenges faced by software developers building systems on top of application frameworks, in general, and specifically, Eclipse. We have observed that the use of bad Eclipse interfaces, i.e., unstable, discouraged and unsupported interfaces, results in incompatibilities of the systems built on top of Eclipse and the new releases of Eclipse. Despite major maintainability risks inherent to the use of bad interfaces, developers prefer to use them due to uniqueness of the functionality provided by these interfaces. Furthermore, we have observed that older bad interfaces are less likely to introduce incompatibilities. We have developed a statistical model predicting the likelihood of a system to remain compatible with a new release of Eclipse. Using this model both the system developers and the system users can decide whether to upgrade Eclipse to a new release. Finally, we have proposed a number of directions in which our work could/should be extended.

Bibliography

[1] http://www.win.tue.nl/~jbusinge/CSMR13.

[2] Architectural patterns and styles. http://msdn.microsoft.com/en-us/library/ee658117.aspx.

[3] Eclipse Marketplace. http://marketplace.eclipse.org, Consulted on October, 2012.

[4] Leveraging Object-Oriented Frameworks. http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/leveragingoo.pdf, Consulted on October, 2012.

[5] Provisional API guidelines. http://wiki.eclipse.org/Provisional_API_Guidelines, Consulted on October, 2012.

[6] Sourceforge. http://sourceforge.net, Consulted on June 01, 2010.

[7] Eclipse juno release train has arrived. http://www.eclipse.org/org/press-release/20120627_junorelease.php, 2012. Consulted on March 28, 2013.

[8] Eclipse modeling framework project (EMF). http://www.eclipse.org/modeling/emf/, 2012.

[9] J2EE standard tools project (JST). http://www.eclipse.org/webtools/jst/main.php, 2012.

[10] Alan Agresti. *Categorical Data Analysis*. John Wiley & Sons, Inc., 2002.

[11] John Arthorne, Tom Eicher, Markus Keller, and David Williams. Version numbering. http://wiki.eclipse.org/Version_Numbering, 2009. Consulted on October 05, 2011.

[12] Iman Attarzadeh and Siew Hock Ow. Software development effort estimation based on a new fuzzy logic model. *International Journal of Computer Theory and Engineering*, 1(4):1793–8201, 2009.

[13] N. C. Barford. *Experimental measurements : precision, error and truth.* Chichester : Wiley, 1985.

[14] Evelyn J. Barry, Chris F. Kemerer, and Sandra Slaughter:. How software process automation affects software evolution: a longitudinal empirical. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(1):1–31, February 2007.

[15] Victor R. Basili. The experimental paradigm in software engineering. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 3–12, London, UK, UK, 1993. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=647362.725507`.

[16] Karl Beecher, Andrea Capiluppi, and Cornelia Boldyreff. Identifying exogenous drivers and evolutionary stages in FLOSS projects. *Journal of Systems and Software*, 82:739–750, 2009.

[17] James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the 9th International Symposium on Software Metrics*, 2003.

[18] Azad Bolour. Notes on the Eclipse plug-in architecture. `\http://www.eclipse.org/articles/Article-Plug-in-architecture/`, 2003. Consulted on October, 2012.

[19] Jan Bosch, Peter Molin, Michael Mattsson, and PerOlof Bengtsson. Object-oriented framework-based software development: problems and experiences. *ACM Computing Surveys (CSUR)*, 32(3), March 2000.

[20] Jan Bosch, Peter Molin, Michael Mattsson, and PerOlof Bengtsson. Object-oriented framework-based software development: problems and experiences. *ACM Computing Surveys*, 32, 2000.

[21] John Businge, Mark van den Brand, and Alexander Serebrenik. Eclipse API usage: The good and the bad. *Software Quality Journal*, 2013. Accepted.

[22] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. An empirical study of the evolution of Eclipse third-party plug-ins. In *ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 63–72, 2010.

[23] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Eclipse API usage: The good and the bad. In *Sixth International Workshop on Software Quality and Maintainability*, pages 55–63.

[24] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Eclipse API usage: the good and the bad. Computer Science reports 11-15, Technische Universiteit Eindhoven, 2011.

[25] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 164–173, 2012.

[26] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Survival of Eclipse third-party plug-ins. In *28th IEEE International Conference on Software Maintenance*, 2012.

[27] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Analyzing the Eclipse API usage: Putting the developer in the loop. In *17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, 2013. Accepted.

[28] Edward Colbert. The object-oriented software development method: a practical approach to object-oriented development. In *Proceedings of the conference on Tri-Ada '89: Ada technology in context: application, development, and deployment*, pages 400–415. ACM, 1989.

[29] Anna B. Costello and Jason W. Osborne. Best practices in exploratory factor analysis: Four recommendations for getting the most from your analysis. *Practical Assessment, Research and Evaluation*, 10:173–178, 2005.

[30] Kevin Crowston and Hala Annabi. Information systems success in free and open source software development: Theory and measures. In *Software Process: Improvement and Practice*, pages 123–148, 2006.

[31] Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. In *Proceedings of the 24th International Conference on Information Systems*, pages 327–340, 2003.

[32] Robert Cudeck. Exploratory factor analysis. In Howard E.A. Tinsley and Steven D. Brown, editors, *Handbook of Applied Multivariate Statistics and Mathematical Modeling*, pages 265–296. 2000.

[33] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *Journal ACM Transactions on Software Engineering and Methodology*, 20:19:1–19:35, 2011.

[34] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software bertillonage: finding the provenance of an entity. In *8th Working Conference on Mining Software Repositories*, pages 183–192, 2011.

[35] Thomas R. Dean, Massamiliano Di Penta, Kostas Kontogiannis, and Andrew Walenstein. Clone detector use questions: A list of desirable empirical studies. In *Duplication, Redundancy, and Similarity in Software*, 2007.

[36] Scott Delap. Understanding how Eclipse plug-ins work with OSGi. `http://www.eclipse.org/articles/`, 2006.

[37] Jim des Rivières. How to use the Eclipse API. `http://www.eclipse.org/articles/`, 2001. Consulted on January 01, 2011.

[38] Jim des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/Evolving_Java-based_APIs`, 2007. Consulted on June 01, 2010.

[39] M. Di Penta, L. Cerulo, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *24th IEEE International Conference on Software Maintenance*, pages 217 –226, 2008.

[40] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring: Research articles. *Journal of Software Maintenance and Evolution*, 18:83–107, 2006.

[41] Mikhail Dmitriev. Language-specific make technology for the Java programming language. In *OOPSLA '02 Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 373–385, 2002.

[42] Regina C. Elandt-Johnson and Norman Lloyd Johnson. *Survival models and data analysis*. Wiley series in probability and statistics. 1999.

[43] Michael English, Chris Exton, Irene Rigon, and Brendan Cleary. Fault detection and prediction in an open-source software project. In *PredictOr Models in Software Engineering*, pages 17:1–17:11, 2009.

[44] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997. URL: `http://doi.acm.org/10.1145/262793.262798`.

[45] The Eclipse Foundation. Application frameworks. `http://www.eclipse.org/home/categories/index.php?category=frameworks`, 2012. Consulted on November 13, 2012.

[46] Michael W. Godfrey, Daniel M. German, Julius Davies, and Abram Hindle. Determining the provenance of software artifacts. In *IWSC'11*, pages 65–66, 2011.

[47] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 131–142. IEEE Computer Society Press, March 2000.

[48] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *ESEM'10*, pages 11:1–11:10, 2010.

[49] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, October 1997. URL: `http://doi.acm.org/10.1145/263700.264352`, `doi:10.1145/263700.264352`.

[50] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[51] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite: a measurement theory perspective. *Software Engineering, IEEE Transactions on*, 22(4):267 –271, apr 1996.

[52] J. L. Hodges and E. L. Lehmann. Estimates of location based on rank tests. *The Annals of Mathematical Statistics*, 34(2):598–611, 1963.

[53] Reid Holmes and Robert J. Walker. Informing Eclipse API production and consumption. In *OOPSLA'07*, pages 70–74, 2007.

[54] Ayelet Israeli and Dror G. Feitelson. The Linux kenel as a case study in software evolution. *The Journal of System and Software*, 83(3):485–501, March 2010.

[55] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Object-Oriented Programming*, 1(2), 1988.

[56] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical report, Urbana-Champaign 91-1696, University of Illinois, 1991.

[57] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1027796.

[58] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *9th IEEE International Working Conference on Source Code Analysis*, pages 168–177, 2009.

[59] Stefan Koch. Software evolution in open source projects–a large-scale investigation. *J. Softw. Maint. and Evol.: Res. and Pract.*, 19(6):361–382, November 2007.

[60] Dino Konstantopoulos, John Marien, Mike Pinkerton, and Eric Braude. Best principles in the design of shared software. In *COMPSAC'09*, pages 287 –292, 2009.

[61] Charles W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, June 1992. doi:10.1145/130844.130856.

[62] Ralf Lammel, Rufus Linke, Ekaterina Pek, and Andrei Varanovich. A framework profile of .net. *Reverse Engineering, Working Conference on*, 0:141–150, 2011. doi:http://doi.ieeecomputersociety.org/10.1109/WCRE.2011.25.

[63] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1317–1324, 2011.

[64] Meir M. Lehman. Evolution and Conservation in the large Program Life Cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[65] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. doi:10.1109/PROC.1980.11805.

[66] Meir M. Lehman. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[67] Meir M. Lehman. Efficient Feature Selection via Analysis of Relevance and Redundancy. *Journal of Machine Learning Research*, 5:1205–1224, 2004.

[68] Meir M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press, London, 1985.

[69] Meir M. Lehman, Dewayne E. Perry, and Juan F. Ramil. On evidence supporting the feast hypothesis and laws of software evolution. In *Proc. Metrics'98*, pages 84–88. IEEE Computer Society Press, 1998.

[70] Meir M. Lehman, Juan F. Ramil, Paul Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS*, pages 20–32, 1997.

[71] Mark S. Litwin. *How to Measure Survey Reliability and Validity*. SAGE Publications Inc., 1995.

[72] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 107–, Washington, DC, USA, 2001. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=872023.872542`.

[73] Robert Martin. OO design: An analysis of dependencies, 1994. `http://www.objectmentor.com/resources/articles/ood-metrc.pdf`, Consulted on June 01, 2010.

[74] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *14th European Conference on Software Maintenance and Reengineering*, pages 107–116, 2010.

[75] Tom Mens, Juan Fernández-Ramil, and Sylvain Degrandsart. The evolution of Eclipse. In *Proc. 24th Int'l Conf. on Software Maintenance*, pages 386–395, October 2008.

[76] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining API popularity. In *Academic & Industrial Conference Practice and Research Techniques*, pages 173–180, 2010.

[77] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[78] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu, and Stéphane Ducasse. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 2012.

[79] Maurizio Morisio, Daniele Romano, and Corrado Moiso. Framework based software development: Investigating the learning effect. In *Proceedings of the 6th International Symposium on Software Metrics*, METRICS '99, pages 260–268, Washington, DC, USA, 1999. IEEE Computer Society.

[80] Simon Moser and Oscar Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9):45–51, 1996.

[81] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 302–321, 2010.

[82] Marija J. Norušis. *SPSS 16.0 Guide to Data Analysis*. Prentice Hall Inc., Upper Saddle River, NJ, 2008.

[83] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, 1992.

[84] Andy Oram and Greg Wilson. Making software: What really works, and why we believe it, 2010.

[85] Jacek Pospychala. PHP development tools. `http://www.eclipse.org/projects/project.php?id=tools.pdt`, 2012.

[86] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[87] R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2010.

[88] Austen Rainer and Stephen Gale. Evaluating the quality and quantity of data on open source software projects. In *International Conference on Open Source Development, Adoption and Innovation*, pages 11–15, 2005.

[89] Colin Robson. *Real World Research*. Prentice Hall, 2011.

[90] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *25th IEEE International Conference on Software Maintenance*, pages 303–312, 2011.

[91] Gerd K. Rosenkranz. A note on the Hodges-Lehmann estimator. *Pharmaceutical statistics*, 9(2):162–167, April 2010.

[92] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival analysis on the duration of open source projects. *Information and Software Technology*, 52:902–922, 2010.

[93] Wilhelm Schäfer, Rubèn Prieto-Diaz, and Masao Matsumoto. *Software Reusability*. Ellis Horwood Ltd., 1994.

[94] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *ISESE'06*, pages 18–27, 2006.

[95] Charles M. Schweik. Identifying success and abandonment of FLOSS commons: A classification of Sourceforge.net projects. *The European Journal for the Informatics Professional VII*, 2, 2007.

[96] Charles M. Schweik, Robert C. English, Meelis Kitsing, and Sandra Haire. Brooks' versus Linus' law: an empirical test of open source projects. In *Proceedings of the 2008 international conference on Digital government research*, pages 423–424, 2008.

[97] Gehane M. K. Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E. Hassan, and Ying Zou. Studying the impact of clones on software defects. In *17th Working Conference on Reverse Engineering*, pages 13–21, 2010.

[98] Alexander Serebrenik, Serguei A. Roubtsov, and Mark G. J. van den Brand. D_n-based architecture assessment of java open source software. In *The 17th IEEE International Conference on Program Comprehension*, pages 198–207. IEEE Computer Society, 2009.

[99] Alexander Serebrenik and Mark G. J. van den Brand. Theil index for aggregation of software metrics values. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–9, 2010.

[100] Chandrasekar Subramaniam, Ravi Sen, and Matthew L. Nelson. Determinants of open source software project success: A longitudinal study. *Decision Support Systems*, 46:576–585, 2009.

[101] Barbara G. Tabachnick and Linda S. Fidell. *Using Multivariate Statistics (6th Edition)*. Prentice Hall, 2012.

[102] Marco Torchiano, Massimiliano Di Penta, Filippo Ricca, Andrea De Lucia, and Filippo Lanubile. Migration of information systems in the Italian industry: A state of the practice survey. *Information and Software Technology*, 53(1):71 – 86, 2011.

[103] Tom Tourwé and Tom Mens. Automated support for framework-based software evolution. In *19th IEEE International Conference on Software Maintenance*, pages 148–157, 2003.

[104] Jaroslav Tulach. API stability. `http://wiki.netbeans.org/API_Stability`, January 7 2012. Consulted on June 19, 2012.

[105] Andrea Valerio, Giancarlo Succi, and Massimo Fenaroli. Domain analysis and framework-based software development. *SIGAPP Appl. Comput. Rev.*, 5(2):4–15, September 1997.

[106] Bogdan Vasilescu, Alexander Serebrenik, and Mark G. J. van den Brand. You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 313–322, 2011.

[107] Wayne F. Velicer and Joseph L. Fava. Effects of variable and subject sampling on factor pattern recovery. *Psychological Methods*, 3(2):231–251, 1998.

[108] Dawid Weiss. Measuring success of open source projects using web search engines. In *The First International Conference on Open Source Systems*, pages 93–99, 2005.

[109] Michel Wermelinger, Yu Yijun, and Angela Lozano. Design principles in architectural evolution: A case study. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 396–405, 2008.

[110] Michel Wermelinger and Yijun Yu. Analyzing the evolution of eclipse plugins. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 133–136, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1370750.1370783`, `doi:10.1145/1370750.1370783`.

[111] Wikipedia. Plug-in (computing). `http://en.wikipedia.org/wiki/Plug-in_(computing)`, accessed 22-July-2010.

[112] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: A hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 325–334, 2010.

[113] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *Proc. 25th IEEE Int'l Conf. on Soft. Maint. (ICSM2009)*, pages 51–60, September 2007.

[114] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported – an Eclipse case study. In *ICSM'06*, pages 458–468, 2006.

[115] Zhenchang Xing and Eleni Stroulia. API-evolution support with Diff-catchup. *Journal of IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[116] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 2006.

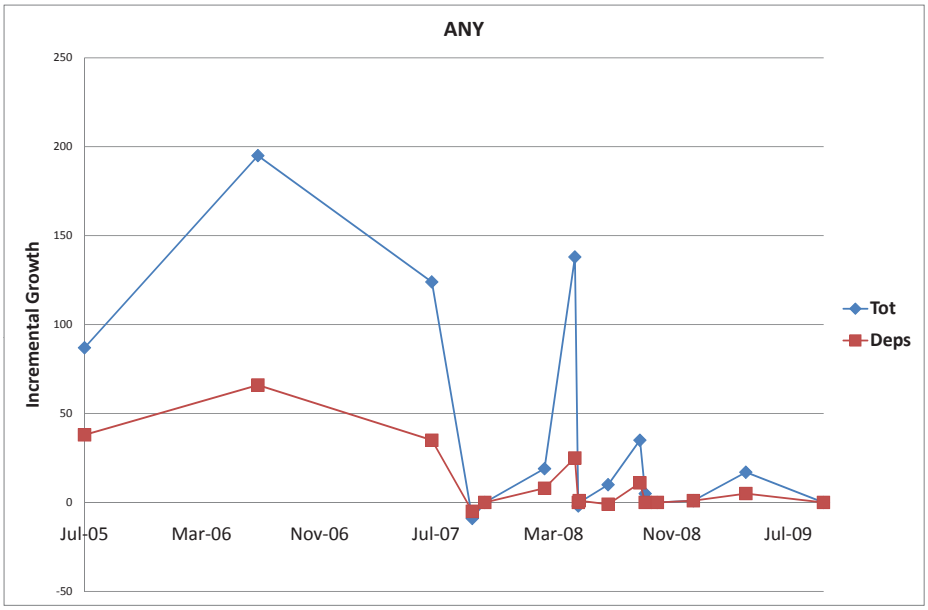
Appendix A

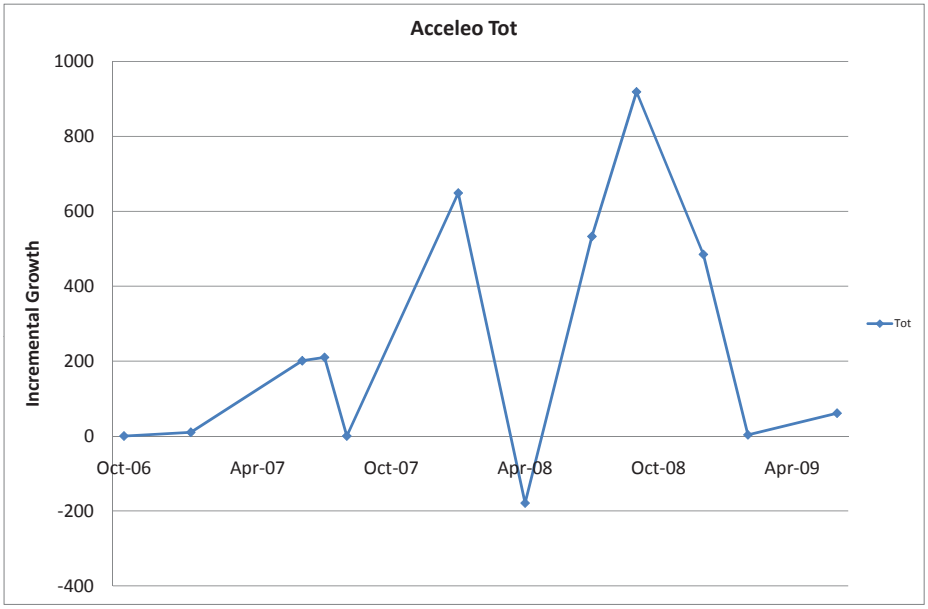
Evolution of ETPs

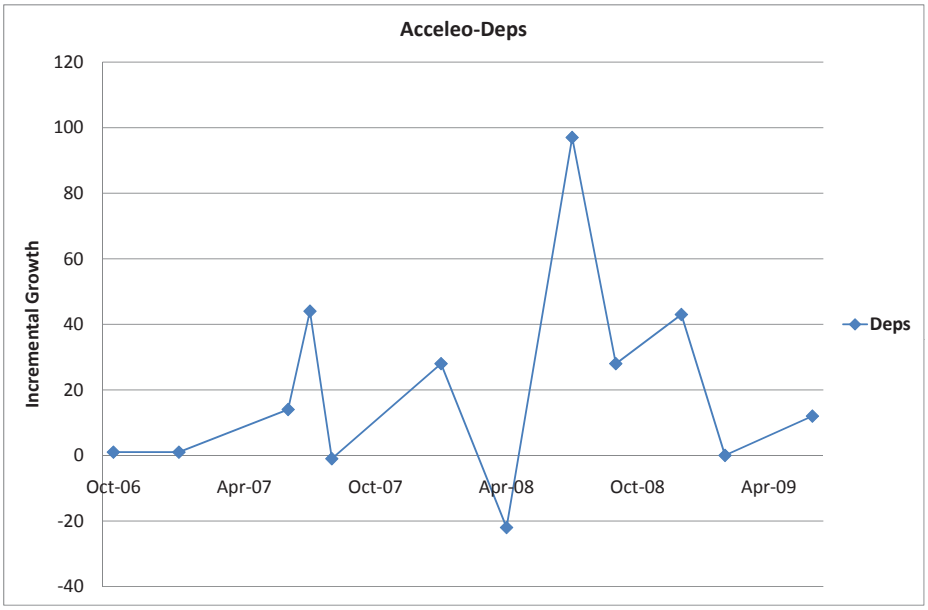
This Appendix presents the additional plots for Chapter 2 on the “chap:Evolution-of-ETPs”

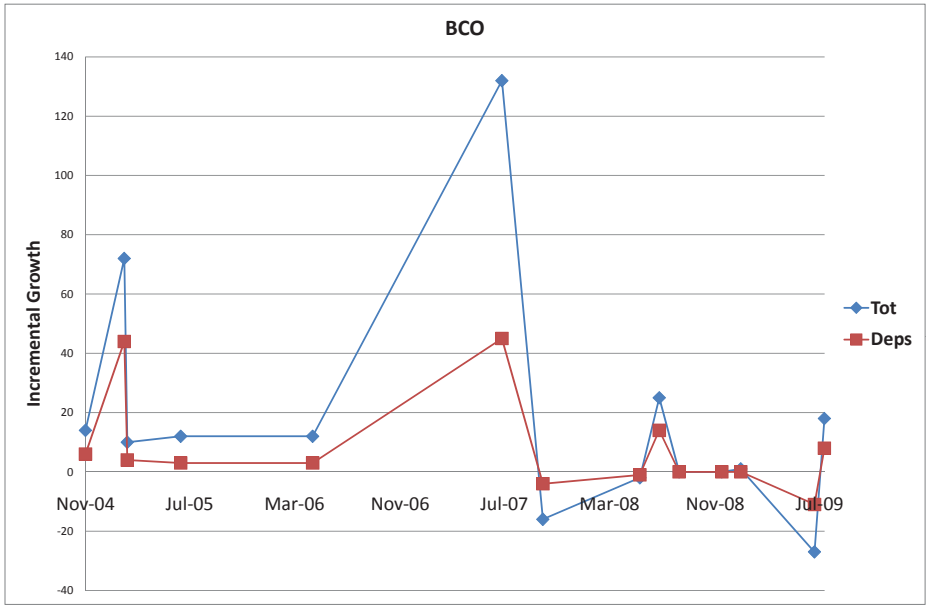
A.1 Self Regulation

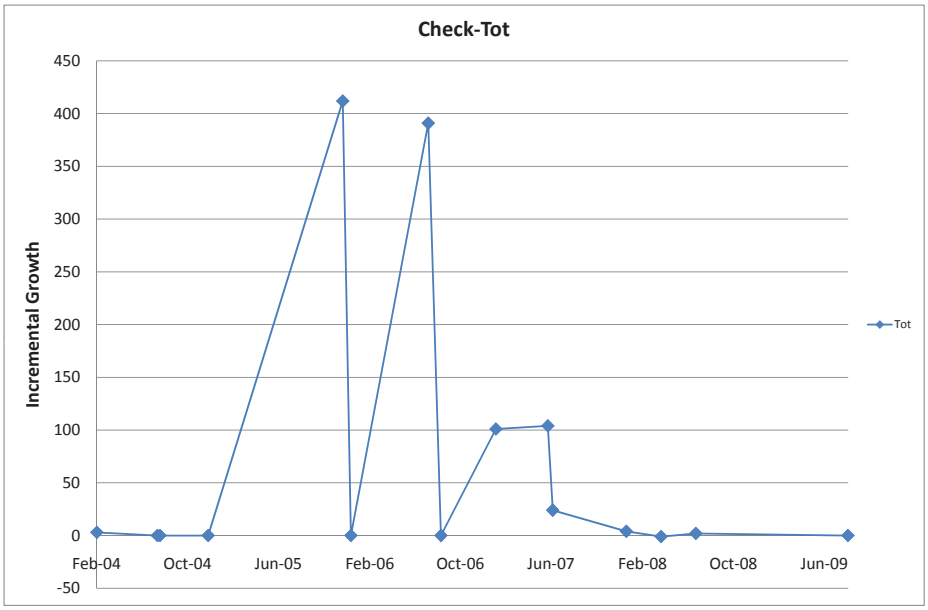
Extra graphs for the continuous growth law in Section 2.4.3 of Chapter 2

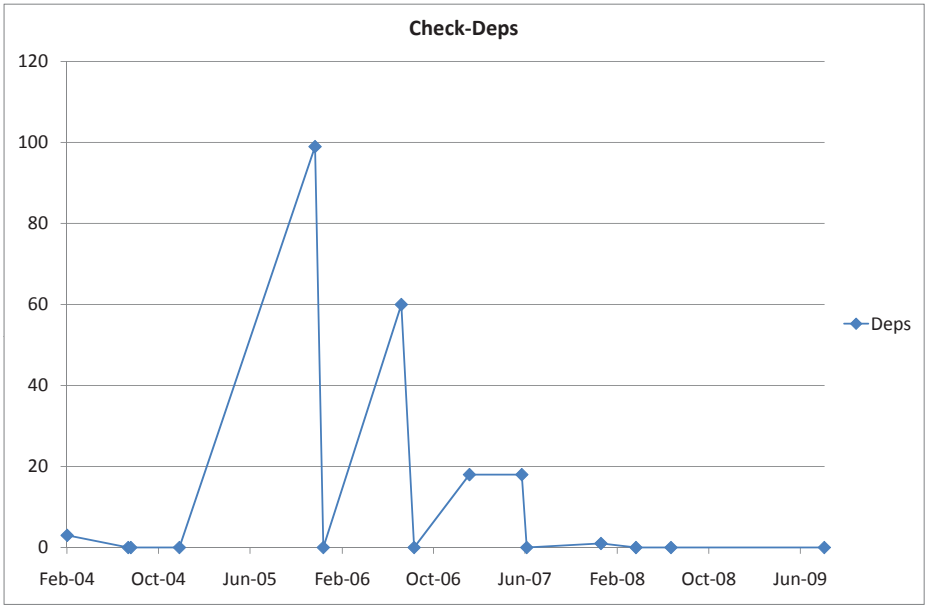

ANY
Incremental Growth
250
200
150
100
50
0
-50
Jul-05
Mar-06
Nov-06
Jul-07
Mar-08
Nov-08
Jul-09
Tot
Deps

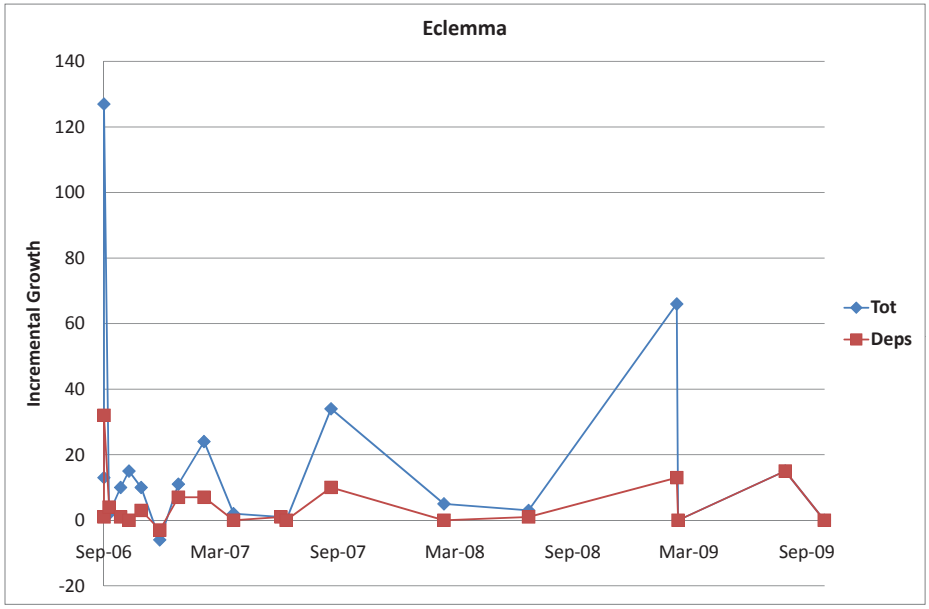

Acceleo Tot
Incremental Growth
1000
800
600
400
200
0
-200
-400
Oct-06
Apr-07
Oct-07
Apr-08
Oct-08
Apr-09
Tot

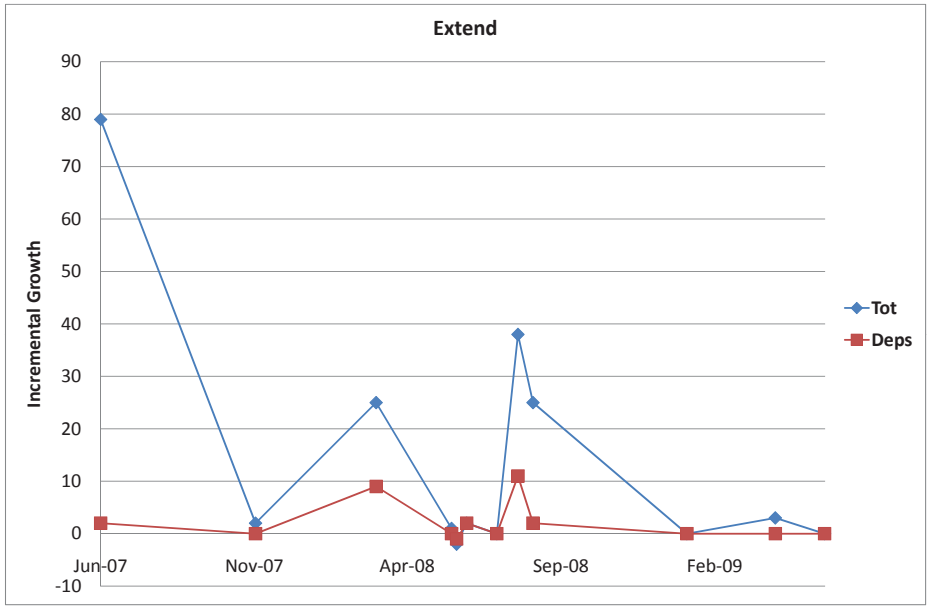

Acceleo-Deps
120
100
80
60
40
20
0
-20
-40
Incremental Growth
Oct-06
Apr-07
Oct-07
Apr-08
Oct-08
Apr-09
Deps

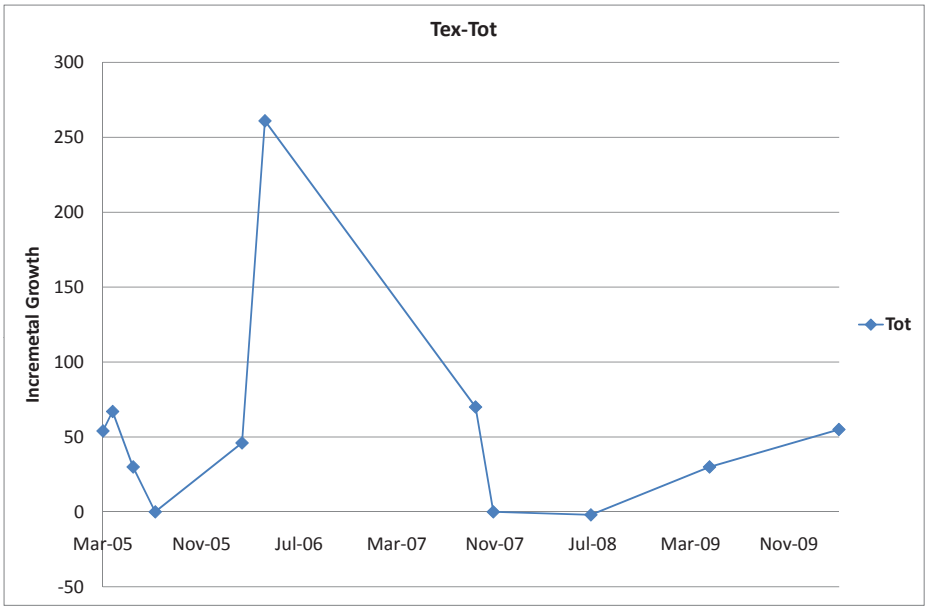

BCO
140
120
100
80
60
40
20
0
-20
-40
Incremental Growth
Nov-04
Jul-05
Mar-06
Nov-06
Jul-07
Mar-08
Nov-08
Jul-09
Tot
Deps

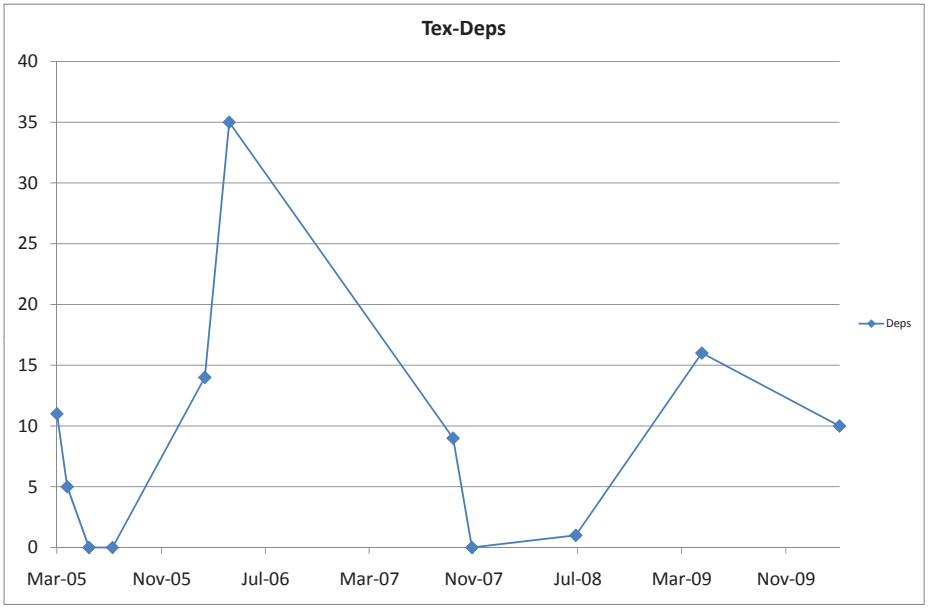

Check-Tot
Incremental Growth
450
400
350
300
250
200
150
100
50
0
-50
Feb-04
Oct-04
Jun-05
Feb-06
Oct-06
Jun-07
Feb-08
Oct-08
Jun-09
Tot

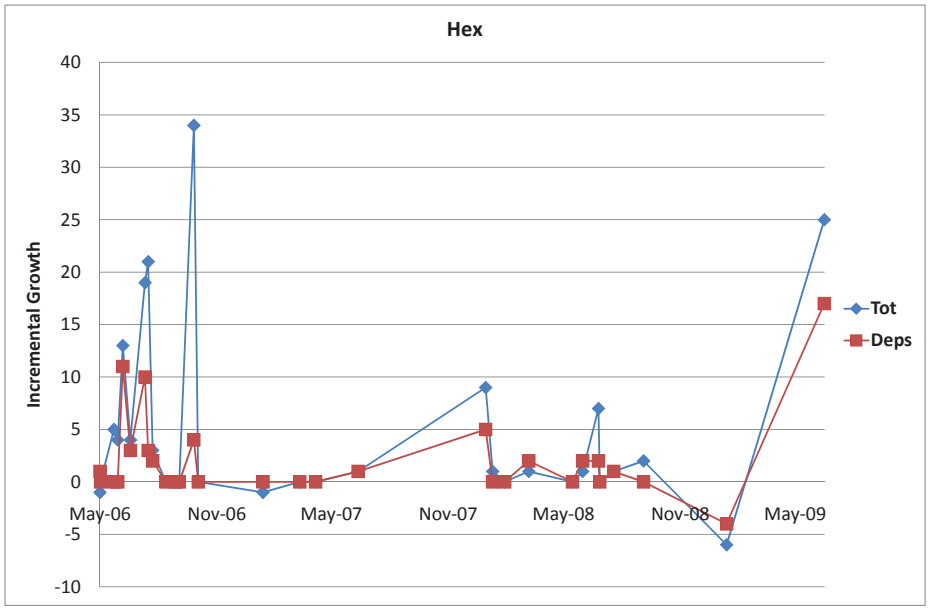

Check-Deps
120
100
80
60
40
20
0
Feb-04
Oct-04
Jun-05
Feb-06
Oct-06
Jun-07
Feb-08
Oct-08
Jun-09
Deps

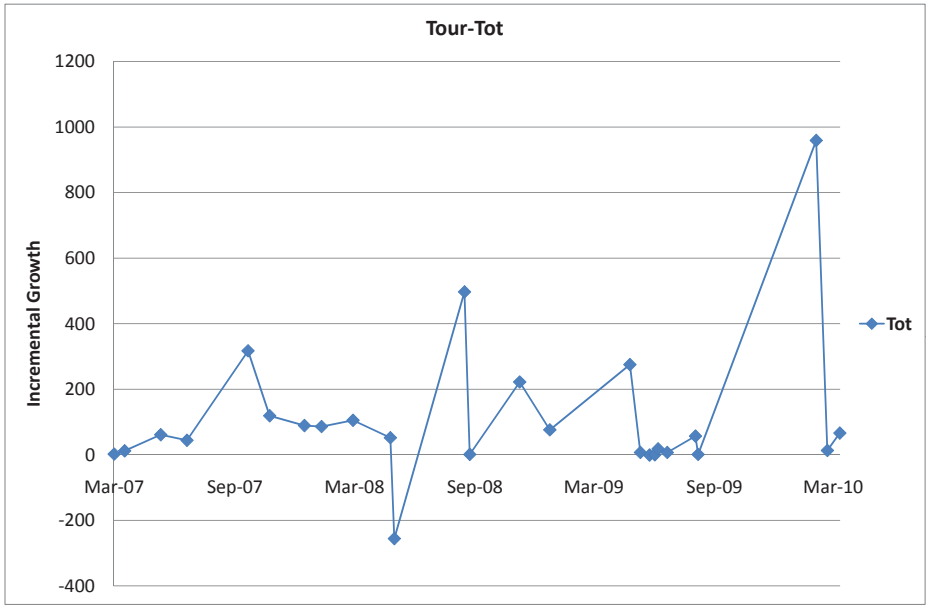

Eclemma
Incremental Growth
140
120
100
80
60
40
20
0
-20
Sep-06
Mar-07
Sep-07
Mar-08
Sep-08
Mar-09
Sep-09
Tot
Deps

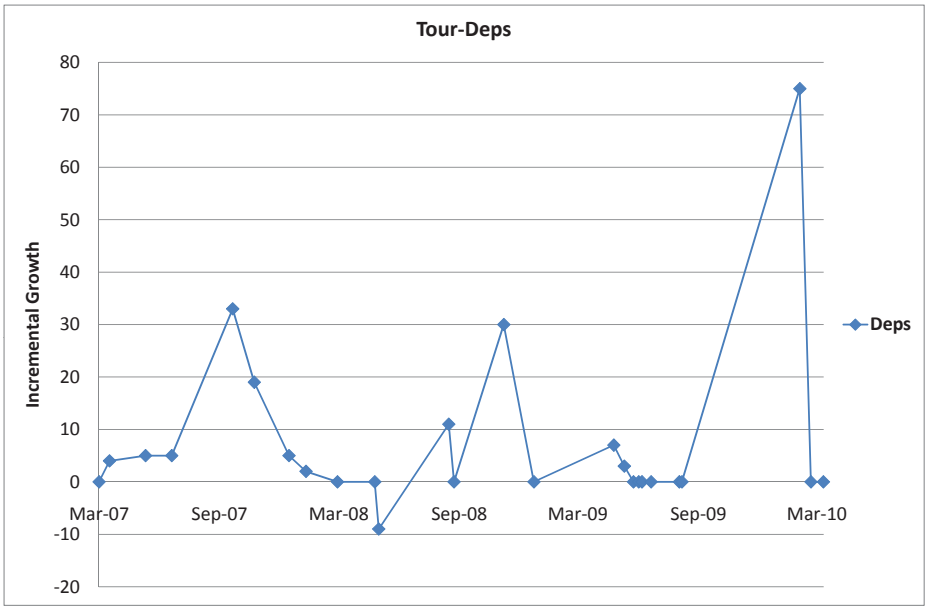

Extend
Incremental Growth
90
80
70
60
50
40
30
20
10
0
-10
Jun-07
Nov-07
Apr-08
Sep-08
Feb-09
Tot
Deps

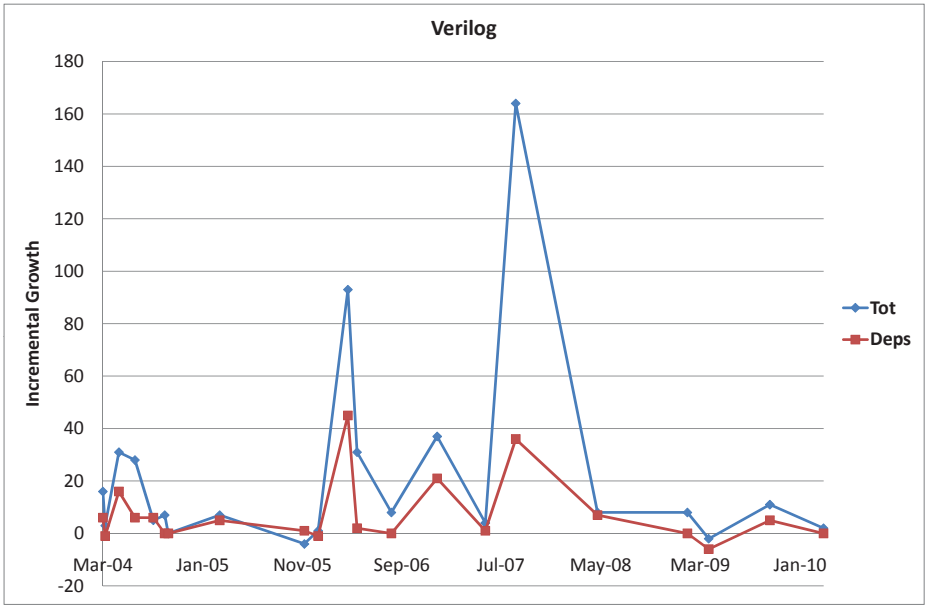

Tex-Tot
Incremetal Growth
300
250
200
150
100
50
0
-50
Mar-05
Nov-05
Jul-06
Mar-07
Nov-07
Jul-08
Mar-09
Nov-09
Tot

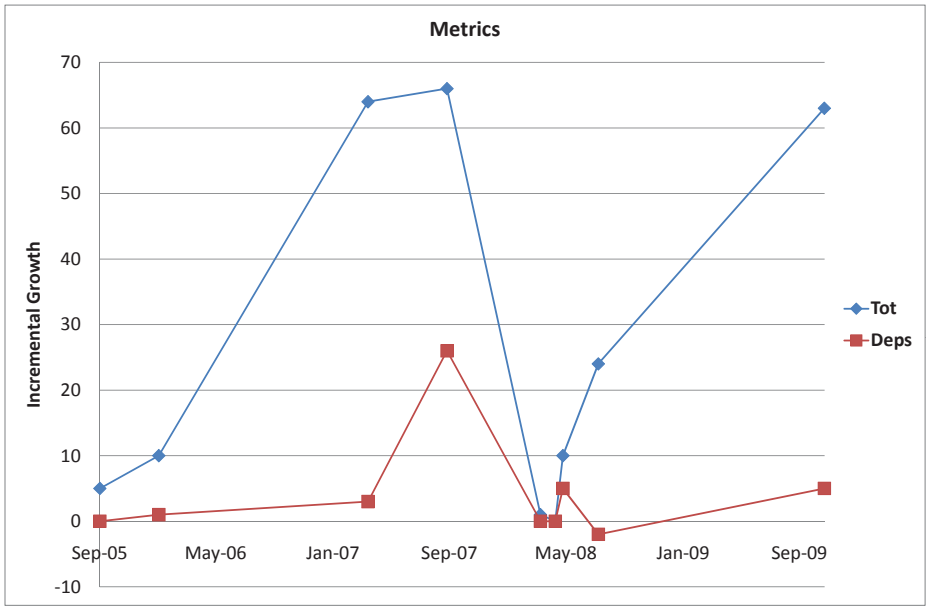

Tex-Deps
40
35
30
25
20
15
10
5
0
Mar-05
Nov-05
Jul-06
Mar-07
Nov-07
Jul-08
Mar-09
Nov-09
Deps

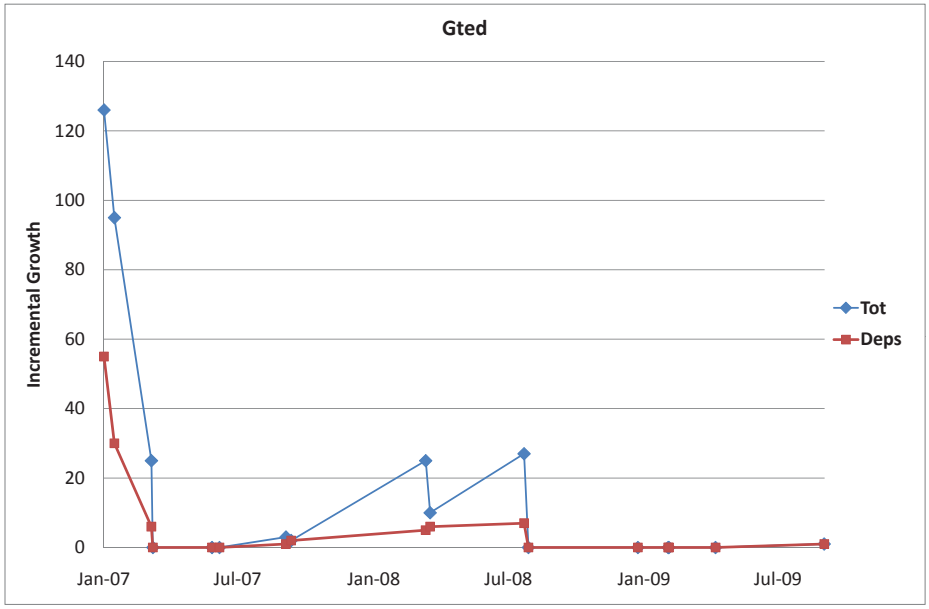

Hex
Incremental Growth
40
35
30
25
20
15
10
5
0
-5
-10
May-06
Nov-06
May-07
Nov-07
May-08
Nov-08
May-09
Tot
Deps

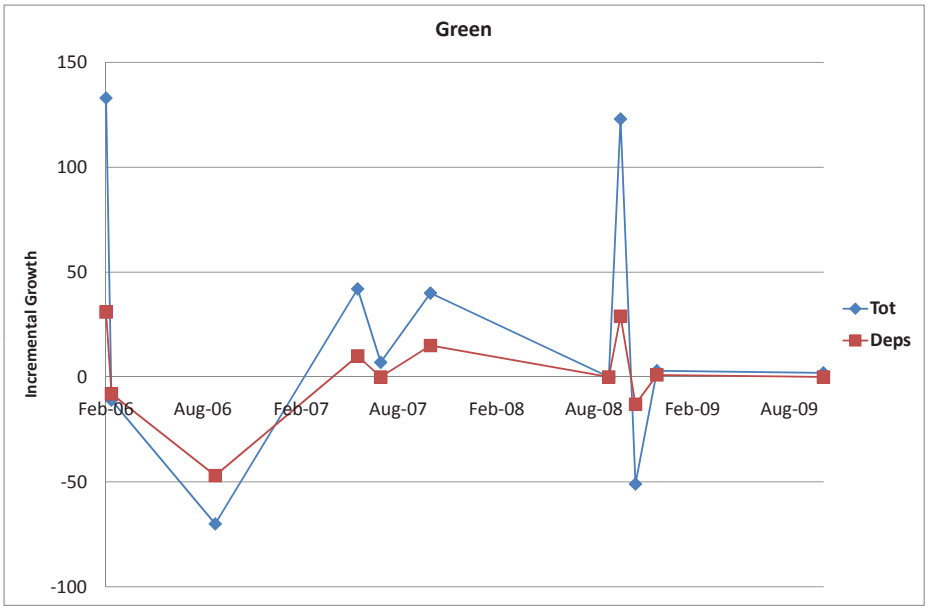

Tour-Tot
Incremental Growth
1200
1000
800
600
400
200
0
-200
-400
Mar-07
Sep-07
Mar-08
Sep-08
Mar-09
Sep-09
Mar-10
Tot

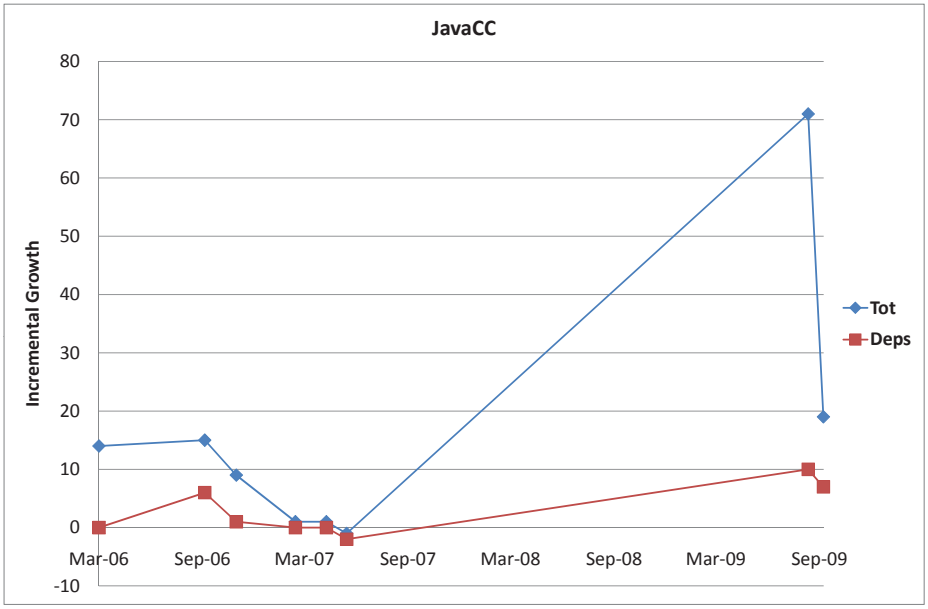

Tour-Deps
Incremental Growth
80
70
60
50
40
30
20
10
0
-10
-20
Mar-07
Sep-07
Mar-08
Sep-08
Mar-09
Sep-09
Mar-10
Deps


Verilog
Incremental Growth
180
160
140
120
100
80
60
40
20
0
-20
Mar-04
Jan-05
Nov-05
Sep-06
Jul-07
May-08
Mar-09
Jan-10
Tot
Deps


Metrics
Incremental Growth
70
60
50
40
30
20
10
0
-10
Sep-05
May-06
Jan-07
Sep-07
May-08
Jan-09
Sep-09
Tot
Deps

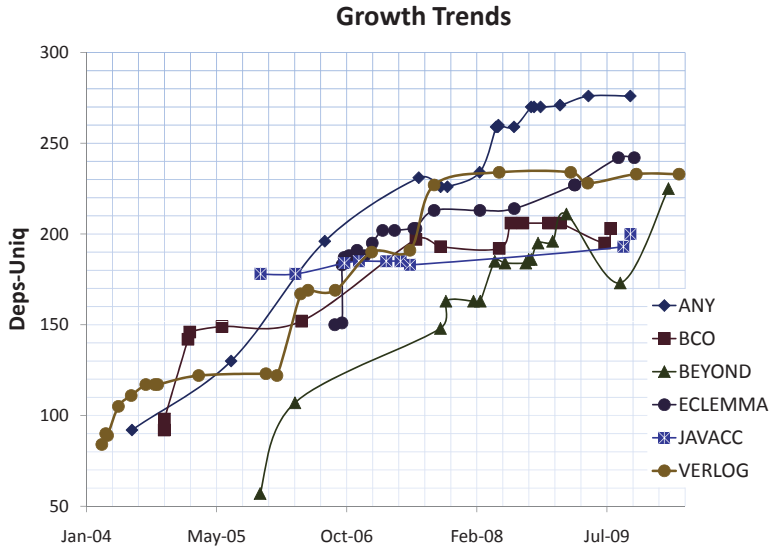

Gted
Incremental Growth
140
120
100
80
60
40
20
0
Jan-07
Jul-07
Jan-08
Jul-08
Jan-09
Jul-09
Tot
Deps

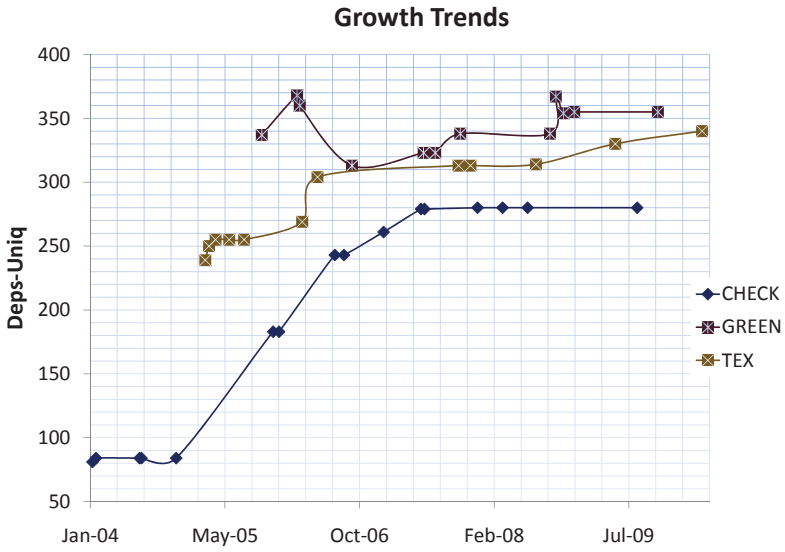

Green
150
100
50
0
-50
-100
Incremental Growth
Feb-06
Aug-06
Feb-07
Aug-07
Feb-08
Aug-08
Feb-09
Aug-09
Tot
Deps

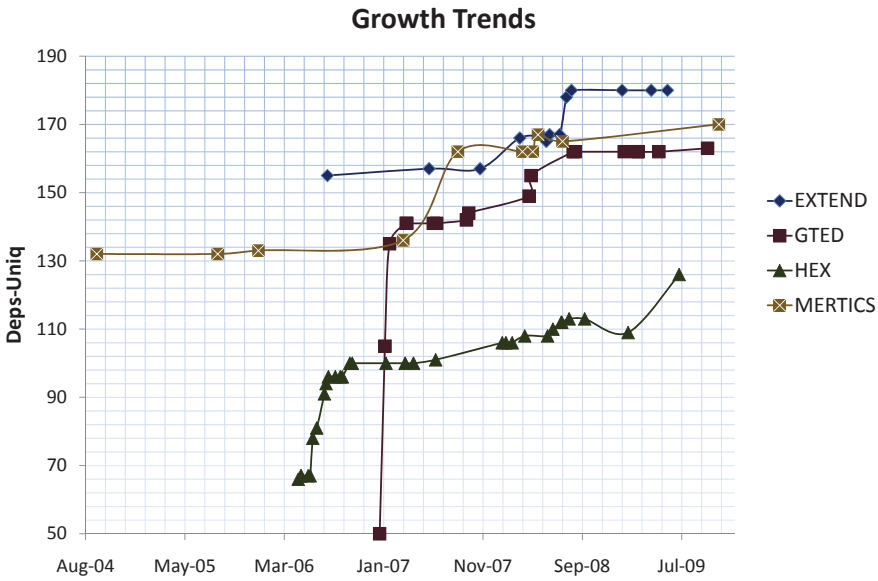

JavaCC
80
70
60
50
40
30
20
10
0
-10
Incremental Growth
Mar-06
Sep-06
Mar-07
Sep-07
Mar-08
Sep-08
Mar-09
Sep-09
Tot
Deps

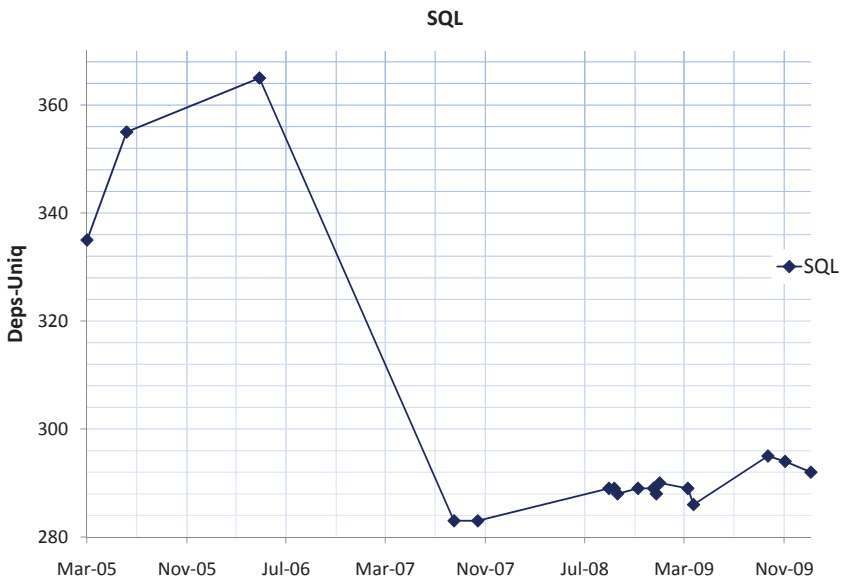
A.2 Continuing Growth

Extra graphs for the continuous growth law in Section 2.4.6 of Chapter 2






Growth Trends
Deps-Uniq
190
170
150
130
110
90
70
50
Aug-04
May-05
Mar-06
Jan-07
Nov-07
Sep-08
Jul-09
EXTEND
GTED
HEX
MERTICS


SQL
Deps-Uniq
360
340
320
300
280
Mar-05
Nov-05
Jul-06
Mar-07
Nov-07
Jul-08
Mar-09
Nov-09
SQL

Appendix B

Interface Usage

This Appendix presents figures accompanying the Chapter 4 on the “Eclipse API Usage: The Good and The Bad”

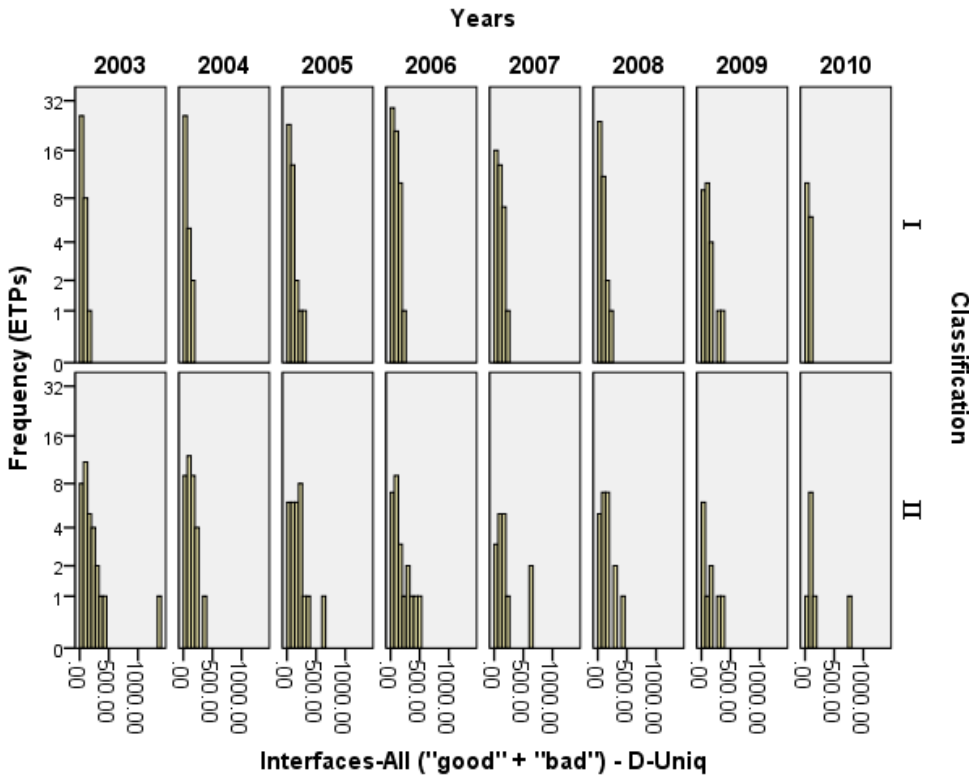


Figure 1: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set I**

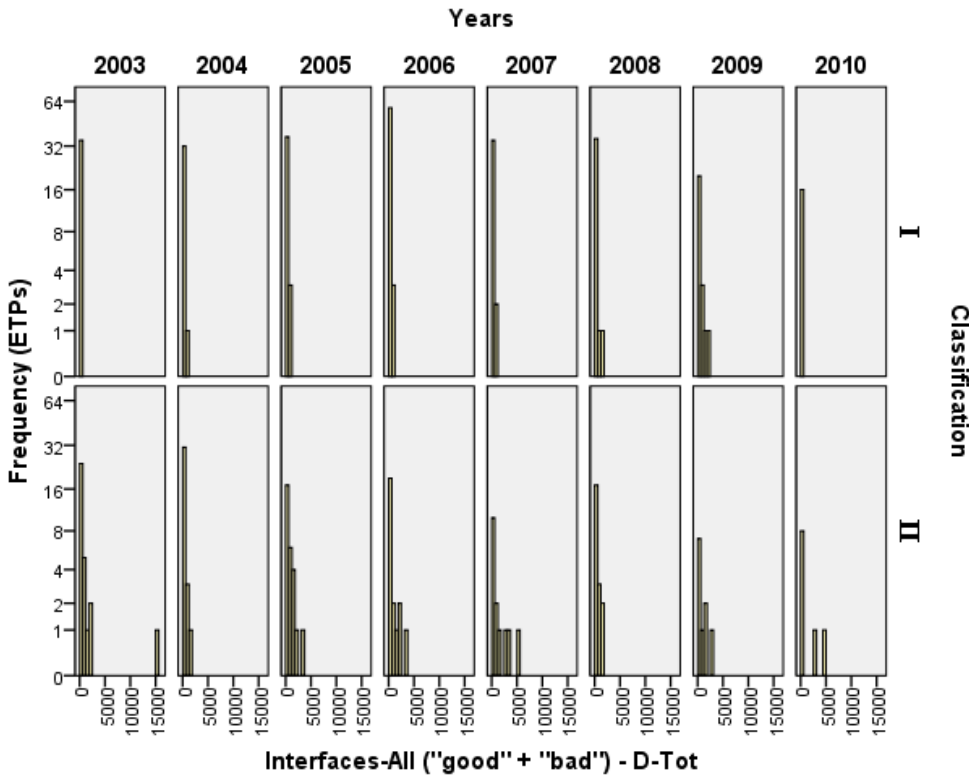


Figure 2: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set I**

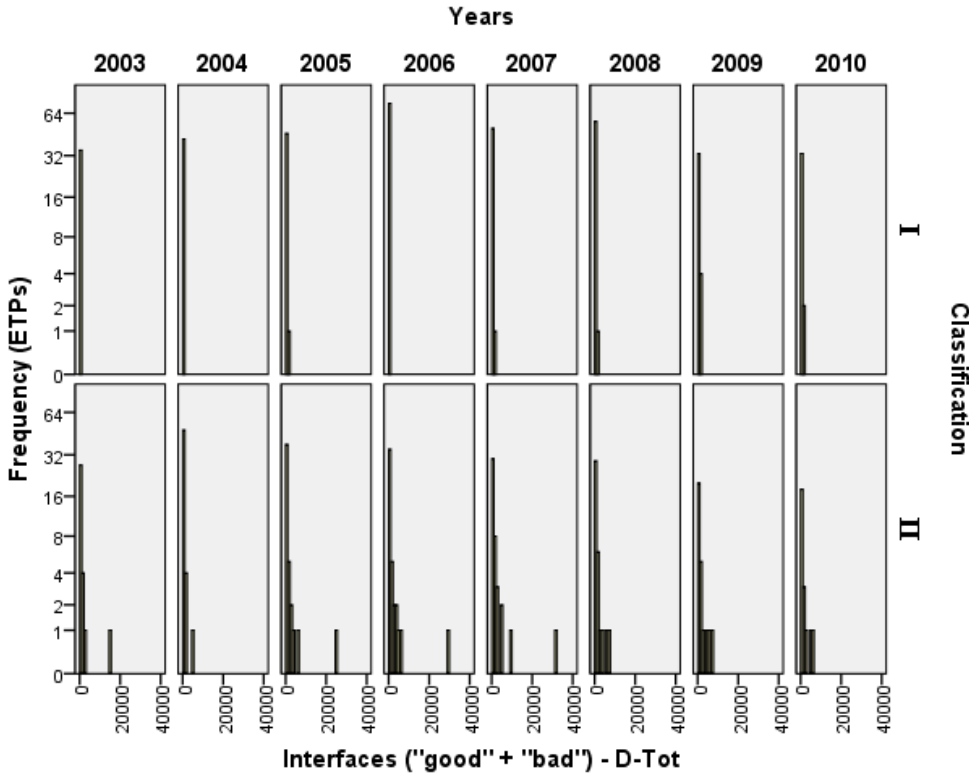


Figure 3: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set II**

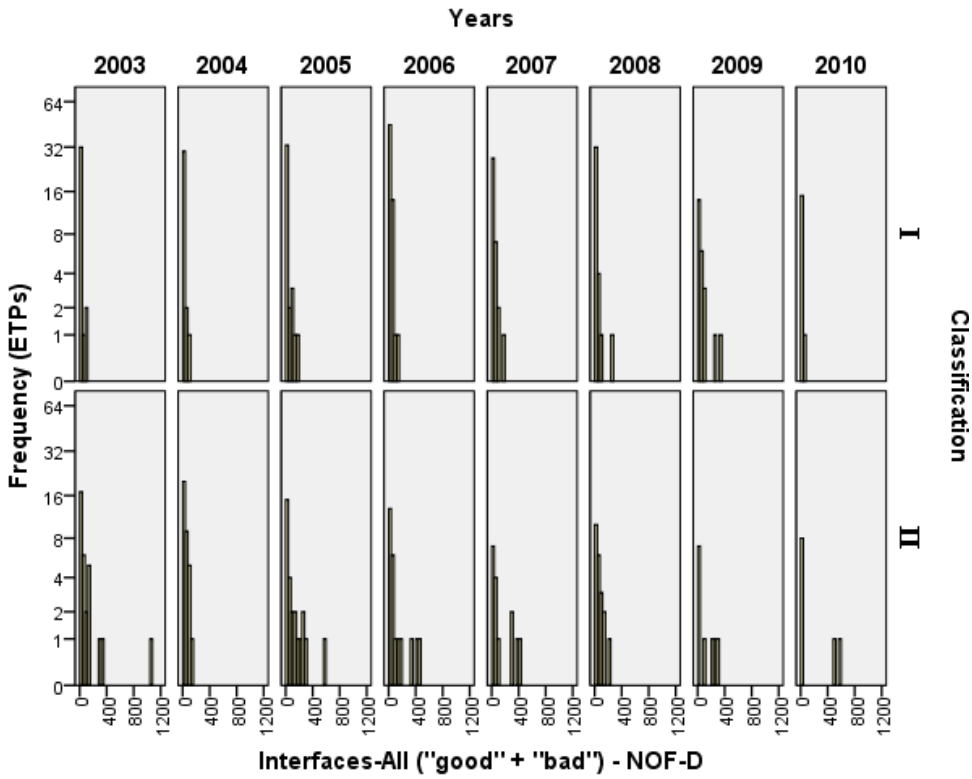


Figure 4: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set I**

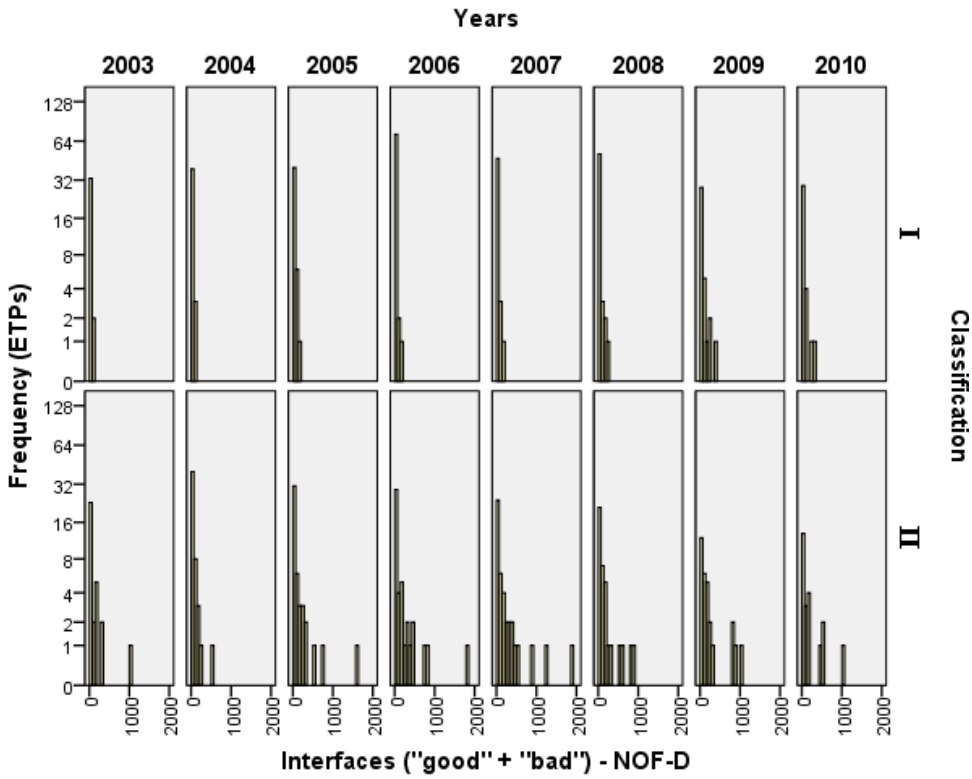


Figure 5: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set II**

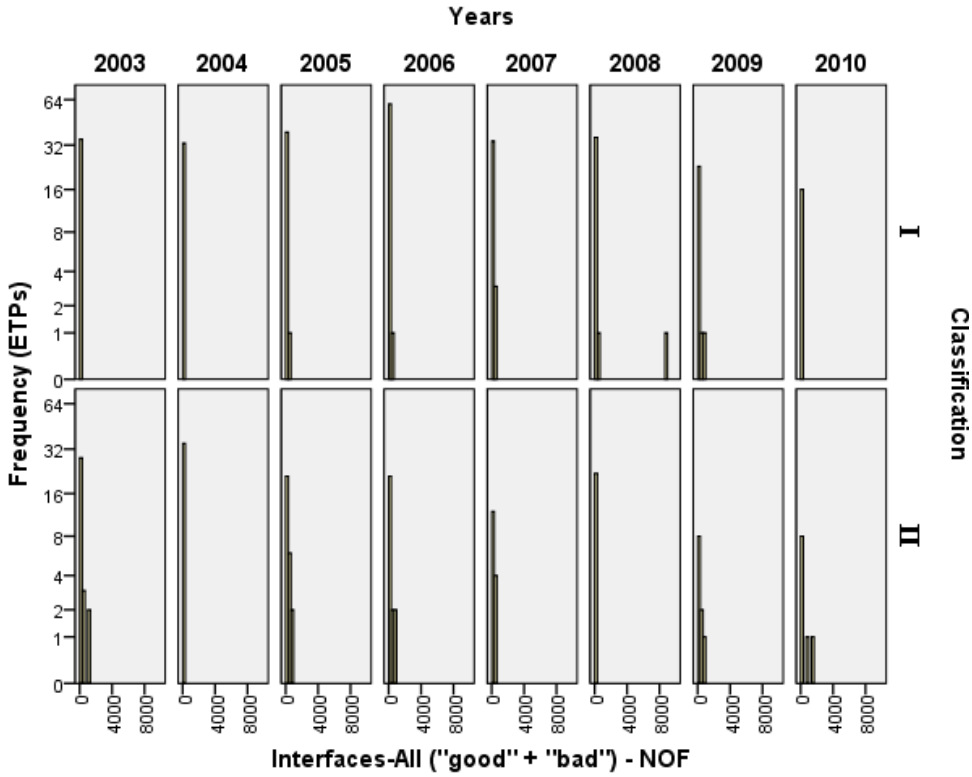


Figure 6: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set I**

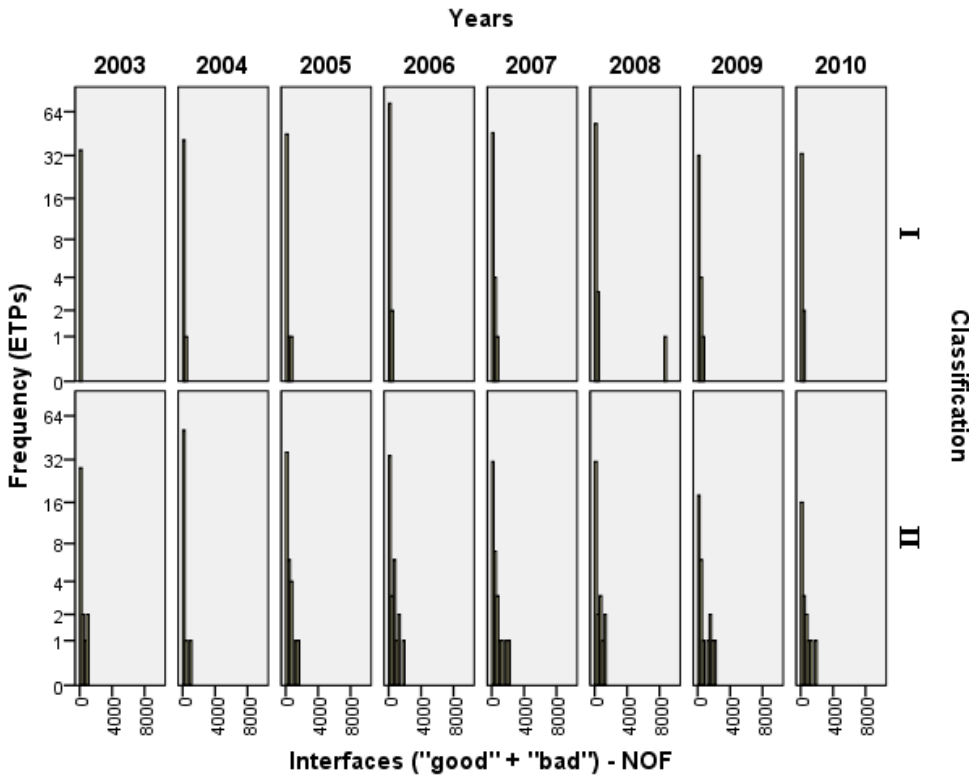


Figure 7: Histograms showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. The y-axis histogram shows the count of ETPs falling in a given range of ECP-Interface-count (x-axis). The the y-axis is a logarithmic scale of *base 2*. **Data-set II**

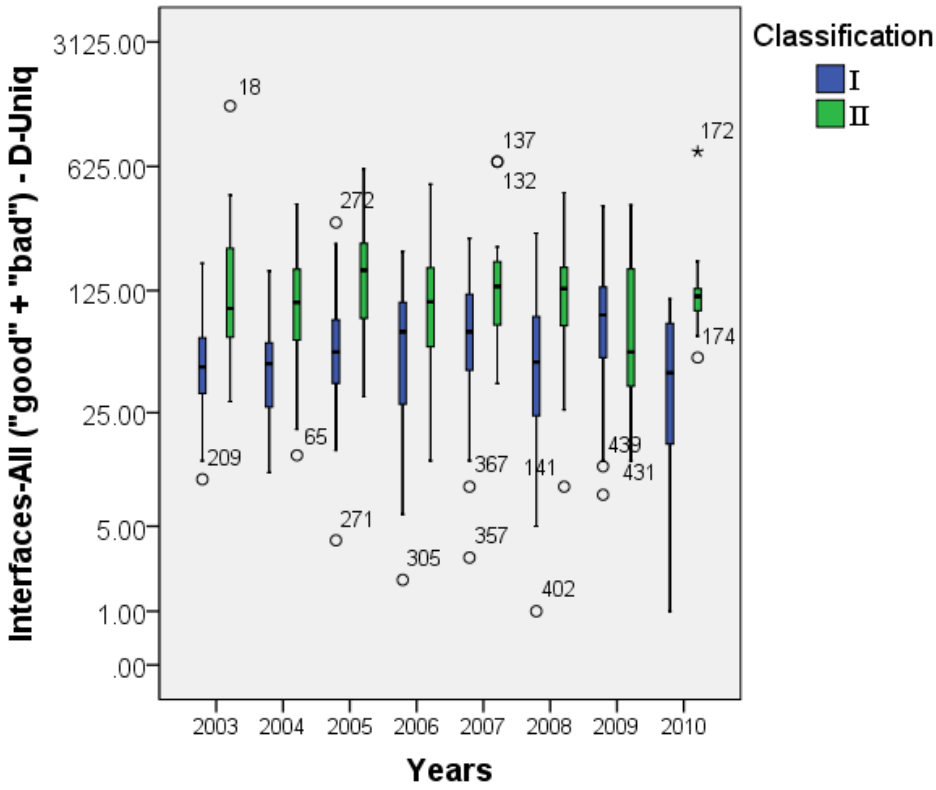


Figure 8: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set I**

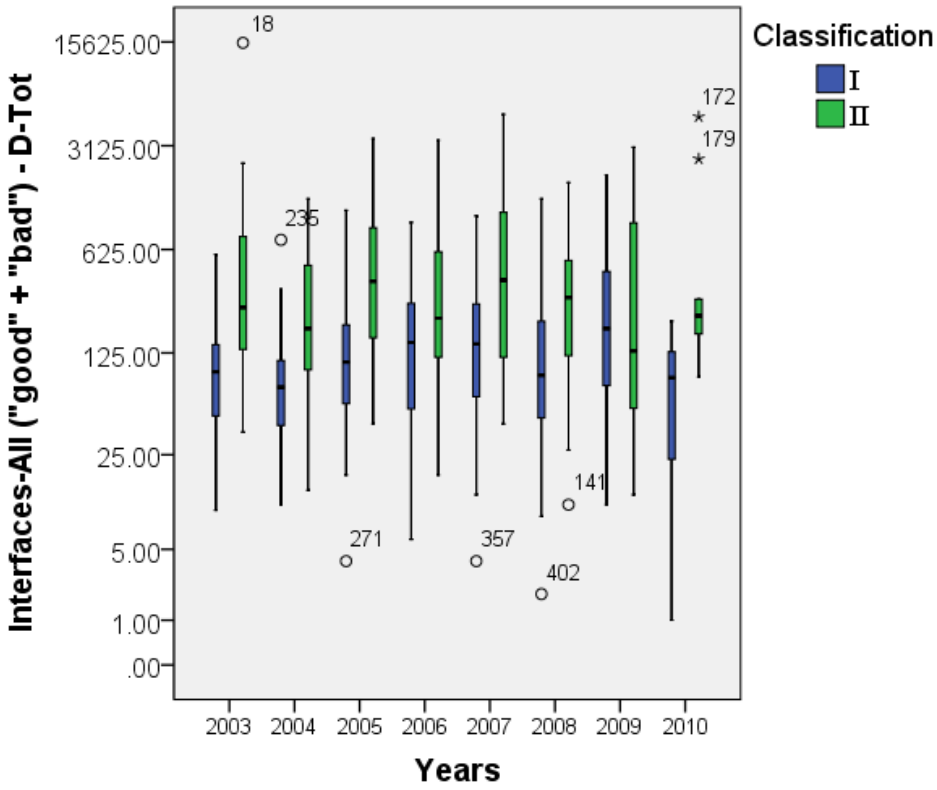


Figure 9: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set I**

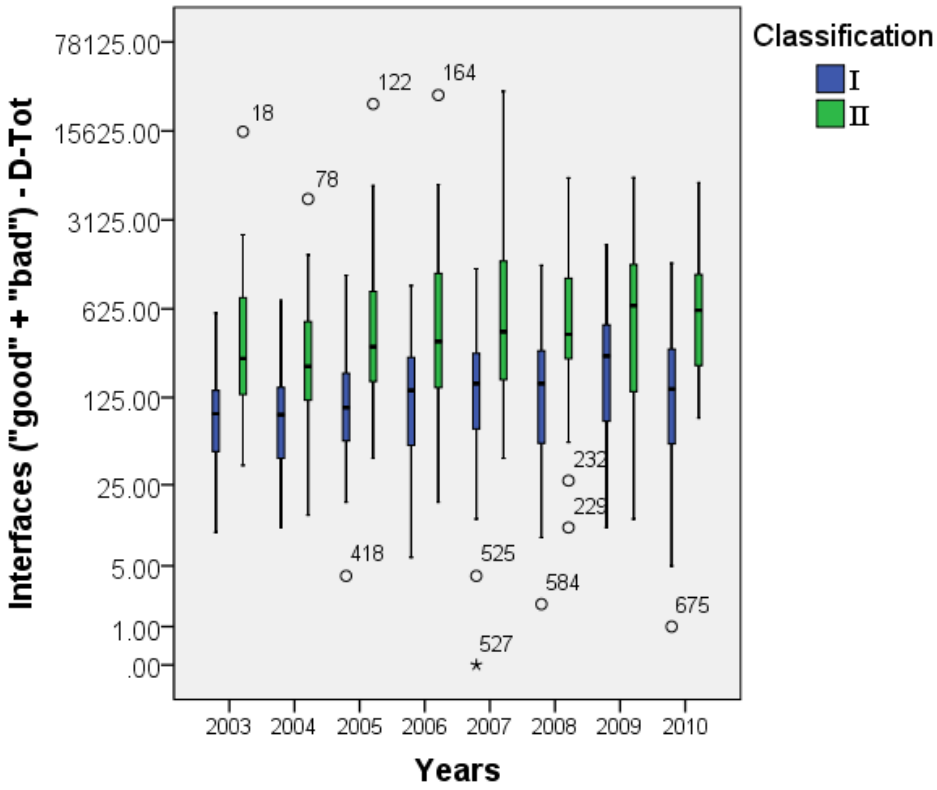


Figure 10: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set II**

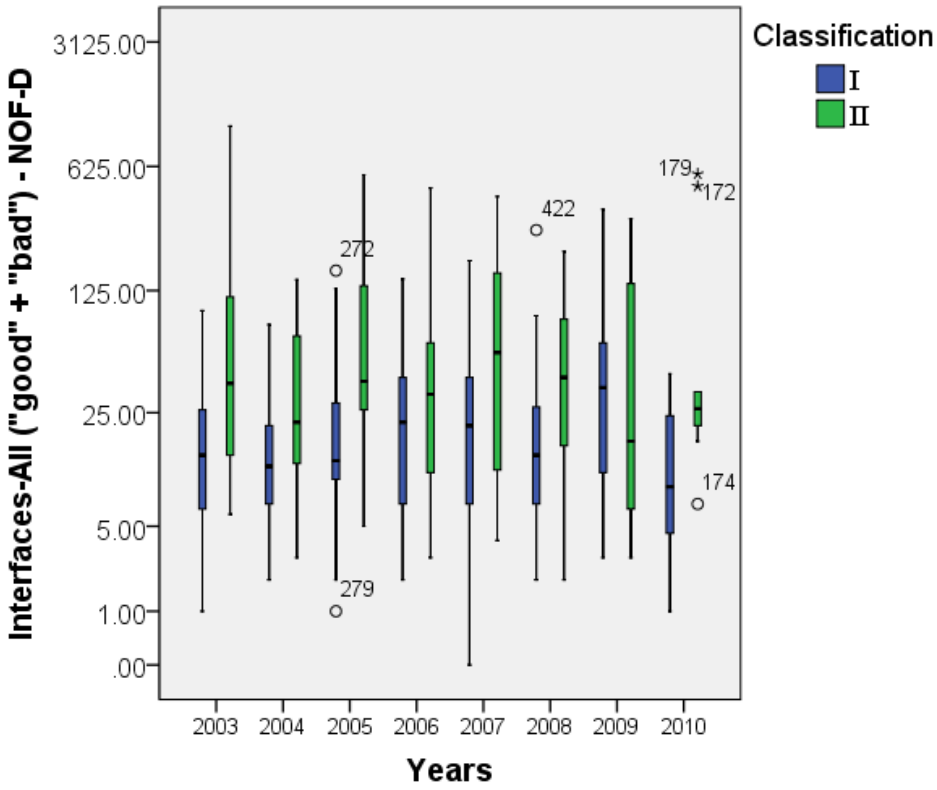


Figure 11: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set I**

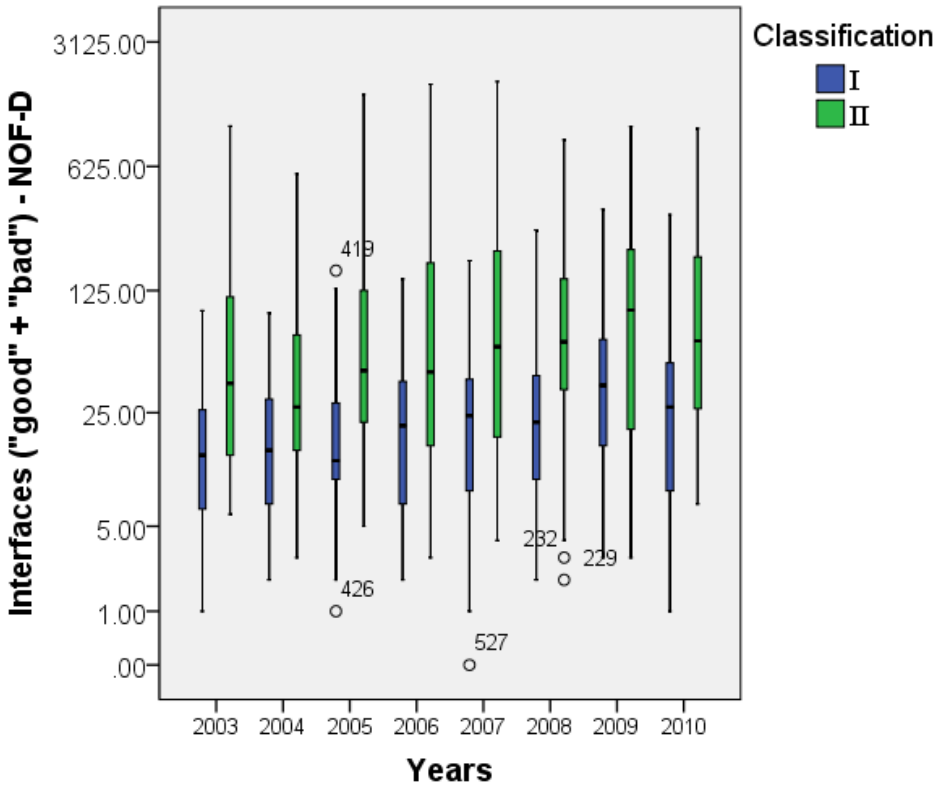


Figure 12: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set II**

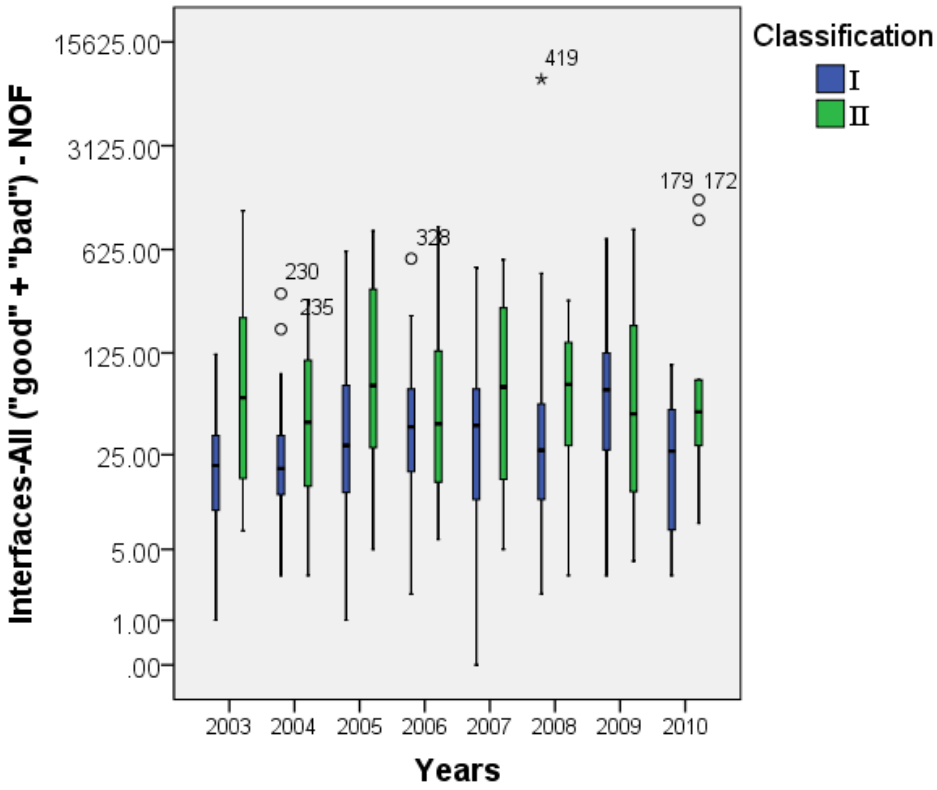


Figure 13: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set I**

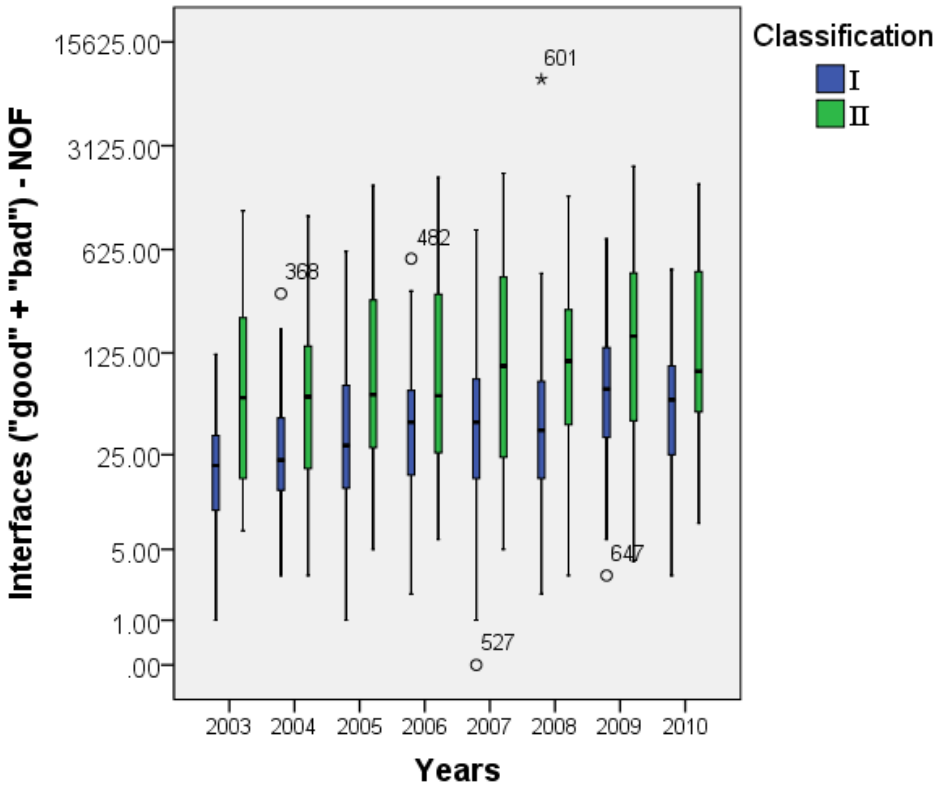


Figure 14: Box plots showing the distributions of the number of ECP Interfaces ("good" + "bad") depended on by ETPs. Each data point in a box-plot is an ETP and the height of the data point is the count of ECP Interfaces used by the ETP. The the y-axis is a logarithmic scale of *base 5*. **Data-set II**

Commonly used non-APIs by ETP-non-APIs

non-API	# ETPs
org.eclipse.jdt.internal.ui.JavaPlugin	28
org.eclipse.jdt.internal.core.JavaProject	15
org.eclipse.ui.internal.ide.IDEWorkbenchPlugin	14
org.eclipse.jdt.internal.corext.util.JavaModelUtil	12
org.eclipse.jdt.internal.ui.JavaPluginImages	11
org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor	9
org.eclipse.jdt.internal.core.PackageFragment	9
org.eclipse.jdt.internal.ui.wizards.TypedElementSelectionValidator	8
org.eclipse.core.internal.resources.Workspace	8
org.eclipse.jdt.internal.ui.util.ExceptionHandler	8
org.eclipse.jdt.internal.core.SourceType	8
org.eclipse.jdt.internal.ui.wizards.TypedViewerFilter	7
org.eclipse.core.internal.resources.File	7
org.eclipse.jdt.internal.ui.wizards.NewWizardMessages	7
org.eclipse.jdt.internal.ui.packageview.PackageExplorerPart	7
org.eclipse.ui.internal.ide.IDEWorkbenchMessages	7
org.eclipse.jdt.internal.core.JavaModelManager	6
org.eclipse.jdt.internal.ui.javaeditor.JavaEditor	6
org.eclipse.core.internal.resources.Project	6
org.eclipse.debug.internal.ui.DebugUIPlugin	6
org.eclipse.jdt.internal.ui.wizards.dialogfields.DialogField	6
org.eclipse.jdt.internal.ui.wizards.dialogfields.IDialogFieldListener	6
org.eclipse.jdt.internal.ui.wizards.dialogfields.IStringButtonAdapter	6
org.eclipse.jdt.internal.ui.javaeditor.EditorUtility	6
org.eclipse.core.internal.resources.ResourceException	6
org.eclipse.team.internal.ccvs.core.CVSProviderPlugin	5
org.eclipse.team.internal.ccvs.core.resources.CVSWorkspaceRoot	5
org.eclipse.team.internal.ccvs.core.CVSException	5
org.eclipse.jdt.internal.corext.codemanipulation.StubUtility	5
org.eclipse.jdt.internal.ui.IJavaHelpContextIds	5
org.eclipse.jdt.internal.core.CompilationUnit	5
org.eclipse.ui.internal.util.BundleUtility	5
org.eclipse.jdt.internal.ui.javaeditor.IClassFileEditorInput	5
org.eclipse.jdt.internal.ui.wizards.NewElementWizard	5
org.eclipse.ui.internal.dialogs.DialogUtil	4
org.eclipse.jdt.internal.ui.text.AbstractJavaScanner	4
org.eclipse.team.internal.ccvs.core.ICVSRemoteResource	4
org.eclipse.jdt.internal.corext.dom.ASTNodeFactory	4
org.eclipse.jdt.internal.corext.dom.GenericVisitor	4
org.eclipse.jdt.internal.corext.refactoring.Checks	4
org.eclipse.jdt.internal.corext.refactoring.util.TextChangeManager	4
org.eclipse.jdt.internal.ui.actions.SelectionConverter	4
org.eclipse.jdt.internal.ui.actions.WorkbenchRunnableAdapter	4
org.eclipse.jdt.internal.corext.util.SuperTypeHierarchyCache	4
org.eclipse.jdt.internal.ui.wizards.dialogfields.LayoutUtil	4
org.eclipse.jdt.internal.debug.ui.JDIDebugUIPlugin	4
org.eclipse.jdt.internal.ui.util.SWTUtil	4
org.eclipse.jdt.internal.ui.dialogs.StatusUtil	4

org.eclipse.jdt.internal.corext.dom.Selection	4
org.eclipse.jdt.internal.corext.util.Messages	4
org.eclipse.jdt.internal.ui.dialogs.StatusInfo	4
org.eclipse.jdt.internal.core.JarPackageFragmentRoot	4
org.eclipse.jdt.internal.core.JavaElement	4
org.eclipse.ui.internal.registry.PerspectiveDescriptor	4
org.eclipse.jdt.internal.formatter.DefaultCodeFormatter	4
org.eclipse.core.internal.runtime.InternalPlatform	4
org.eclipse.ui.internal.progress.ProgressMonitorJobsDialog	4
org.eclipse.ui.internal.ide.IIDEHelpContextIds	4
org.eclipse.ui.internal.wizards.datatransfer.DataTransferMessages	4
org.eclipse.team.internal.ccvs.core.ILogEntry	3
org.eclipse.jdt.internal.corext.refactoring.util.RefactoringFileBuffers	3
org.eclipse.jdt.internal.corext.refactoring.code.flow.FlowContext	3
org.eclipse.jdt.internal.ui.util.StringMatcher	3
org.eclipse.jdt.internal.ui.jarpackager.JarPackagerMessages	3
org.eclipse.jdt.internal.ui.viewsupport.ImageDescriptorRegistry	3
org.eclipse.team.internal.ccvs.core.ICVSResource	3
org.eclipse.team.internal.ccvs.ui.actions.CVSAction	3
org.eclipse.jdt.internal.ui.javaeditor.JavaSourceViewer	3
org.eclipse.jdt.internal.ui.text.JavaWhitespaceDetector	3
org.eclipse.jdt.internal.ui.text.JavaWordDetector	3
org.eclipse.ui.internal.editors.text.EditorsPlugin	3
org.eclipse.ui.internal.ide.dialogs.ResourceTreeAndListGroup	3
org.eclipse.jdt.internal.compiler.env.IBinaryType	3
org.eclipse.jdt.internal.compiler.impl.CompilerOptions	3
org.eclipse.core.internal.runtime.FindSupport	3
org.eclipse.ui.internal.views.navigator.ResourceNavigatorMessages	3
org.eclipse.swt.internal.win32.OS	3
org.eclipse.core.internal.resources.Resource	3
org.eclipse.team.internal.ccvs.core.CVSTeamProvider	3
org.eclipse.jdt.internal.corext.codemanipulation.GetterSetterUtil	3
org.eclipse.jdt.internal.ui.dialogs.PackageSelectionDialog	3
org.eclipse.jdt.internal.corext.dom.NodeFinder	3
org.eclipse.jdt.internal.corext.refactoring.rename.JavaRenameProcessor	3
org.eclipse.jdt.internal.corext.refactoring.rename.MethodChecks	3
org.eclipse.jdt.internal.corext.util.JdtFlags	3
org.eclipse.jdt.internal.ui.text.java.JavaCompletionProposal	3
org.eclipse.jdt.internal.ui.wizards.IStatusChangeListener	3
org.eclipse.jdt.internal.ui.dialogs.OptionalMessageDialog	3
org.eclipse.jdt.internal.ui.util.CoreUtility	3
org.eclipse.jdt.internal.ui.wizards.dialogfields.SelectionButtonDialogField	3
org.eclipse.jdt.internal.ui.wizards.dialogfields.StringButtonDialogField	3
org.eclipse.jdt.internal.ui.refactoring.RefactoringMessages	3
org.eclipse.jdt.internal.ui.refactoring.actions.RefactoringStarter	3
org.eclipse.jdt.internal.corext.dom.ASTNodes	3
org.eclipse.jdt.internal.corext.dom.Bindings	3
org.eclipse.jdt.internal.corext.dom.SelectionAnalyzer	3
org.eclipse.jdt.internal.corext.refactoring.RefactoringCoreMessages	3
org.eclipse.jdt.internal.corext.refactoring.base.JavaStatusContext	3

org.eclipse.jdt.internal.ui.viewsupport.JavaElementImageProvider	3
org.eclipse.jdt.internal.ui.wizards.dialogfields.IListAdapter	3
org.eclipse.jdt.internal.ui.wizards.dialogfields.ListDialogField	3
org.eclipse.jdt.internal.corext.dom.TokenScanner	3
org.eclipse.jdt.internal.corext.template.java.CodeTemplateContext	3
org.eclipse.jdt.internal.corext.template.java.CodeTemplateContextType	3
org.eclipse.jdt.internal.corext.util.CodeFormatterUtil	3
org.eclipse.jdt.internal.corext.util.Strings	3
org.eclipse.jdt.internal.ui.dialogs.FilteredTypesSelectionDialog	3
org.eclipse.jdt.internal.ui.search.JavaSearchScopeFactory	3
org.eclipse.jface.internal.text.html.BrowserInformationControl	3
org.eclipse.jface.internal.text.html.HTMLPrinter	3
org.eclipse.help.internal.browser.BrowserManager	3
org.eclipse.ui.internal.dialogs.WorkbenchWizardElement	3
org.eclipse.jdt.internal.launching.LaunchingPlugin	3
org.eclipse.ui.internal.ide.DialogUtil	3
org.eclipse.search.internal.ui.SearchMessages	3
org.eclipse.search.internal.ui.SearchPlugin	3
org.eclipse.search.internal.ui.SearchPluginImages	3
org.eclipse.core.internal.resources.WorkspaceRoot	3
org.eclipse.ui.internal.texteditor.TextEditorPlugin	3
org.eclipse.jdt.internal.ui.javaeditor.JarEntryEditorInput	3
org.eclipse.pde.internal.core.PDECore	3
org.eclipse.ui.internal.ide.dialogs.IDEResourceInfoUtils	3
org.eclipse.pde.internal.core.ifeature.IFeatureModel	3
org.eclipse.jdt.internal.corext.dom.ModifierRewrite	2
org.eclipse.jdt.internal.ui.SharedImages	2
org.eclipse.ui.internal.dialogs.WorkbenchPreferenceDialog	2
org.eclipse.jdt.internal.core.builder.JavaBuilder	2
org.eclipse.jdt.internal.ui.viewsupport.AppearanceAwareLabelProvider	2
org.eclipse.jdt.internal.ui.viewsupport.ImageImageDescriptor	2
org.eclipse.team.internal.ccvs.core.CVSTag	2
org.eclipse.team.internal.ccvs.core.ICVSFile	2
org.eclipse.team.internal.ccvs.core.ICVSRemoteFile	2
org.eclipse.team.internal.ccvs.ui.IHelpContextIds	2
org.eclipse.team.internal.ui.Utils	2
org.eclipse.team.internal.ui.actions.TeamAction	2
org.eclipse.team.internal.ui.synchronize.SyncInfoModelElement	2
org.eclipse.jdt.internal.ui.preferences.OverlayPreferenceStore	2
org.eclipse.jdt.internal.ui.text.java.JavaAutoIndentStrategy	2
org.eclipse.jdt.internal.ui.text.java.JavaCodeScanner	2
org.eclipse.jdt.internal.ui.text.java.JavaCompletionProcessor	2
org.eclipse.jdt.internal.ui.text.java.JavaDoubleClickSelector	2
org.eclipse.jdt.internal.ui.text.javadoc.JavaDocAutoIndentStrategy	2
org.eclipse.jdt.internal.ui.text.javadoc.JavaDocCompletionProcessor	2
org.eclipse.team.internal.ui.TeamUIPlugin	2
org.eclipse.jdt.internal.corext.refactoring.changes.CopyResourceChange	2
org.eclipse.jdt.internal.corext.refactoring.reorg.INewNameQuery	2
org.eclipse.jdt.internal.ui.viewsupport.StorageLabelProvider	2
org.eclipse.jdt.internal.ui.wizards.dialogfields.StringButtonStatusDialogField	2

org.eclipse.team.internal.ccvs.core.ICVSFolder	2
org.eclipse.team.internal.ccvs.core.util.Util	2
org.eclipse.jdt.internal.ui.packageview.ClassPathContainer	2
org.eclipse.jdt.internal.compiler.ClassFile	2
org.eclipse.jdt.internal.compiler.DefaultErrorHandlingPolicies	2
org.eclipse.jdt.internal.compiler.IErrorHandlingPolicy	2
org.eclipse.jdt.internal.compiler.IProblemFactory	2
org.eclipse.jdt.internal.compiler.ast.CompilationUnitDeclaration	2
org.eclipse.jdt.internal.compiler.env.ICompilationUnit	2
org.eclipse.jdt.internal.compiler.env.INameEnvironment	2
org.eclipse.jdt.internal.compiler.lookup.ClassScope	2

Appendix C

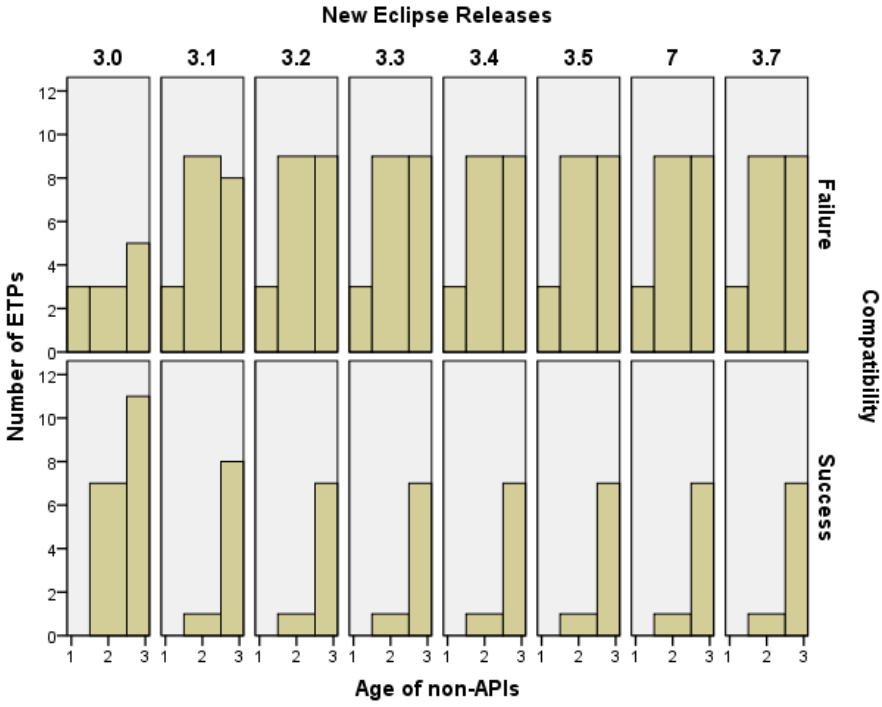
Compatibility Prediction

This Appendix presents tables and figures accompanying the Chapter 6 on the "Compatibility Prediction of Eclipse Plug-ins Over New Eclipse Releases"

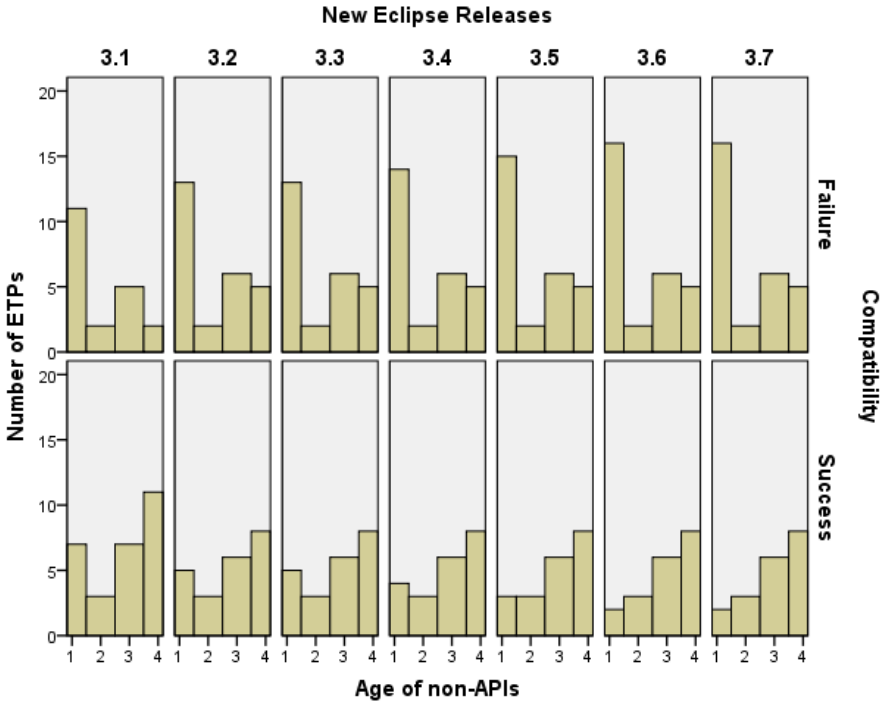
					non-APIs from Eclipse Releases									
		Compatibility	# ETPs	Aggregation	1.0	2.0	2.1	3.0	3.1	3.2	3.3	3.4	3.5	3.6
ETPs Supported in Eclipse Releases	2.1	Failure	21	Median	2	1	0							
				Mean	3.7	2.9	0.1							
				Max	14	20	1							
				Min	0	0	0							
				Sum	77	60	3							
		Success	8	Median	1	0	0							
				Mean	2	0	0							
				Max	3	2	0							
				Min	1	0	0							
				Sum	12	2	0							
	3.0	Failure	29	Median	2	2	0	1						
				Mean	3.9	4.6	1.1	2						
				Max	24	24	11	17						
				Min	0	0	0	0						
				Sum	113	134	31	58						
		Success	19	Median	2	0	0	0						
				Mean	1.8	0.8	0.2	0.1						
				Max	5	6	2	1						
				Min	0	0	0	0						
				Sum	34	16	4	2						
	3.1	Failure	20	Median	4	3	0	2	0					
				Mean	4	5	1	3	1					
				Max	12	20	2	7	7					
				Min	0	0	0	0	0					
				Sum	75	93	13	50	12					
		Success	14	Median	1.5	0	0	0	0					
				Mean	1.6	0.4	0.1	0.5	0					
				Max	4	3	1	4	0					
				Min	0	0	0	0	0					
				Sum	23	5	1	7	0					
	3.2	Failure	24	Median	2.5	1	0	2	0	0				
				Mean	5.8	4.5	0.5	2.3	1.4	0.5				
				Max	32	20	2	8	11	4				
				Min	0	0	0	0	0	0				
				Sum	139	107	13	55	34	11				
		Success	16	Median	2	0	0	0	0	0				
				Mean	2.2	0.0	0.1	0.3	0.1	0				
				Max	8	3	1	2	1	0				
				Min	1	0	0	0	0	0				
				Sum	35	10	2	4	2	0				
	3.3	Failure	18	Median	5	2	0	3	1.5	0	0			
				Mean	5.2	3.4	0.6	6.2	3.8	1.4	1.4			
				Max	12	12	3	45	20	10	14			
				Min	0	0	0	0	0	0	0			
				Sum	93	62	11	111	68	26	26			
		Success	20	Median	2	0	0	0	0	0	0			
				Mean	2.9	0.6	0.2	0.3	0.2	0.1	0			
				Max	9	3	1	2	1	2	0			
				Min	0	0	0	0	0	0	0			
				Sum	58	12	4	5	3	2	0			
	3.4	Failure	8	Median	6	6.5	0	3.5	2	1	0	0		
				Mean	5.6	5.5	0.5	3.9	1.6	0.8	0	0.4		
				Max	12	10	2	8	4	2	0	2		
				Min	0	0	0	0	0	0	0			
				Sum	45	44	4	31	13	6	0	3		
		Success	28	Median	1.5	0	0	0	0	0	0	0		
				Mean	2.3	1.8	0.3	0.9	0.5	0.3	0.3	0.1		
				Max	13	15	2	5	7	6	5	2		
				Min	0	0	0	0	0	0	0	0		
				Sum	65	50	8	24	14	8	9	4		
	3.5	Failure	10	Median	1.5	0	0	0	0	0	0	0	0	
				Mean	2.3	1.8	0.3	0.9	0.5	0.3	0.3	0.1	0	
				Max	16	17	4	12	17	6	5	3	0	
				Min	0	0	0	0	0	0	0	0	0	
				Sum	65	50	8	24	14	8	9	4	0	
		Success	23	Median	1	0	0	0	0	0	0	0	0	
				Mean	2	2.3	0.1	1.5	0.9	0.1	0.1	0.1	0	
				Max	10	15	1	8	6	1	2	2	0	
				Min	0	0	0	0	0	0	0	0	0	
				Sum	45	53	3	34	20	3	2	2	0	
	3.6	Failure	2	Median	17	3	2	7	9	2	0	4	1	0
				Mean	17	3	2	7	9	2	0	4	1	0
				Max	20	5	4	13	17	3	0	7	1	0
				Min	14	1	0	1	1	1	0	0	0	0
				Sum	34	6	4	14	18	4	0	8	2	0
		Success	28	Median	1	1	0	0	0	0	0	0	0	0
				Mean	2	3	1	1	1	1	0	0	0	0
				Max	10	19	2	12	10	6	5	2	0	0
				Min	0	0	0	0	0	0	0	0	0	0
				Sum	64	77	14	38	27	16	6	4	0	0

Table I: The descriptive statistics of non-APIs in ETPs supported in the different Eclipse releases.

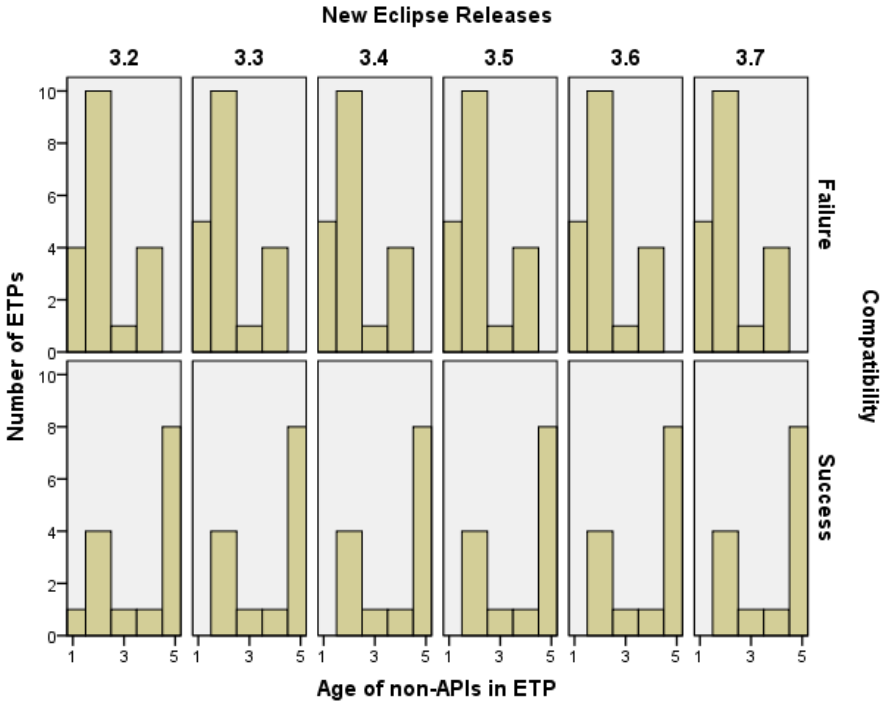
ETP Supported in Eclipse SDK 2.1



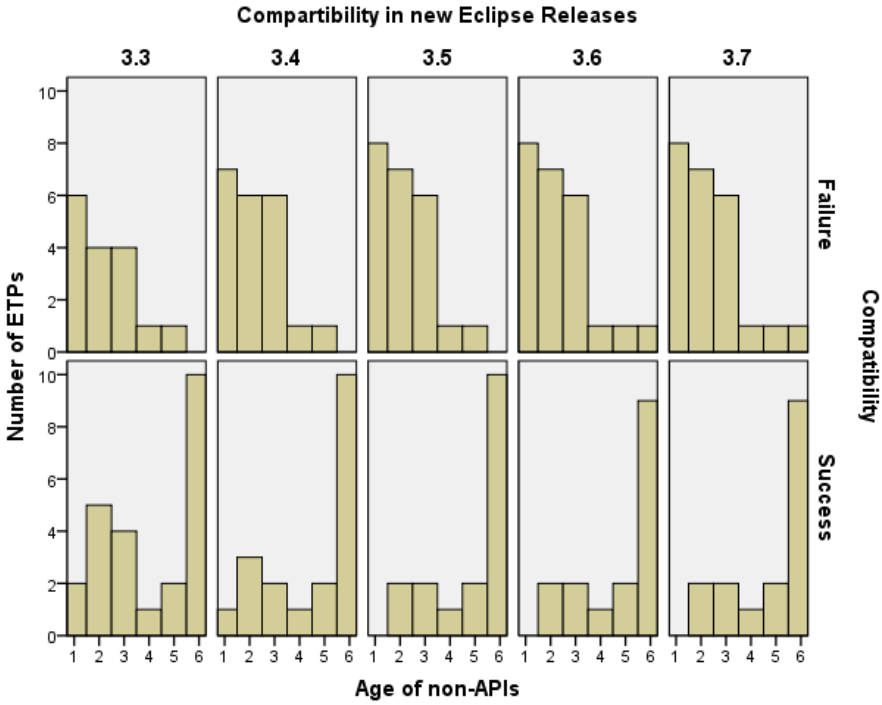
ETP Supported in Eclipse SDK 3.0



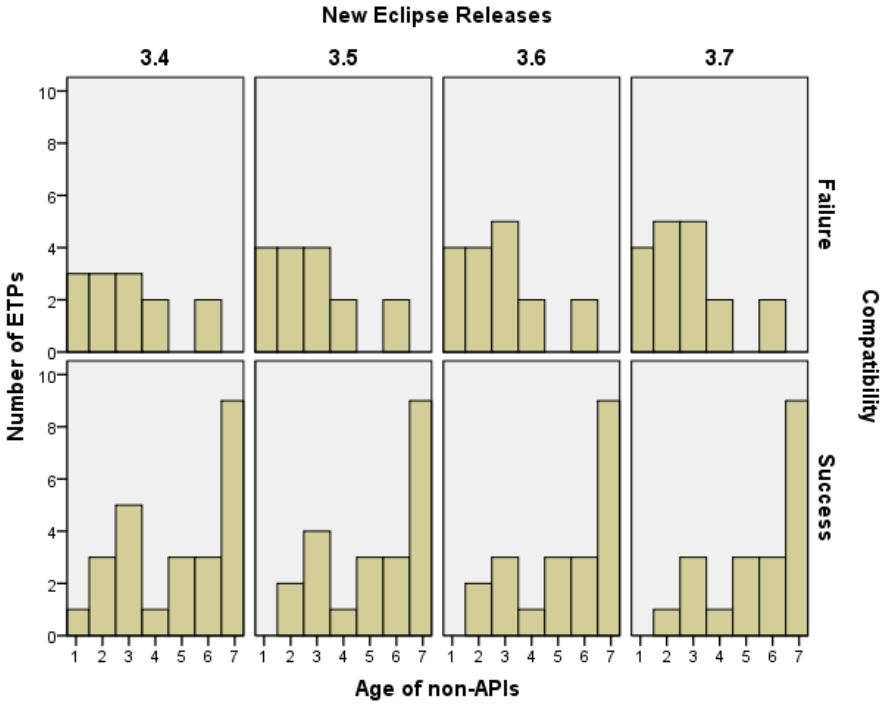
ETP Supported in Eclipse SDK 3.1



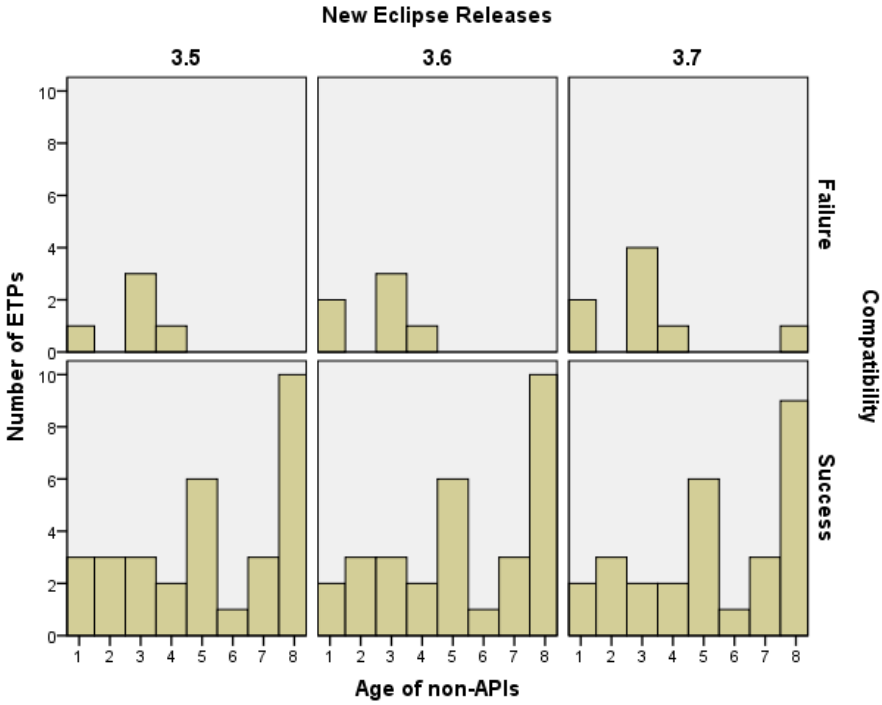
ETP Supported in Eclipse SDK 3.2



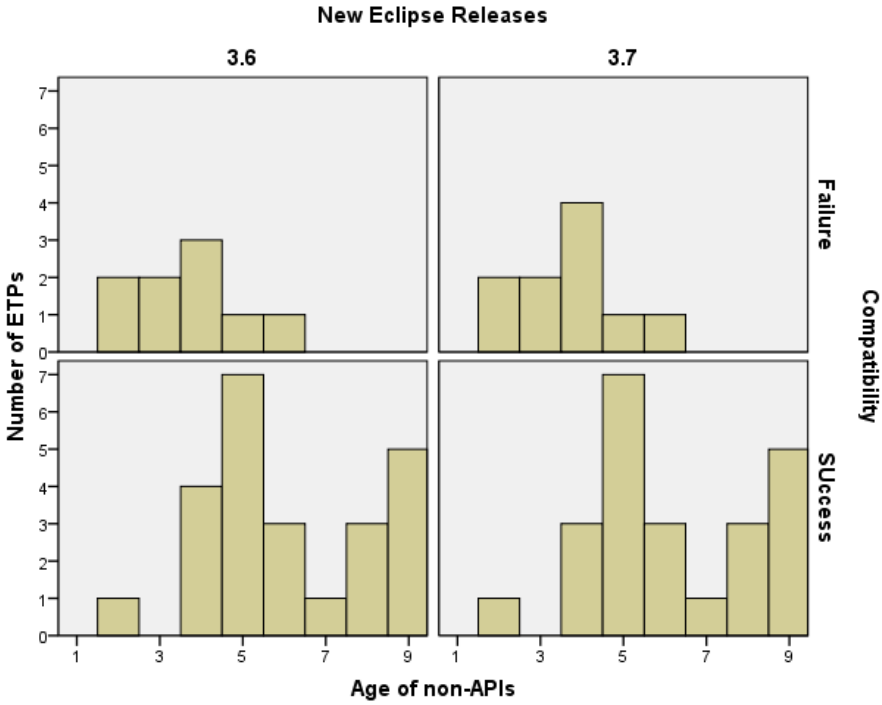
ETP Supported in Eclipse SDK 3.3



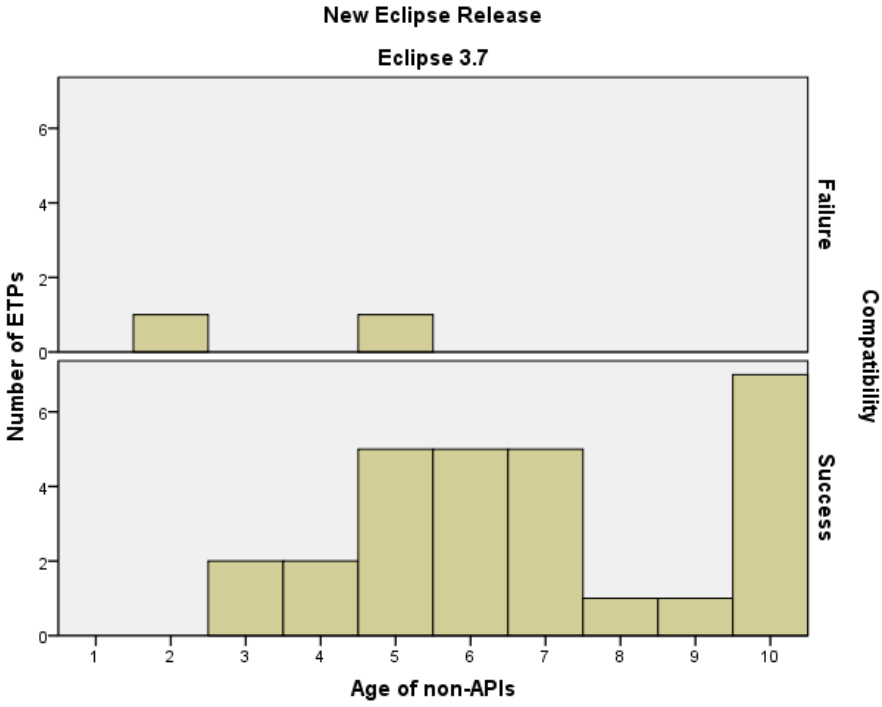
ETP Supported in Eclipse SDK 3.4



ETP Supported in Eclipse SDK 3.5



ETP Supported in Eclipse SDK 3.6



		Predictor variables								
SDK support	**New SDKs**	**1.0**	**2.0**	**2.1**	**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**
2.1	**3.0**	1	1	0						
	3.1	1	1	0						
	3.2	1	1	0						
	3.3	1	1	0						
	3.4	1	1	0						
	3.5	1	1	0						
	3.6	1	1	0						
	3.7	1	1	0						
3.0	**3.1**	1	1	0	0					
	3.2	1	1	0	0					
	3.3	1	1	0	0					
	3.4	0	1	0	1					
	3.5	0	1	0	1					
	3.6	0	1	0	1					
	3.7	0	1	0	1					
3.1	**3.2**	0	1	0	0	0				
	3.3	1	1	0	1	0				
	3.4	1	1	0	1	0				
	3.5	1	1	0	1	0				
	3.6	1	1	0	1	0				
	3.7	1	1	0	1	0				
3.2	**3.3**	1	1	0	0	1	1			
	3.4	1	0	1	0	0	1			
	3.5	0	0	0	1	0	0			
	3.6	0	0	0	1	0	0			
	3.7	1	0	0	1	0	0			
3.3	**3.4**	1	0	0	1	0	0	0		
	3.5	1	0	0	1	1	0	0		
	3.6	1	0	0	1	1	0	0		
	3.7	0	1	0	1	0	0	0		
3.4	**3.5**	1	0	0	1	0	0	0	0	
	3.6	1	0	0	1	0	0	0	0	
	3.7	0	0	0	1	0	0	0	0	
3.5	**3.6**	0	0	1	0	0	1	0	0	0
	3.7	1	1	0	1	0	1	0	0	0
Selected significant		**25**	**24**	**2**	**20**	**3**	**4**	**0**	**0**	**0**
Total selected		**35**	**35**	**35**	**27**	**20**	**14**	**9**	**5**	**2**
Percentage		**71,4**	**68,6**	**5,7**	**74,1**	**15,0**	**28,6**	**0,0**	**0,0**	**0,0**

Each SDK support has got a model in the new SDKs
1 = predictor variable significant in the model
0 = predictor variable insignificant in the model

Percentage = Selected Significant/Total Selected %

Figure 1: The frequency of the predictor variables in the models. Significant–number of times a predictor variable was considered significant,
Total–number of times a predictor was used to build the model.

		3.0		3.1		3.2-3.7	
		β	Sig	β	Sig	β	Sig
2.1	1.0	**-0.318**	**.089**	**-0.747**	**.122**	**-0.592**	**.160**
	2.0	**-0.388**	**.101**	**-1.133**	**.175**	**-0.886**	**.227**
	2.1	–	–	–	–	–	–
	const	**2.301**	**.008**	**1.472**	**.149**	**0.899**	**.327**

Table II: Prediction variables for ETPs supported in Eclipse 2.1 in in new Eclipse releases

		3.1		3.2-3.3		3.4		3.5		3.6-3.7	
		β	Sig	β	Sig	β	Sig	β	Sig	β	Sig
3.0	1.0	**-0.342**	**.021**	**-0.257**	**.086**	-0.183	.252	-0.124	.459	-0.275	.062
	2.0	**-0.248**	**.033**	**-0.399**	**.021**	**-0.420**	**.028**	**-0.431**	**.028**	**-0.430**	**.032**
	2.1	0.098	.798	0.388	.353	0.282	.529	0.068	.891	.350	.487
	3.0	-0.028	.926	-0.302	.362	**-0.702**	**.071**	**-1.315**	**.033**	**-1.669**	**.031**
	const	**1.924**	**.002**	**1.270**	**.025**	**0.944**	**.044**	**1.057**	**.032**	**1.008**	**.042**

Table III: Prediction variables for ETPs supported in Eclipse 3.0 in in new Eclipse releases

		3.2		3.3-3.7	
		β	Sig	β	Sig
3.1	1.0	-0.320	.211	**-1.178**	**.027**
	2.0	**-0.786**	**.017**	**-1.262**	**.015**
	2.1	-1.418	.198	-1.018	.387
	3.0	-0.220	.532	**-1.102**	**.065**
	3.1	0.383	.677	–	–
	const	**0.972**	**.071**	**5.409**	**.010**

Table IV: Prediction variables for ETPs supported in Eclipse 3.1 in in new Eclipse releases

		3.3		3.4		3.5		3.6-3.7	
		β	Sig	β	Sig	β	Sig	β	Sig
3.2	1.0	**-0.366**	**.025**	**-0.324**	**.033**	-0.184	.351	-0.130	.473
	2.0	**-0.368**	**.106**	-0.021	.836	-0.107	.649	-0.092	.651
	2.1	-1.169	.222	**-2.588**	**.012**	-1.280	.223	-1.075	.292
	3.0	0.755	.149	0.111	.810	**-2.025**	**.002**	**-1.804**	**.004**
	3.1	**1.503**	**.050**	0.318	.394	0.128	.892	0.034	.970
	3.2	**-2.550**	**.038**	**-2.944**	**.027**	–	–	–	–
	const	**2.201**	**.004**	**2.234**	**.004**	**1.349**	**.018**	**1.047**	**.048**

Table V: Prediction variables for ETPs supported in Eclipse 3.2 in in new Eclipse releases

		3.4		3.5		3.6		3.7	
		β	Sig	β	Sig	β	Sig	β	Sig
3.3	1.0	**-0.397**	**.009**	**-0.302**	**.073**	**-0.252**	**.128**	-0.237	.183
	2.0	-0.005	.995	-0.023	.978	0.039	.878	**-0.905**	**.039**
	2.1	-0.669	.527	-0.344	.778	-0.131	.920	1.024	.505
	3.0	**-0.457**	**.034**	**-0.681**	**.038**	**-0.884**	**.027**	**-1.522**	**.029**
	3.1	0.265	.371	**-0.666**	**.109**	**-0.617**	**.155**	-0.397	.489
	3.2	-0.145	.746	0.091	.867	0.431	.464	0.499	.633
	3.3	-1.729	.344	–	–	–	–	–	–
	const	**3.267**	**.001**	**3.241**	**.003**	**3.008**	**.003**	**3.033**	**.002**

Table VI: Prediction variables for ETPs supported in Eclipse 3.3 in in new Eclipse releases. Variable 3.3 was omitted in 3.5-3.7 since it had very high S.E (indicates multi-collinearity between the predictors)

		3.5		3.6		3.7	
		β	Sig	β	Sig	β	Sig
3.4	1.0	**-0.462**	**.064**	**-0.235**	**.106**	-0.119	.336
	2.0	-0.840	.708	-0.109	.459	-0.086	.490
	2.1	-1.240	.382	0.137	.886	0.343	.649
	3.0	**-0.788**	**.065**	**-0.575**	**.035**	**-0.603**	**.008**
	3.1	0.903	.225	0.126	.729	-0.131	.717
	3.2	–	–	-0.009	.983	0.115	.782
	3.3	–	–	–	–	–	–
	3.4	–	–	–	–	–	–
	const	**6.770**	**.023**	**4.091**	**.001**	**2.480**	**.000**

Table VII: Prediction variables for ETPs supported in Eclipse 3.4 in in new Eclipse releases. Variable 3.3 was omitted in 3.5-3.7 since it had very high S.E (indicates multi-collinearity between the predictors)

		3.6		3.7	
		β	Sig	β	Sig
3.5	1.0	-0.139	.393	**-0.927**	**.067**
	2.0	-0.141	.243	**-0.528**	**.077**
	2.1	**-2.042**	**.068**	0.422	.904
	3.0	-0.050	.930	**0.867**	**.147**
	3.1	0.056	.883	-0.564	.373
	3.2	**-3.018**	**.028**	**-3.583**	**.088**
	3.3	1.274	.370	1.701	.554
	3.4	-0.973	.316	-1.309	.381
	3.5	–	–	–	–
	const	**3.293**	**.005**	**6.025**	**.020**

Table VIII: Prediction variables for ETPs supported in Eclipse 3.5 in in new Eclipse releases. None of the non-APIs present in the ETPs was introduced in Eclipse 3.5

		New SDK releases															
		3.0		3.1		3.2		3.3		3.4		3.5		3.6		3.7	
		S	F	S	F	S	F	S	F	S	F	S	F	S	F	S	F
2.1	S	16	2	6	3	5	3	5	3	5	3	5	3	5	3	5	3
	F	4	7	1	19	2	19	2	19	2	19	2	19	2	19	2	19
3.0	S			24	4	16	6	16	6	17	4	16	4	16	3	16	3
	F			7	13	10	16	10	16	7	20	7	21	7	22	7	22
3.1	S					12	3	12	2	12	2	12	2	12	2	12	2
	F					6	13	1	19	1	19	1	19	1	19	1	19
3.2	S							22	2	15	4	14	3	13	3	13	3
	F							5	11	3	18	3	20	4	20	4	20
3.3	S									22	3	20	2	20	1	18	2
	F									6	7	3	13	3	14	3	15
3.4	S											30	1	29	1	26	2
	F											2	3	3	3	5	3
3.5	S													24	0	23	0
	F													4	5	1	9

Table IX: Classification results for model training. S–Compatibility, F–Incompatibility

	3.0			3.1			3.3			3.3			3.4			3.5			3.6			3.7		
	A	P	R	A	P	R	A	P	R	A	P	R	A	P	R	A	P	R	A	P	R	A	P	R
2.1	79	89	80	86	67	86	83	63	71	83	63	71	83	63	71	83	63	71	83	63	71	83	63	71
3.0				77	86	77	67	73	62	67	73	62	77	81	71	77	80	70	79	84	70	79	84	70
3.1							74	80	67	91	86	92	91	86	92	91	86	92	91	86	92	91	86	92
3.2										83	92	81	83	79	83	85	82	82	83	81	76	83	81	76
3.3													76	88	79	87	95	87	89	95	87	87	90	86
3.4																92	97	94	89	97	91	81	93	84
3.5																			88	100	86	97	100	96

Table X: Error analysis for model training. A–Accuracy, P–Precision and R–Recall

			New SDK releases													
			3.1		3.2		3.3		3.4		3.5		3.6		3.7	
			S	F	S	F	S	F	S	F	S	F	S	F	S	F
3.0	OSP	S	9	0	7	1	7	1	8	0	8	0	8	0	8	0
		F	6	3	6	4	6	4	8	2	8	2	8	2	8	2
	AP	S	9	0	8	0	8	0	8	0	8	0	8	0	8	0
		F	6	3	5	5	5	5	5	5	7	3	5	5	5	5
3.1	OSP	S			17	13	22	8	22	8	22	8	22	8	22	8
		F			4	3	5	2	5	2	5	2	5	2	5	2
	AP	S			8	22	22	8	22	8	22	8	22	8	22	8
		F			3	4	5	2	5	2	5	2	5	2	5	2
3.2	OSP	S					41	3	37	6	35	7	35	6	35	6
		F					0	1	0	2	0	3	0	4	0	4
	AP	S					41	3	38	5	31	11	31	10	31	10
		F					0	1	0	2	0	3	0	4	0	4
3.3	OSP	S							61	1	59	0	56	1	53	4
		F							3	3	5	4	6	5	5	6
	AP	S							61	1	59	0	56	1	53	4
		F							3	3	4	5	6	5	2	9
3.4	OSP	S									82	0	79	0	79	0
		F									5	0	6	2	6	2
	AP	S									81	1	79	0	79	0
		F									3	2	6	2	4	4
3.5	OSP	S											102	6	96	9
		F											4	0	4	3
	AP	S											97	11	96	9
		F											3	1	3	4

Table XI: Classification results for model testing. S–Compatibility, F–Incompatibility, OSP–Only significant predictors, AP–all predictors ignoring the significance.

		3.1			3.2			3.3			3.4			3.5			3.6			3.7		
		A	P	R	A	P	R	A	P	R	A	P	R	A	P	R	A	P	R	A	P	R
3.0	OSP	67	100	60	61	88	54	61	88	54	56	100	50	56	100	50	56	100	50	56	100	50
	AP	67	100	60	72	100	62	72	100	62	72	100	62	61	100	53	72	100	62	72	100	62
3.1	OSP				54	57	81	65	73	82	65	73	82	65	73	82	65	73	82	65	73	82
	AP				32	27	73	73	82	65	73	82	65	73	82	65	73	82	65	73	82	65
3.2	OSP							93	93	100	87	86	100	84	83	100	87	86	100	87	86	100
	AP							91	93	98	89	88	100	76	74	100	76	76	100	76	76	100
3.3	OSP										94	98	95	93	100	92	90	98	90	87	93	91
	AP										94	98	95	94	100	94	90	98	90	91	93	96
3.4	OSP													94	100	94	93	100	93	93	100	93
	AP													95	98	96	93	100	93	95	100	95
3.5	OSP																91	94	96	88	91	96
	AP																88	90	97	89	91	97

Table XII: Error analysis for model testing. A–Accuracy, P–Precision and R–Recall, OSP–Only significant predictors, AP–all predictors ignoring the significance.

Appendix D

Developer Survey

This Appendix presents survey questionnaire accompanying the Chapter 7 on the "Analyzing the Eclipse API Usage: Putting the Developer in the Loop"

Eclipse API Usage Survey

Dear Eclipse product/solution developer,

We are carrying out research about Eclipse API usage in Eclipse products/solutions as part of a PhD research. The aim of the research is to understand the evolution of third-party plug-ins in relation to Eclipse API usage. Your input is requested to guide us in validating our research hypotheses .

We intend to come up with a scientific publication to be submitted to an international conference/workshop/journal after analysis of the results. In addition, a summary of anonymized results together with the raw data will be published on the internet as soon as the study is finished.

The survey takes a maximum of 15 minutes to complete.

There are 34 questions in this survey

Experience of Respondent

1 [1]How many years of education after secondary school graduation do you have? *

Please choose **only one** of the following:

- ○ < 2 years
- ○ >= 2 AND < 4 years
- ○ >= 4 years AND < 6 years
- ○ >= 6 years AND < 8 years
- ○ >= 8 years AND < 10 years
- ○ >= 10 years

2 [2]How many years of experience as a software developer do you have? *

Please choose **only one** of the following:

- ○ < 2 years
- ○ >= 2 years AND < 4 years
- ○ >= 4 Years AND < 6 years
- ○ >= 6 years AND < 8 years
- ○ >= 8 years AND < 10 years
- ○ >= 10 years

3 [3]How many years of experience as a Java developer do you have? *

Please choose **only one** of the following:

- ○ < 2 years
- ○ >= 2 AND < 4 years
- ○ >= 4 years AND < 6 years
- ○ >= 6 years AND < 8 years
- ○ >= 8 years AND < 10 years
- ○ >= 10 years

4 [4]How many years of experience as an Eclipse product/solution developer do you have? *

Please choose **only one** of the following:

- ○ < 2 years
- ○ >= 2 years AND < 4 years
- ○ >= 4 years AND < 6 years
- ○ >= 6 years AND < 8 years
- ○ >= 8 years AND < 10 years
- ○ >= 10 years

Eclipse product development

5 [5]What is the name of the Eclipse product you developed? If you developed more than one Eclipse product, choose the one where you have been most involved with respect to releasing new versions. The next questions will be referring to this choice of the product. *

Please write your answer here:

6 [6]Why did you develop your Eclipse product? *

Please choose **only one** of the following:

- ○ It is a full-time paid job
- ○ It is a part-time paid job
- ○ Developing plug-ins is a hobby
- ○ It is of an educational course
- ○ Other

7 [7]What is the licence type of your Eclipse product? *

Please choose **only one** of the following:

- ○ Open Source
- ○ Commercial

8 [8]Where did you publish the source code of your Eclipse product? *

Only answer this question if the following conditions are met:
° Answer was 'Open Source' at question '7 [7]' (What is the licence type of your Eclipse product?)

Please choose **only one** of the following:

- ○ SourceForge
- ○ Google Code
- ○ GitHub
- ○ Other [ ]

9 [9]

What is the average size of your Eclipse product development team?

*

Please choose **only one** of the following:

- ○ 1 person
- ○ 2 to 4 persons
- ○ 5 to 7 persons
- ○ 8 to 10 persons
- ○ more than 10 persons

10 [10]How important is it to update your Eclipse product in relation to new versions of Eclipse SDK APIs? *

Please choose **only one** of the following:

- ○ Very important
- ○ Moderately important
- ○ Not important

11 [11]Please give reasons for your choice of importance of updating your Eclipse product *

Please write your answer here:

12 [12]How many new versions of your Eclipse product have you released since the initial release? *

Please choose **only one** of the following:

- ○ None
- ○ 1 to 4
- ○ 5 to 10
- ○ 11 to 15
- ○ 15 to 20
- ○ 20 to 25
- ○ More than 25

13 [13]What is the size (estimate) of the latest version of your Eclipse product in terms of number of files? *

Please choose **only one** of the following:

- ○ less than 100 files
- ○ 101 - 500 files
- ○ 501 - 1000 files
- ○ more than 1000 files

14 [14]On average, how many hours a week do you dedicate updating your Eclipse product in relation to Eclipse SDK API usage? *

Please choose **only one** of the following:

- ○ < 10 hours
- ○ >= 10 hours AND < 20 hours
- ○ >= 20 hours AND < 30 hours
- ○ >= 30 hours AND < 40 hours
- ○ >= 40 hours
- ○ Other

Eclipse Interfaces

15 [15]Are you aware of the "Eclipse Provisional API Guidelines"? *

Please choose **only one** of the following:

- ○ This is the first time I have heard of them
- ○ I am aware of them

16 [16]Do you follow "Eclipse Provisional API Guidelines" when developing your Eclipse product(s)? *

Please choose **only one** of the following:

- ○ Always
- ○ Sometimes
- ○ Never
- ○ I don't know, because I don't know the guidelines.

17 [17]Why don't you always follow the guidelines? *

Only answer this question if the following conditions are met:
° Answer was 'Sometimes ' or 'Never ' at question '16 [16]' (Do you follow "Eclipse Provisional API Guidelines" when developing your Eclipse product(s)?)

Please write your answer here:

18 [18]Are you aware that Eclipse provides two types of interfaces: stable and supported (APIs) and unstable and unsupporterd interfaces (non-APIs)?
Note: non-APIs are classes and interfaces containing the sub-string internal in the package name e.g., `org.eclipse.foo.internal.* *`

Please choose **only one** of the following:

- ○ I was already aware
- ○ You have just informed me

Eclipse non-API Usage 1

19 [19]What challenges do you normally face to develop and update yourproducts while using Eclipse Interfaces? *

Please write your answer here:

Eclipse non-API usage 2

20 [20] How do you find the functionality you are interested in from the Eclipse Interfaces? *

Please choose **only one** of the following:

- ○ Using a fully-automated tool
- ○ Using a semi-automated tool
- ○ Manually
- ○ Other

21 [21]What is the name of the tool you use? *

Only answer this question if the following conditions are met:
° Answer was NOT 'Manually' at question '20 [20]' (How do you find the functionality you are interested in from the Eclipse Interfaces?)

Please write your answer here:

22 [22] What are the challenges you face in finding the functionality you require from Eclipse Interfaces? *

Please write your answer here:

Eclipse non-API usage 3

23 [23]Do you use Eclipse internal implementation (non-APIs)? *

Please choose **only one** of the following:

- ○ YES
- ○ NO

24 [24]Do you deliberately avoid using Eclipse non-APIs?

*

Only answer this question if the following conditions are met:
° Answer was 'NO' at question '23 [23]' (Do you use Eclipse internal implementation (non-APIs)?)

Please choose **only one** of the following:

- ○ YES
- ○ NO

25 [25]Please give an explanation of why you deliberately avoid the use of Eclipse non-APIs. *

Only answer this question if the following conditions are met:
° Answer was 'NO' at question '23 [23]' (Do you use Eclipse internal implementation (non-APIs)?) *and* Answer was 'YES' at question '24 [24]' (Do you deliberately avoid using Eclipse non-APIs?)

Please write your answer here:

26 [26]Do you deliberately use the non-APIs? *

Only answer this question if the following conditions are met:
° Answer was 'YES' at question '23 [23]' (Do you use Eclipse internal implementation (non-APIs)?)

Please choose **only one** of the following:

- ○ YES
- ○ NO

27 [27]

Please give an explanation of why you deliberately use Eclipse non-APIs.

*

Only answer this question if the following conditions are met:
° Answer was 'YES' at question '23 [23]' (Do you use Eclipse internal implementation (non-APIs)?) *and* Answer was 'YES' at question '26 [26]' (Do you deliberately use the non-APIs?)

Please write your answer here:

28 [28]Please give us an explanation of why you use Eclipse non-APIs.

Only answer this question if the following conditions are met:
° Answer was 'YES' at question '23 [23]' (Do you use Eclipse internal implementation (non-APIs)?) *and* Answer was 'NO' at question '26 [26]' (Do you deliberately use the non-APIs?)

Please write your answer here:

Testing Eclipse product

29 [29] On which Eclipse SDK release did you develop the first version of your Eclipse product? *

Please choose **only one** of the following:

- ○ 1.0
- ○ 2.0
- ○ 2.1
- ○ 3.0
- ○ 3.1
- ○ 3.2
- ○ 3.3
- ○ 3.4
- ○ 3.5
- ○ 3.6
- ○ 3.7
- ○ Not certain

30 [30] Please give us possible reasons why you are not certain about the SDK on which you developed your Eclipse product. *

Only answer this question if the following conditions are met:
° Answer was 'Not certain' at question '29 [29]' (On which Eclipse SDK release did you develop the first version of your Eclipse product?)

Please write your answer here:

31 [31]Have you tested your Eclipse product on newer Eclipse SDK releases, betas or milestones? *

Please choose **all** that apply:

- [ ] Yes, on the next new release
- [ ] Yes, on a release after the next or later
- [ ] Yes, on a new milestone version
- [ ] Yes, on a new beta version
- [ ] No

32 [32]Are you always/frequently testing your Eclipse product on new Eclipse SDK releases? *

Only answer this question if the following conditions are met:
° Answer was 'Yes, on a release after the next or later' or 'Yes, on the next new release' at question '31 [31]' (Have you tested your Eclipse product on newer Eclipse SDK releases, betas or milestones?)

Please choose **only one** of the following:

- ○ Yes
- ○ No

33 [33]Please give us some of the challenges you have faced in testing your product with newer Eclipse SDK releases *

Only answer this question if the following conditions are met:
° Answer was NOT 'No' at question '31 [31]' (Have you tested your Eclipse product on newer Eclipse SDK releases, betas or milestones?)

Please write your answer here:

Contact Details

34 [33] If you wish to know the results of the study and to be contacted for follow-up, please leave your e-mail or postal address below.

Please write your answer here:

Summary

Co-evolution of the Eclipse Framework and its Third-party Plug-ins

Developing an application faster without compromising its quality and that is easily maintainable and extensible is a dream for every software developer or a company involved in software development. Application frameworks are software systems developed with the main goal of helping software developers reduce development costs, achieve higher quality, and faster software development.

Although application frameworks promise a lot of benefits, these frameworks are not free of costs. Frameworks constantly evolve to improve their quality and to extend the functionality they provide to developers. However, as the framework evolves, it may change its interfaces causing the applications that depend on the changed interfaces to break when ported to the new release of the framework. The application developers need to decide whether they prefer to enjoy the benefits of a new framework release at a risk of being forced to resolve the problems introduced by the interface changes. This thesis quantifies/qualifies the challenges faced by application developers and propose solutions to the identified challenges during the evolution of an application framework ecosystem.

We consider the Eclipse application framework as our case study. The choice of the Eclipse application framework was because it is widely adopted and has been evolving for over a decade. The framework is an open-source and extensible platform that provides core services for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by wrapping their tools in pluggable components (update sites for the plug-ins into the framework), called *Eclipse plug-ins*. The plug-ins reuse and extend the functionality provided by the framework. The majority of the Eclipse plug-ins are Eclipse third-party plug-ins (ETPs) in varing sizes and domains.

To quantify/qualify the challenges faced by the ETP developers, we carried out a series of empirical studies on the co-evolution of the Eclipse framework and its third-party plug-ins (ETPs). The research was carried out incrementally as follows:

First, we carried out a exploratory study on the evolution of ETPs. In this study we wanted know how the evolution of ETPs compare to the known evolution of software systems in general. The comparison was based on the Lehman laws of software evolution. Our findings confirm the laws of *continuing change*, *increasing complexity*, *self regulation*, and *continuing growth* when metrics related to dependencies between the plug-ins and the Eclipse architecture are considered. We could not validate the *conservation of familiarity* and *conservation of organizational stability* law, and the results for the *declining quality* law were inconclusive. We did not consider the law of *feedback system*. Our overall

observation is that the trends observed for constrained evolution of ETPs are similar to those presented by earlier research on the general evolution of software systems.

Second, like many other evolving software systems, the Eclipse platform has both stable and supported interfaces (APIs or *good interfaces*) and unstable, discouraged and unsupported interfaces (non-APIs or *bad interfaces*). However, despite being discouraged by Eclipse, in our experience, the usage of *bad interfaces* is relatively common in practice. For this reason, we wanted to investigate: 1) to what extent ETPs depend on *bad interfaces*, 2) whether developers continue to use *bad interfaces*, and 3), whether there are differences between the ETPs that use non-APIs and those that do not. Furthermore, we also investigated the commonly used *bad interfaces*. To perform these investigations, we conducted empirical studies based on 512 ETPs, altogether having 1,873 versions. We discovered that 44% of the 512 analyzed ETPs depends on *bad interfaces* and that developers continue to use *bad interfaces*. The empirical study also shows that plug-ins that use or extend at least one *bad interface* are larger and use more functionality from Eclipse than those that only use *good interfaces*. Furthermore, the findings show that the ETPs use a diverse set of *bad interfaces*.

Third, the findings about the two groups of ETPs that use the good and bad interfaces above, raised further questions. We wanted to get more insights in how the two groups of ETPs are affected during the evolution of Eclipse. We decided to investigate how the compatibility of ETPs that depend solely on *good interfaces* compares to that of ETPs that depend on at least one *bad interface* in new Eclipse SDK releases. We observed that: ETPs that depend solely on *good interfaces* have a very high source compatibility success rate compared to those that depend on at least one bad interface. However, we have also observed that recently released plug-ins that depend on *bad interfaces* also have a very high forward source compatibility success rate. This high source compatibility success rate is due to the dependency structure of these plug-ins: recently released plug-ins that depend on *bad interfaces* predominantly depend on old Eclipse *bad interfaces* rather than on newly introduced ones. Furthermore, we observed that the *bad interfaces* are being eliminated from the ETPs' source code because they cause, or ETP developers believe them to cause, incompatibilities when a version of the ETP is ported to new SDK release.

Fourth, we wanted to understand the factors that cause incompatibilities as a step to solving these incompatibilities. We carried out an empirical investigation on 11 Eclipse SDK releases (1.0 to 3.7) and 288 ETPs with two main goals: First, to determine the relationship between the age of Eclipse *bad interfaces* used by an ETP and the compatibility of the ETP. We found that third-party plug-in that only use old non-APIs have a high chance of compatibility success in new SDK releases compared to those that use at least one newly introduced non-API. Second, our goal was to build and test a predictive model for the compatibility of an ETP, supported in a given SDK release in a newer SDK release. Our findings produced 35 statistically significant prediction models having high values of the strength of the relationship between the predictors and the prediction (logistic regression R^2 of up to 0.810). Finally, the results from model testing indicate high values of up to 100% of precision and recall and up to 98% of accuracy of the predictions.

Finally, we wanted to understand the state-of-practice of Eclipse interface usage by the developers of ETPs. We conducted a survey on Eclipse interface usage in which 30 Eclipse product developers took part. The results from the analysis reveal a number of findings: 1) We observed that the experience of an ETP developer plays a major role in the development and maintenance of the ETPs. We have observed that developers with a level of education of up to master degree have a tendency not to read product

manuals/guidelines. 2) While for less experienced developers instability of the *bad interfaces* overshadows their benefits, more experienced developers prefer to enjoy the benefits of non-APIs despite the instability problem. Finally, we have observed that there are no significant differences between Open Source and commercial Eclipse products in terms of awareness of Eclipse guidelines and interfaces, Eclipse product size and the updating of Eclipse products in the new SDK releases.

To summarize, the research conducted in this thesis aims at quantifying/qualifying the challenges faced by software developers building systems on top of application frameworks, in general, and specifically, Eclipse. We have observed that the use of bad Eclipse interfaces, i.e., unstable, discouraged and unsupported interfaces, results in incompatibilities of the systems built on top of Eclipse and the new releases of Eclipse. Despite major maintainability risks inherent to the use of bad interfaces, developers prefer to use them due to uniqueness of the functionality provided by these interfaces. Furthermore, we have observed that older bad interfaces are less likely to introduce incompatibilities. We have developed a statistical model predicting the likelihood of a system to remain compatible with a new release of Eclipse. Using this model both the system developers and the system users can decide whether to upgrade Eclipse to a new release.

Curriculum Vitae

Personal Information

Name: John Businge
Date of birth: May 23, 1979
Place of birth: Fort Portal, Uganda

Education

Masters of Computing Science *2004–2006*
University of Groningen
Groningen, The Netherlands

Bachelor of Computer Science *1998–2002*
Makerere University
Kampala, Uganda

Secondary school *1992–1998*
Nyakasura School
Fort Portal, Uganda

Professional Experience

Ph.D. candidate *2009–2013*
Eindhoven University of Technology
Eindhoven, The Netherlands

Lecturer *2006–2009*
Mbarara University of Science and Technology
Mbarara, Uganda

Assistant Lecturer *2002–2004*
Mbarara University of Science and Technology
Mbarara, Uganda

Titles in the IPA Dissertation Series since 2007

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multidisciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09