

Diploma Thesis

CPC

an Eclipse framework for automated clone life cycle tracking and
update anomaly detection

Thesis

Valentin Weckerle
Freie Universität Berlin

<http://cpc.anetwork.de>

January 23, 2008

This thesis consists of five parts:

1. A printed copy of the *Thesis (this document)*
2. A printed copy of the *CPC Core API Specification* booklet
3. A printed copy of the *Appendix and Recommended Readings* booklet
4. A CD-ROM containing CPC sources, binaries and additional documentation
5. The official CPC website at: <http://cpc.anetwork.de>

Acknowledgements:

I want to thank my supervisors Prof. Lutz Prechelt and Prof. Elfriede Fehr for their time.

Special thanks go to Prof. Stan Jarzabek, Damith Chatura Rajapakse and Hamid Abdul Basit at the National University of Singapore for their valuable feedback and support.

Thanks also go to Sebastian Jekutsch, Christopher Oetzbeck, Marian Schwarz, Benjamin Schröter, Nicolai Kamenzky, Maximilian Höflich and Ulrich Stärk at the Freie Universität Berlin.

Furthermore, I'd like to thank Patricia Jablonski and Daqing Hou at the Clarkson University for their feedback.

Abstract

This thesis covers the development of *Copy-Paste-Change (CPC)*, a framework for copy and paste clone¹ tracking and update anomaly² warnings within the Eclipse IDE.

CPC represents the first step towards an integrated and feature rich clone tracking environment which increases the general awareness about clones in a software system and provides notifications and warnings about potential clone related errors.

It is our hope that *CPC* will provide in-depth data about the day to day copy and paste habits of programmers in real environments which can help to improve our overall understanding of the ‘Micro-process of Software Development’, the small day to day activities of a developer.

CPC is written in *Java* 1.5 and is licensed under the *GPL*. It can be obtained from <http://cpc.anetwork.de>.

¹Clones are duplicated source code fragments within a software application.

²Update anomalies can occur if a modification to the content of a clone is not propagated consistently to all other copies of the source code fragment. A typical example are defect corrections which usually need to be applied to all copies of the defective code section. A developer can easily forget to update some of the copies.

A journey of a thousand miles begins with a single step.

Contents

1	Introduction	9
1.1	The Micro-process and ECG	9
1.2	Terminology	9
1.3	Clone Research	10
1.3.1	Pervasiveness of Cloning	10
1.3.2	Copy and Paste	11
1.3.3	Risks and Benefits	11
1.3.4	Summary and Conclusion	12
1.4	Goals of this Thesis	13
1.5	Outline of this Thesis	13
2	Requirements	15
2.1	Vision	15
2.2	Related Work	17
2.2.1	CnP and CReN	18
2.2.2	CloneTracker	18
2.2.3	CbR - Clone-based Reengineering	19
2.2.4	Others	20
2.2.5	Related Work at FU Berlin	20
2.3	Requirements for CPC	21
2.3.1	Potential Extensions	21
2.3.2	Requirements	23
2.3.3	Limitations	25
3	Design and Implementation	27
3.1	The Eclipse Platform	27
3.2	Generic Design Goals and Approaches	28
3.3	The CPC Core	32
3.3.1	Service Provider API	33
3.3.2	Event Hub API	34
3.3.3	Clone Data Objects	37
3.4	The CPC Modules	40
3.4.1	CPC Sensor - Eclipse Event Hooks	40

3.4.2	CPC Track - Clone Tracking	41
3.4.3	CPC Store - Data Persistence	43
3.4.4	CPC Mapping - Data Mapping	46
3.4.5	CPC Classification - Clone Categorisation	46
3.4.6	CPC Similarity - Semantic Equivalence and Differences	47
3.4.7	CPC Notification - Clone Modification Warnings	47
3.4.8	CPC Reconciler - External Modification Handling	48
3.4.9	CPC Store Remote - Remote Synchronisation	50
3.4.10	CPC Imports and Exports	54
3.4.11	CPC UI and Notifications UI	57
4	Heuristics	61
4.1	Clone Classification	61
4.2	Clone Similarity	63
4.3	Clone Modification Notifications	66
4.3.1	Modification Evaluation	66
4.3.2	Delayed Notifications	69
5	Challenges and Setbacks	71
5.1	Planning and Risk Assessment	71
5.2	Reuse and Performance	72
5.3	The Eclipse API	73
5.3.1	General Complexity and Documentation	73
5.3.2	Inconsistent, Inappropriate or Missing APIs	75
5.3.3	Conservative Development	79
5.4	Team Providers	80
5.5	Failures, Defects and Solutions	84
6	Testing and Evaluation	87
6.1	Testing	87
6.1.1	Testing and Debugging Support	87
6.1.2	Unit Testing	88
6.2	Evaluation	89
6.2.1	Survey of Existing Data	89
6.2.2	CPC	93
7	Conclusion and Future Work	99
7.1	Looking back	99
7.2	Looking ahead	100
7.3	Conclusion	101
	Bibliography	103

List of Figures

3.1	Basic layout of the Eclipse environment (<i>simplified</i>)	28
3.2	COD: Example of design for multi-level reuse in CPC	30
3.3	COD: The CPC Core module	32
3.4	COD: The CPC service provider concept	33
3.5	SED: Provider registry usage for initial lookup of a singleton provider	34
3.6	COD: The CPC event hub concept	35
3.7	SED: Event hub registry usage with synchronous and asynchronous listeners	35
3.8	CLD: The CPC Event hierarchy (<i>simplified</i>)	36
3.9	The CPC Clone Data Object	38
3.10	CLD: The CPC Clone Data hierarchy (<i>simplified</i>)	39
3.11	CUD: Module collaboration during the handling of a paste operation (<i>simplified</i>)	40
3.12	SMD: Internal states during cut, copy and paste events (<i>simplified</i>)	41
3.13	COD: The Store Provider concept	43
3.14	CUD: Module collaboration during the handling of a clone modification (<i>simplified</i>)	48
3.15	CUD: Module interaction during reconciliation of external modifications (<i>simplified</i>)	49
3.16	COD: Structure of the CPC Remote Synchronisation framework (<i>simplified</i>)	51
3.17	COD: Structure of the CPC Imports/Exports framework (<i>simplified</i>)	55
3.18	CPC UI clone marking via coloured bars (<i>rulers</i>)	57
3.19	Actions available via the notification icon in the editor ruler	58
3.20	CPC notification in problems view	58
3.21	Simple CPC clone data in list viewer	59
3.22	CPC clone data tree viewer	59
3.23	CPC clone history replay view	60
3.24	Extendible CPC preferences dialogs	60
6.1	Comparison of average clone creation rate per hour	89
6.2	Comparison of clone size distribution	91
6.3	Comparison of cut, copy and paste event distribution	92
6.4	Comparison of clone size distribution	93
6.5	Clone content classifications and modification states	94
6.6	Size distribution of clones by clone state	95
6.7	Clone state distribution by clone size category	95

6.8	Size of clone modifications in content length difference and Levenshtein distance . .	96
6.9	Size distribution of clone groups	96
6.10	Delay between creation of first and last clone in group	96
6.11	Delay in hours between clone creation and last modification	97
6.12	Number of modifications made to a clone's content	97
6.13	Size distribution of modifications made to a clone's content	97

For UML diagrams, the type is indicated by the first three characters.

COD: Component Diagram, SED: Sequence Diagram, CLD: Class Diagram

CUD: Communication Diagram, SMD: State Machine Diagram

Chapter 1

Introduction

1.1 The Micro-process and ECG

The ‘Micro-process of Software Development’ represents one of the areas of interest of the software engineering research group of the Department of Computer Science at the Freie Universität Berlin [32, 33]. Sometimes also called ‘Actual Process’, the research focuses on the small, every day actions of the programmer (*i.e. browsing code or documentation, modifying a method, copying text, ...*) and sequences of such actions, the so called *episodes*.

In order to obtain very fine grained data about such actions and episodes, the *ECG Lab* toolkit¹ has been developed [53, 52]. It uses a set of sensors to automatically collect data about programmer actions from multiple sources. The main sensor instruments the Eclipse IDE and records all interactions between the programmer and the IDE. These automatic data collection methods have been used extensively to gather data in a number of different experiments.

However, the micro-process research is still in its infancy and it remains unclear what benefits might someday be derived from it. The focus of this thesis and its predecessor [46] is therefore only a very narrow part of the micro-process: The programmer’s copy and paste actions and the resulting source code duplications (*clones*) in the software system under construction. The so called ‘Copy, Paste, Change Episodes’.

We believe that by narrowing down the focus in this way, it is possible to attain a small but meaningful part of the benefits which might someday arise out of the micro-process research, today.

1.2 Terminology

Similarities and duplications are a common phenomenon in the source code of most software applications. These so called ‘clones’ often become a hindrance during software maintenance and have emerged as a controversial research topic.

The terminology in the field of clone research has traditionally been very inhomogeneous and the research community has not yet adopted a common definition of a ‘clone’ [51]. The precursor to this thesis has made some effort to specify a notation for clone descriptions [46].

¹ElectroCodeoGram

In the area of static clone detection the question of what constitutes a clone and what doesn't tends to be strongly intertwined with the clone detection approach used. This approach mainly stems from the often very limited availability of information concerning the evolution of similar source code which makes it hard to specify exactly what was copied from where or whether observed similarities might be purely accidental.

The narrow focus on a programmer's copy and paste actions and the resulting clones adopted in this thesis allows for a more intuitive approach to cloning. All source code which is duplicated due to a copy and paste episode of the programmer can safely be considered a clone. Thus some of the challenging problems of static clone detection are avoided, while others are only postponed. One of the issues discussed in this thesis, deciding at which point two individually evolving copies of the same source code should no longer be treated as clones, can be seen as one of these 'postponed' problems.

For the remainder of this thesis we thus use the term **clone** to describe a section of source code which was copy and pasted from one location to another. The origin of a copy and paste action is referred to as the **origin clone**. Clones which were copied from the same location are considered to form a **clone group**. If a new clone is created by copying an existing clone, the new clone is considered to be a member of the clone group of the existing clone.

During the evolution of a software system clones may cease to be part of any clone group (*i.e. because the origin clone and all other clone group members were deleted*). While such clones do no longer constitute 'clones' in the basic sense, as there is no longer any source code duplication, it may at times still be beneficial to take them into consideration. We call such specimen **orphaned clones**.

Furthermore, clones which were introduced into a software system before the adoption of a clone tracking tool will be referred to as **legacy clones**. Similar code sections which only appear to be clones are termed **accidental clones**, these will be covered in the next section.

Activities of a developer which modify only a part of the members of a clone group can lead to **update anomalies**. Update anomalies occur when the semantics of members of a clone group diverge unintentionally and represent one of the biggest clone related problems during software maintenance.

1.3 Clone Research

Duplication of source code in software systems has been an active field of research for more than a decade. Roy and Cordy as well as Tairas provide a good overview of the cloning related publications to date [51, 55].

1.3.1 Pervasiveness of Cloning

Past and current research findings strongly indicate that cloning in software applications is a pervasive phenomenon [24, 36]. A study of the authors of *CP-Miner* found a considerable amount of cloning in some major open source projects [42]. The study paints an interestingly homogeneous picture of the clone coverage across very different software projects. The examined systems ranged

from operating systems like *Linux* (version 2.6.6, 4,365 kLOC, 22.3% cloning) and *FreeBSD* (version 5.2.1, 3,299 kLOC, 20.4% cloning) to web servers like *Apache* (version 2.0.49, 223 kLOC, 17.7% cloning) and data base systems like *PostgreSQL* (version 7.4.2, 458 kLOC, 22.2% cloning).

Other studies confirm these numbers and some uncover even higher cloning rates in specific application domains. An investigation into cloning in web applications by the authors of *XVCL* examined 17 applications of different sizes, based on different technologies and found clone coverages of 17 to 63% (average 41%) [50].

1.3.2 Copy and Paste

In comparison with static clone detection techniques, copy and paste actions by individual programmers have received only little attention by the research community. The amount of available empirical findings is thus very limited.

Kim et al. conducted an ethnographic study focusing on copy and paste practices [38]. After observing nine professional programmers for 60 hours (*50h automated, similar to ECG; 10h manually*), they found that the programmers mainly copied very small code parts (*74% less than a line, 17% blocks, 8% methods, 1% classes*). However, they also observed roughly four non-trivial copy and paste clones per hour of development. During this relatively short study, programmers applied changes consistently to all clone instances. Clones created during this study were also often removed due to refactoring or other, unrelated code changes. When interviewed, programmers said that they deliberately delayed code restructuring until a clone was copied multiple times in order to discover the right level of abstraction. The authors also argue that some copy and paste actions have the potential to capture important design decisions.

Kim et al. introduce the notion of **structural templates** which they divide into **syntactic templates** and **semantic templates** depending on whether the programmer intends to reuse the semantics of the copied code or whether only a particular syntax is of interest. **Semantic templates** are subdivided into a number of finer categories. While this typology may be vague and very hard to decide programmatically in many situations, it provides a useful conceptual tool when considering the importance of clones and their evolution (*see section 4.1*).

1.3.3 Risks and Benefits

The presence of clones in applications has long been considered to be an indication of poor software quality. Clones were purely regarded to be ‘bad smells’ which hinder program understanding and increase maintenance costs [26, 51]. Aggressive removal of clones with support of automated clone detection and refactoring approaches has thus received a lot of attention [51].

In light of newer research findings this tough stance on cloning has been reconsidered and there are now many advocates of a more lenient approach to clone removal. It is argued that limitations of the programming language often make cloning impossible to avoid [38, 40]. Basit et al. examined code cloning in the *Java Buffer Library* and the *C++ Standard Template Library (STL)* and identified a large number of clones which could not be removed with the normal mechanisms offered by *Java* or *C++* [17, 18]. Unavoidable cloning may also be caused by the presence of conflicting design goals.

Some studies have shown that programmers create clones intentionally [22, 37, 38]. It is argued that cloning can be viable design decision in certain scenarios where issues such as code stability, risk minimisation, design complexity, experimental development, performance or code ownership are important considerations. Roy and Cordy provide a good overview over the different reasons for cloning².

Furthermore, it has been observed that the majority of clone groups tend to be maintained consistently [14, 40]. In small, single contributor software systems clones are maintained even more consistently [14]. Newly introduced clones are also often volatile. They tend to either be removed or to evolve independently within a short period of time (*48-72% disappeared within 8 checkins*) [40]. Aggressive, immediate refactoring of clones may thus be neither necessary nor beneficial.

It has been noted that the use of static clone detection techniques to identify clones for refactoring is likely to also identify code sections which are only accidentally similar [13]. Common usage patterns of an API, design patterns and mental templates are typical examples. Unification of such accidental clones is likely to increase the future maintenance effort as it bears the high risk of potentially propagating changes to code sections which were never meant to be affected by a change.

However, even the supporters of a more lenient approach acknowledge that clones, or at least certain types of clones, in software systems are likely to have negative effects on the long term maintenance effort. Findings show that source sections which contain clones are likely to require more modifications during maintenance than clone free segments [43].

1.3.4 Summary and Conclusion

There is wide spread support for the notion that cloning in software systems increases the long term maintenance effort. The introduction of potential update anomaly risks is one of the key arguments. At the same time recent findings suggest that there are many reasons which make cloning inevitable and sometimes even desired. Many potential remedies have been suggested, ranging from new language features to special preprocessors or meta-languages [58, 51].

For this thesis the reasons for and effects of copy and paste cloning are of chief interest. The copy and paste functionality of current software development environments has many undisputed benefits. Preventing copy and paste cloning, if that would even be desirable, by deactivating such functions, while possible, would clearly be counterproductive.

Rather than completely preventing copy and paste cloning (*or the status quo - completely ignoring it*), a more sensible approach, which has been suggested by many studies, would be to add special clone tracking features to the software development environments [30, 37, 38, 40, 44]. The idea is to retain the convenience and positive aspects of copy and paste cloning while at the same time trying to ease or prevent some of its major pitfalls. This perceived need for tool support is one of the key motivations for this thesis.

²The corresponding tree-diagram can be found in the *Appendix and Recommended Readings* booklet.

1.4 Goals of this Thesis

This thesis aims at providing a versatile and highly flexible framework for clone tracking within the Eclipse IDE named *Copy-Paste-Change*³ (*CPC*). *CPC* should provide a base for future work in the area of clone tracking and should facilitate the collection of data on typical copy and paste cloning activities of programmers. It should furthermore improve the general awareness about cloning in an application by providing visualisations of clone data and should establish a basis for future notifications of the developer about potential update anomalies. This is covered in detail in chapter 2.

1.5 Outline of this Thesis

This section shortly summarises the content of each of the remaining six parts of this thesis.

Requirements: This chapter starts off with outlining a vision for future clone tracking tool support, provides an overview of existing work in this area and closes with a list of requirements for *CPC*. Furthermore, the chapter provides examples of a number of potential future uses and extensions for *CPC* which need to be taken into consideration during requirements elicitation and lists the scope limiting assumptions made for this thesis. (*chapter 2*)

Design and Implementation: After a very brief introduction to the Eclipse platform, this chapter introduces the general design goals and approaches of *CPC*. The remainder of the chapter covers the *CPC* design and implementation in-depth by first commenting on the *CPC* core and continuing with a listing of the major *CPC* modules. (*chapter 3*)

Heuristics: Heuristics for clone classification, similarity and the detection of potential update anomalies represent one of the key aspects of *CPC*. This chapter provides a broad overview of the corresponding *CPC* modules and introduces the currently implemented heuristics as well as providing ideas for future improvement. (*chapter 4*)

Challenges and Setbacks: During the course of this thesis a number of challenges emerged and some approaches proved to be infeasible. This chapter gives a detailed account of all the major problems and their solutions. (*chapter 5*)

Testing and Evaluation: This chapter provides some insight into the testing process of *CPC* followed by a short analysis of copy and paste clone data collected in earlier experiments as well as data collected during the evaluation of *CPC*. (*chapter 6*)

Conclusion and Future Work: The final chapter summarises the work done so far, provides an outlook into the future and finishes with the overall conclusion. (*chapter 7*)

Additional material of interest can be found in the supplied *Appendix and Recommended Readings* booklet and the *CPC Core API Specification* booklet as well as on the submitted CD-ROM and the official CPC website at: <http://cpc.anetwork.de>

³Following the ‘Copy, Paste, Change Episodes’ of the ‘Micro-process of Software Development’

Chapter 2

Requirements

This chapter first provides an outlook on possible benefits of mature tool support, followed by a short overview of some of the existing tools. Finally the chapter closes with a list of requirements which can be inferred for the tool *CPC* which is to be developed over the course of this thesis.

2.1 Vision

*Optimism is an occupational hazard of programming:
feedback is the treatment.*

[Kent Beck]

There seems to be a number of potential benefits of integrated tool support for copy and paste clone tracking, as was already hinted at in the introduction. This section sums up some of the interesting long term goals for this research area. While most of these ideas are clearly beyond what can reasonably be achieved within the timeframe of this thesis, they provide long term targets which represent important considerations for the framework design.

In an *ideal* scenario we know about *all* clones, even those which were not introduced by copy and paste actions but for instance by transcribing from a print-out and those which were introduced into the software system before a clone tracking tool was first installed (*legacy clones*). Without *any* false positives.

This requirement poses two serious problems. First, while static clone detection has made considerable progress in recent years, recall and precision of the available tools are still very far away from reaching a state where these tools could be used for clone detection without risking inaccurate clone data. And second, even if the static detection techniques would work flawlessly, the matter of *accidental* clones would remain a serious concern [13]. Even in a scenario where a software system was developed entirely under the control and supervision of a clone tracking tool, the question of what constitutes a *real* clone as opposed to an *accidental* clone is hard to answer.

By giving up on the goal of covering *all* clones and instead limiting ourselves to clones which are created due to some kind of source code copying, we can circumvent many of the hard problems like *accidental* clones, while at the same time accepting the loss of potential benefits which could be gained by tracking other types of clones too. However, even with this limitation, many problems

remain. Though most of them are *only* technical issues.

For a concrete tool this would imply a need for tracking of all clones which are generated by copy and paste actions of all types as well as any other means of copying source code. Clone tracking would need to be in effect system wide. A developer might copy and paste between different applications, use different tools to modify source code or make a copy of a source file with some file management application. An all-encompassing solution would thus be to have clone tracking support either directly integrated into the operating system or in all applications which are likely to be used for source code modifications [44].

Until such a high level of clone tracking integration is reached, any clone tracking tool would need to be able to detect and recover from modifications of source code files by external applications. While such external modifications would be discouraged, as it is impossible to guarantee a perfect recovery from external edits, it would still be important to reduce the likelihood of clone data loss as much as possible.

Furthermore, the tool would need to support clone tracking in a distributed environment where multiple members of a development team are potentially modifying the source code concurrently. Clone data would need to be synchronised between the workstations of the team members and potential conflicts would need to be resolved. Conflict resolution would have to be completely automatic with a graceful fallback strategy in situations where part of the clone data conflict can not be resolved. Synchronisation would need to be limited to the periods in which repository operations are executed, as the workstations might not be continuously connected to the network. The tool should support all major repository providers and should offer a mechanism to store and synchronise the clone data via the source code repository, for ease of use. A standalone server mode for larger installations might also be beneficial.

By tracking copy and paste activities in the IDE, the absence of false positives¹ can be guaranteed. However, it may be prudent to soften this requirement in situations where the clone tracking tool was introduced late in the development cycle of a software project and where most cloning activities are thus likely to have taken place already. In such scenarios it may be beneficial to use a static clone detection utility to ‘jump start’ the clone database, accepting potential false positives.

An important aspect of any clone tracking tool would be support for good visualisations of the collected clone data. Different views of the data which are tailored to typical tasks or questions should be provided. It is crucial to make the clone data as easy to grasp as possible, even in large systems with thousands of clones. In most scenarios a developer is likely to be interested only in a small subset of the clone data. Powerful, yet easy to use, filtering mechanisms are thus essential.

If implemented correctly, such visualisations are likely to increase the general awareness of cloning in the application among all developers. Even without explicit support for warnings about update anomalies and other clone related errors, this increased general awareness might quite plausibly reduce the likelihood of any such errors. A developer might very well choose to deactivate advanced features like notifications and warnings about potential update anomalies and still benefit from the available clone data visualisations.

¹According to our clone definition all code which is copied due to cut, copy and paste operations of the developer is a clone. Whether these ‘clones’ are of interest is another matter.

The just mentioned notifications and warnings about potential errors are the by far most demanding problem to be dealt with. The tool needs to be able to make a large number of potentially very challenging decisions. This begins with the creation of a clone: *‘Should the current copy and paste action even be considered as a clone?’ ‘What kind of clone is it?’ ‘What kind of classifications are sensible in the first place?’ ‘How important is the clone?’*

And becomes even more controversial once a clone is modified: *‘Should the modification be propagated to other clones of this group?’ ‘To all of them?’ ‘Has the classification or importance of this clone changed due to the modification?’ ‘Is the clone now evolving independently and should be removed from its current group?’*

In other words the tool would need to make an ‘educated’ guess about the semantics of the copied code as well as the intentions of the developer. While a perfect solution to this problem is likely to be impossible even for the far future, it seems plausible that an approximation, at least for special cases, might be achievable. Good heuristics, artificial intelligence and special tailoring to the individual programmer might prove to be fruitful steps towards this goal.

Another question is how notifications or warnings should be displayed to the programmer and how the programmer can interact with them to correct the problems. A non-intrusive approach might be best. However, it would seem sensible to choose different ways of notifying the developer, depending on the calculated importance of the event and the confidence in its correctness. There might also be a demand for different ‘notification styles’ from which each user can pick a favourite.

It is painfully obvious that our current understanding of copy and paste cloning practices and even cloning in general is far from sufficient to implement a solution which comes even close to solving the problems posed by notifications and warnings as outlined above. Good, empirically confirmed heuristics can only be developed if a large base of real world cloning data from all kinds of different projects is available. Small, short term lab experiments with a hand full of students are clearly not an adequate substitute. A very important aspect for any early version of such a clone tracking tool would therefore be data collection. If potential users can expect some benefits even from early versions and are facing only minimal effort for the adoption of the tool, acquiring a large, very heterogeneous data set may be possible.

Such data would also be very interesting in light of other research questions. One example would be the ongoing quest for new ways of comparing different static clone detection techniques. A large project with a complete copy and paste clone history could be used as a new kind of benchmark for the recall of static clone detectors.

2.2 Related Work

This section highlights some of the existing approaches and solutions which at least partly address some of the issues pointed out in the previous section. Other interesting tools and research findings, which do not directly address our ‘vision’, are covered in the chapters for which they are relevant (*i.e. in chapter 4*).

2.2.1 CnP and CReN

In the end of 2007, Patricia Jablonski at the Clarkson University started work on a PhD thesis with the topic *Techniques for Detecting and Preventing Copy-and-Paste Errors during Software Development* [29, 30]. The goals of her dissertation proposal show a very high overlap with the issues covered in this thesis.

The proposed tool *CnP* will provide automated tool support for copy and paste tracking in the Eclipse IDE. Its main features, clone ‘detection’ based on copy and paste actions, visualisation of clone data and detection of potential cloning related inconsistencies or errors are very similar to *CPC*. However, the main focus is placed on the development and evaluation of heuristics and on empirical studies on typical copy and paste related defects.

So far no final version of *CnP* is available, it is currently in a very early stage of development and November 2009 seems to be the tentative date for the final release. A specialised, proof of concept implementation called *CReN* was presented at OOPSLA 2007 [31]. *CReN* is an Eclipse plug-in which tracks copy and paste actions and uses the abstract syntax tree (*AST*) generated by Eclipse to support the developer in the task of consistent identifier renaming. Jablonski argues that this is one of the typical copy and paste related tasks which is likely to introduce defects into a software system. The very limited focus of *CReN* and its proof of concept nature distinguishes it from *CPC*.

However, the high overlap between the proposed *CnP* tool and *CPC* strongly suggests that interesting knowledge exchange and reuse opportunities are likely to emerge as Jablonski’s dissertation progresses. Contact has been established and discussions are ongoing. *CPC* might emerge as a suitable base framework for future *CnP* development.

2.2.2 CloneTracker

In mid 2007 Duala-Ekoko and Robillard presented a clone tracking tool called *CloneTracker* [23]. Similarly to *CPC*, it is an Eclipse plug-in which is aiming at supporting the developer during software maintenance by highlighting clones and issuing warnings when a member of a clone group is modified. It furthermore supports limited linked editing of two clone instances and provides some basic visualisations for tracked clones.

CloneTracker employs a 3rd party static clone detection utility to obtain a list of potential clone instances in the system. The developer can then manually inspect the detection results and select clones which should be tracked. Copy and paste actions by the developer are not taken into account and clones found by means of the static clone detector are not automatically tracked.

Duala-Ekoko et al. use a very interesting clone tracking approach. Instead of storing line or character offsets and updating them during source modifications, they try to extract a robust meta description of the clone segment from the surrounding source code. This so called *Clone Region Descriptor (CRD)* is based on file location, file name, class name, method name (*optional*) and multiple block descriptors (*optional*) as well as some additional source code metrics. It is designed to be resilient to many types of modification potentially affecting the source file.

The advantage of this approach is that it does not require continuous tracking of clone positions

over all modifications of a document. Duala-Ekoko et al. argue that there is a high likelihood that the position of a clone can be identified even after a 3rd party modified the document.

However, as a result of the necessary trade-off between robustness to document modifications and positioning precision as well as some general characteristics of the *CRD* design, *CloneTracker* can often only approximate the position of a clone. For one, a *CRD* is always aligned with a *Java* block. A clone smaller than a block or a clone spanning multiple blocks results in a *CRD* pointing to the next higher block which completely encompasses the clone, the clone ‘grows’. Another aspect is the strong reliance on nesting levels. If the nesting depth of a source code section is increased or decreased (*i.e.* *addition/removal of a surrounding conditional statement*), all *CRDs* within the section are invalidated. Furthermore, some other specific changes to the code can also invalidate all *CRDs* nested within them. *Any* modification to the predicate expression of a conditional statement, changes to the condition of a loop and addition/removal of exceptions caught by a try/catch block are some examples of such changes.

Duala-Ekoko et al. evaluated the precision of their tracking approach on multiple revisions of a number of open source *Java* projects. They found that the *CRD* related ‘growth’ of clones is on average less than four lines and the average number of ‘missed’ lines (*lines of a clone which were lost due to specifics of the CRD design*) is less than two. However, of the 3,275 clones examined, only 164 (5%) were completely lost.

While this approach seems promising the considerable loss in precision is likely to limit its usefulness in a lot of scenarios. For the tracking of the often small and unstructured copy and paste clones typically faced by *CPC* it seems impracticable. In a fully automated clone tracking system like *CPC* the approximation of positions could also lead to confusion and mistrust of the developers once they become aware of code sections which are incorrectly marked as being part of a clone. In this sense *CRDs* introduce a new type of false positives.

2.2.3 CbR - Clone-based Reengineering

In 2003 Simon Giesecke developed the tool *CbR* as part of his master thesis *Clone-based Reengineering für Java auf der Eclipse-Plattform* [28]. *CbR* is an Eclipse plug-in for Java developers which has some overlap with the features of *CPC*. It applies static clone detection techniques to identify duplicated source code and includes basic functionalities for clone visualisation.

The main focus of *CbR* lies on the incremental update of the detected clone data. If the developer modifies part of the source code the clone data is either directly updated or, for larger edits, static clone detection is only reexecuted for those areas which are potentially affected. *CbR* relies heavily on the AST generated by Eclipse. The *CbR* implementation retains an experimental character and is presented as a basis for further work, rather than a tool which is ready for use. It was based on Eclipse 2.x and is no longer compatible with current versions of Eclipse. Development has been discontinued since 2003.

CbR does not track copy and paste actions of the developer but relies entirely on static clone detection, accepting the potential problems such an approach entails (*false positives/accidental clones/low precision, low recall*). Furthermore, it does not try to provide warnings about potential inconsistencies between members of a clone group. As such its focus is different from that of *CPC*.

2.2.4 Others

A number of other tools are either available or have been described in publications. However, compared to the tools so far presented, none of these additional tools has a similar overlap with the goals of *CPC*.

In 2004 a tool called *C4D* was described by Udo Borkowski [20]. So far no more information about this tool has been made available and the exact state and capabilities of the implementation remain unclear.

Li et al. developed the tool *CP-Miner* which combines static clone detection and identification of potential ‘copy and paste’ inconsistencies [42]. The first version of *CP-Miner* focused on detection of inconsistent identifier renaming (*like CReN*) and was able to identify 28 such defects in the *Linux* kernel as well as 23 in *FreeBSD*. However, it supports neither *real* copy and paste clone tracking nor does it provide IDE integration.

Tommim et al. described the prototype editor *Codelink*, a *XEmacs* extension which supports ‘linked editing’ of members of a specific clone group [57]. The developer manually selects similar code sections in the source code and ‘links’ them together. During this process the similarities and differences between the selected code parts are analysed. Once ‘linked’ a modification to the common block of a clone will be automatically propagated to all members of the clone group while modifications to clone specific blocks will affect only the current clone. *Codelink* also offers some features for clone visualisation and for navigation between members of a clone group. Automatically ‘linking’ clones after copy and paste actions and the use of static clone detection tools is mentioned by Tommim et al. but has not been implemented.

Furthermore there are a large number of other clone related tools with IDE integration which exclusively employ static clone detection techniques to find duplicated source code segments. Dudziak and Wloka implemented a *NetBeans* plug-in which tries to automatically detect refactoring opportunities in a software system [25]. Among other features, the tool provides support for clone identification and removal. *SimScan* by Blue Edge is commercial static clone detector which can be integrated into a number of Java IDEs [21]. It focuses solely on the detection and display of clones. *PMD* is a general purpose defect detection tool which also detects code duplications [8]. The *SDD* clone detection plug-in for Eclipse represents another static detection approach [34, 35]. It provides the base for the currently implemented *CPC* legacy clone import functionality (*see section 3.4.10*).

2.2.5 Related Work at FU Berlin

In early 2007 Sofoklis Papadopoulos submitted the diploma thesis *Verfolgen von Kodekopien zur Defektvermeidung in Eclipse* [46] with a very similar focus on copy and paste actions of the programmer as in this thesis. However, his work did not include Eclipse integration and focused solely on the development of an *ECG Lab* based detection approach for use in lab experiments.

The resulting *ECG Lab* module had multiple crucial shortcomings when considering it for general use. All tracking was line based which is too coarse for accurate tracking of clones. The implementation was furthermore monolithic in design, had performance issues and did not provide persistence for the collected clone data.

In effect this made his work only viable for small, experimental setups and provided no support for copy and paste clone tracking in real environments. Unfortunately this also meant that only a very small part of his work could be reused for this thesis.

Early versions of *CPC* were based on a heavily refactored version of the *ECG Eclipse Sensor* which was developed by Frank Schlesinger and later extended for detection of copy and paste operations by Sofoklis Papadopoulos [52]. Furthermore the copy and paste data collected with this sensor could be used to gain some general insight into copy and paste frequencies and common clone sizes (*see section 6.2.1*).

Due to some critical limitations of the *ECG Eclipse Sensor*, it had to be replaced and the current version of *CPC* does no longer include any noteworthy amount of source code of previous *ECG Lab* based works (*see section 5.2*).

2.3 Requirements for CPC

A number of the requirements for *CPC* can be directly inferred from our overall goal. Others may be less obvious. As *CPC* is only the first step towards our vision of ubiquitous clone tracking potential future extensions may also contribute further requirements. This section starts with some examples of potential future uses of *CPC*, in order to better gauge their potential impact on the requirements and design of *CPC*. After which the requirements for *CPC* are summarised and a list of limiting assumptions presented.

2.3.1 Potential Extensions

Heuristics and Visualisations

The two most probable areas of customisation are heuristics and clone data visualisations. The initial implementation of *CPC* covers these areas only to a very limited degree and further work is thus clearly needed.

Heuristics are an integral part of *CPC* which directly affect its usefulness. The current lack of cloning data and the limited general understanding of the relevant factors indicate that heuristics are likely to be improved in iterative and potentially incremental steps. This expected volatility of any heuristic implementation strongly suggests a modularised approach which minimises the coupling between heuristics and *CPC* as well as the coupling between heuristics. The three main types of heuristics; classification, similarity and modification notification are covered in more detail in chapter 4.

As was already outlined earlier, one of the crucial contributions of *CPC* is the potentially increased general awareness of clones among software developers. In order to achieve this goal, *CPC* would need to provide a variety of visualisation capabilities which could not be covered during this thesis. Visualisation contributions and their API requirements are therefore an important point to be considered.

An example for an interesting *CPC* visualisation extension would be the reuse of the *AJDT Visualiser* which is part of the *AspectJ Development Tools* project (*AJDT*) as suggested by Tairas et al. [56]. They argue that code duplication in software applications has many similarities with

aspects in an aspect oriented software system. Their clone visualisation prototype thus successfully reused a number of *AJDT* visualisations.

Change Propagation and Linked Editing

While the current implementation of *CPC* only passively warns about inconsistent clone modifications, fully automated propagation of changes to all affected clone instances and ‘linked editing’ as described by Toomim et al. represent interesting, potential additions [57].

Refactoring

Tool support for refactoring suggestions and automated refactoring has been a topic of active research for some time. So far such approaches have focused on static program analysis and clone detection [25, 51]. For some situations, the copy and paste clone data collected by *CPC* might be more suited for such a usage scenario than the output of static clone detection utilities.

In light of this possible future use for *CPC* clone data, it is important to ensure that as much of the potentially helpful types of information as possible is persisted for later use. The full modification history of each clone instance in relation to its clone group is an example of such data. The modification history provides much deeper insight into the differences and commonalities within a clone group than the simple application of a *longest common subsequence* based algorithm like the one implemented by the *Unix diff* utility.

Automated Template Extraction

Kim et al.’s observations of copy and paste practices and their classification of copy and paste clones into **syntactic templates** and **semantic templates** highlighted a critical point, programmers often copy code not to reuse its semantics but to reuse its basic structure [38]. Our personal experience and experience gained from available copy and paste data from a number of experiments also support this (*see also chapter 4 and section 6.2*).

An interesting future extension for *CPC* could be automatic detection and extraction of such templates. These could then be either added to the existing code templates of Eclipse or be made available via a separate function. Tailoring of these templates to the programming practices of the individual developer would be a key factor for the usefulness of such a feature.

XVCL Back Propagation Tool

The Software Engineering Department at the National University of Singapore (*NUS*) has investigated reasons for cloning in software applications and potential remedies for many years. Jarzabek et al. advocate the use of a language independent meta programming language called *XVCL* in order to address inevitable and desired cloning [58]. As a source code generating meta language *XVCL* is facing one of the typical problems of other code generation techniques, back porting of manual modifications of the generated output.

To address this issue, the *XVCL Back Propagation Tool (BP Tool)* was developed. It can automatically detect manual modifications made to the generated *XVCL* output and can support

the programmer in situations in which the back propagation of the changes into the meta model is intended. To allow tracking of sections and their origin within the generated output, the current implementation adds a large number of special marker comments.

While this approach works, it can lead to poor readability of the generated source code and tends to distract the developer. Furthermore, the marker comments are vulnerable to accidental modification and duplication by the programmer. Unintended removal or copying of these comments can potentially lead to incorrect back propagation recommendations and could render the *BP Tool* unusable. Thus requiring the developer to manually identify and propagate all modifications.

CPC could provide the base for the next generation of *XVCL* back propagation support. Instead of ‘instrumenting’ the generated source code with special comments, the *BP Tool* could leverage the clone tracking capabilities of *CPC*. While the basic tracking requirements would be similar to a normal *CPC* installation, the creation, display and final processing would differ considerably.

The *BP Tool* is one example from a group of potential future use cases of *CPC* which only reuse a subset of the available functionality and which might not even be related to clone tracking.

Others

The list of potentially interesting extensions is clearly not limited to the ones listed thus far. Further ideas have been suggested in different publications (*i.e. consistent identifier renaming support as implemented by CReN [31]*) and there are certainly a number of potential extensions which simply did not occur to the author thus far.

2.3.2 Requirements

*You can't please everyone
so aim to displease everyone equally.
[Joshua Bloch]*

When the original vision is combined with potential future extensions as outlined above, a large set of requirements readily emerges. However, the tight time constraints for this thesis make it crucial to sensibly prioritise these requirements and to limit the overall project scope to a manageable subset. This section identifies the main requirements for *CPC*.

Framework: The key aspect for *CPC* is extensibility and flexibility. The version implemented during this thesis can only cover a very small subset of the interesting and required functionality. From the very outset it was thus clear that only an iterative and incremental development approach could possibly lead to the intended results in the long term.

CPC should thus provide a stable, well documented base for future work in the field of clone tracking within the Eclipse IDE. While this places an emphasis on the typical aspects of good framework and API design [19, 27], the following considerations are of special interest.

low initial adoption cost: A developer reusing *CPC* should only need to know the parts of the API which directly affect his task and should be shielded from details which are

not relevant. The framework should furthermore detect and report incorrect API usage whenever possible.

high flexibility: Advanced users should be able to customise *CPC* to fit even uncommon requirements which were not envisaged during the initial development of *CPC* as far as possible.

multi-level reuse: The initial design will not be able to cover all potential future needs. Depending on the degree of customisation required, it should be possible to reuse *CPC* functionality on different levels of granularity. Starting with the contribution of a new strategy and ending with the replacement of entire subsystems and core services.

stable core: It should be possible to address even unusual needs without having to modify the *CPC* core module. Ideally a contributor would include the *CPC* code from the official *CPC Update Site*² and would provide all modifications within additional plugins. This would allow contributors to still benefit from *CPC* improvements and bug fixes, even when using a heavily modified version, and is also essential for the next point.

3rd party contributions: Multiple parties should be able to contribute extensions to a *CPC* installation without introducing the risk of conflicts and incompatibilities. Interaction between contributions of different parties should be possible via defined channels.

The resulting design and implementation aspects are covered in detail in section 3.2.

Copy and Paste Clone Tracking: The version of *CPC* developed in the course of this thesis focuses on clones created by copy and paste actions of the developer. Detection of such actions and continuous tracking of clone positions across document modifications is thus a key requirement.

Robustness: Any clone tracking system without instrumentation of the source code is highly vulnerable to modifications made outside of its supervision. A small modification in an external editor or any document modifying processing of a source file by other tools could potentially lead to a loss of clone data. Synchronisation problems could also be caused by a number of other events like an IDE or OS crash.

CPC should be able to detect *all* such modifications and should try to reconcile external changes on a best effort basis. If clone positions can not or only partly be reconciled, *CPC* should adopt a graceful fallback strategy and drop any potentially invalidated clone information.

Remote Synchronisation: *CPC* should support distributed development environments in which multiple programmers are potentially modifying the same source segment concurrently. Such modifications could potentially cause merge conflicts and network connectivity might only be available intermittently.

Data Collection: To facilitate further clone research, *CPC* should be able to collect and export a wide variety of cloning related data. Of special interest are copy and paste activities by

² *Update Sites* represent the main software distribution and updating concept of the Eclipse platform.

the programmer in general, the resulting clone instances as well as the complete modification history of each clone.

Initial Clone Import: To ‘jump start’ *CPC* on existing projects, it should be possible to import static clone detection results. In light of the large number of available static clone detectors and the ongoing research in this area, the emphasis should lie on the development of a versatile import API rather than the support of one specific detection tool.

All imported clones should be clearly marked, as they could represent false positives and might sometimes need to be treated differently from normal copy and paste clones.

Ease of Use: In order to achieve the desired adoption by a broad range of programmers on all kinds of projects, the installation and setup of *CPC* should be as simple as possible. Ideally the end user would just add the *CPC Update Site* to Eclipse and select the desired *CPC* modules.

Non-Intrusiveness / Safety: *CPC* should *not* affect the normal programming practice of its users. For common project sizes it should *not* affect the overall system performance in a notable way. It should *not* leave any visible artifacts within the source code files. And *no* conceivable type of *CPC* failure must be able to affect the integrity of the software application under construction.

2.3.3 Limitations

*Every big computing disaster has come from taking
too many ideas and putting them in one place.*

[Gordon Bell]

In order to reduce the scope of this thesis to a manageable subset some limiting assumptions needed to be made. This section lists the major limitations.

Copy and Paste Clones: This version of *CPC* only detects and tracks clones which were created by copy and paste actions of a developer. All other types of cloning are ignored. Additional, operating system specific copy and paste types like the middle mouse paste functionality on *Linux* are not supported. The Eclipse Platform does not provide sufficient information to track such events.

Homogeneous Environment: All team members use the latest Eclipse IDE and have the *CPC* plug-in installed. The *CPC* version across all workstation matches and there are only minor differences in configuration.

Java Development: The software under construction consists mainly of *Java* source files. Cloning in other types of source files is not taken into account. Support for other languages could easily be added as only very few areas of *CPC* need explicit language support (*see also chapter 4*).

CVS Repository: As none of the two major *SVN* team provider plug-ins for Eclipse implements all the required Eclipse team APIs the only supported team provider for the initial version

of *CPC* is the *CVS* team provider which ships with the Eclipse IDE. This problem is covered in-depth in section 3.4.9, 5.4 and 5.4.

Limited External Modification: External modifications to source files outside of the supervision of *CPC* are rare events and usually only affect a small subsection of the software application under development. Very high rates of external modifications may render *CPC* unusable due to the potentially high degree of clone data loss³.

Limited Team Conflicts: Merge conflicts due to concurrent modification of the same source file by multiple developers do only occur infrequently and only affect a small number of files. As with external modifications, excessive amounts of merge conflicts may reduce the benefits which can potentially be derived from adopting *CPC* considerably.

System Performance: The workstations have sufficiently recent hardware to comfortably run the Eclipse IDE and a number of 3rd party plugins. On old systems with large projects and/or a large number of clones the background activities required for clone tracking tasks could lead to performance degradations.

³External modifications to source files which do not contain clone data are of no concern.

Chapter 3

Design and Implementation

This chapter provides a short introduction into the Eclipse Platform followed by an overview over the generic design goals and approaches employed during the development of *CPC*. The chapter then provides detailed design and implementation information about the *CPC* core component and the *CPC* module components.

3.1 The Eclipse Platform

The Eclipse Platform has already served as a platform for many of the *ECG Lab* based works and was introduced in length in the corresponding papers and theses [12]. It will therefore only be covered very briefly here. Readers who are not familiar with the Eclipse Platform as a modular, highly extendible framework are encouraged to pause here and take the time to read the *Eclipse Platform Technical Overview*¹.

While the Eclipse Project is best known for its integrated *Java* development environment (*IDE*), the Eclipse SDK, it offers much more than the *Java* IDE. The Eclipse RCP core provides a foundation for the development of all kinds of platform independent rich client applications and the Eclipse Platform is a versatile base for building arbitrary IDEs.

The Eclipse Platform undoubtedly represents a hallmark of modular design. It serves as a central integration point for a multitude of contributions by other Eclipse Project teams and third parties. The Eclipse SDK is entirely built upon loosely coupled, reusable components. One of the main drivers which makes this possible is a key concept of the Eclipse Platform, the **plug-in**.

Plug-ins are selfcontained modules which can freely modify and extend the functionality and user interface of the Eclipse Platform. The plug-in framework builds up an in-memory registry of all installed plugins at startup, resolves dependencies and provides plug-ins with extensive context information. The life cycle of all plug-ins is managed by the framework which automatically defers the loading of a plug-in until its functionality is actually being requested by the user or an already activated plug-in.

Another key concept of the Eclipse Platform is the **extension point**. Each plug-in can define its own extension points as well as extensions to extension points of other plugins in its plug-in

¹This article is included in the *Appendix and Recommended Readings* booklet.

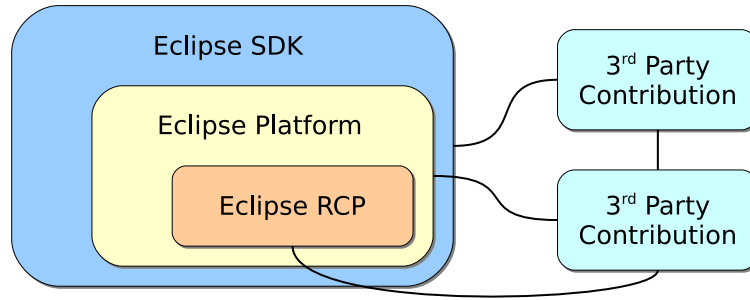


Figure 3.1: Basic layout of the Eclipse environment (*simplified*)

manifest. An extension point can be used to convey information (*i.e. specification of a file type which should be hidden*) or to provide implementations of specific API interfaces (*i.e. a callback method to be executed when a specific menu item is selected*).

Every plug-in has its own Java class loader which checks and enforces the dependencies and visibility rules specified in the plug-in manifest. Eclipse defines three additional visibility types for packages within plug-ins: **public**, **internal** and **private**. Public packages may be accessed by all plug-ins, once they’ve declared the corresponding dependency in their plug-in manifest. Internal packages should only be access by so-called ‘friends’, other plug-ins which were specifically listed as having access to the package. Private packages are only available for plug-in internal use. While access restrictions are not enforced for internal packages, any access to a private package will result in a runtime exception.

While the Eclipse API specification clearly states that only official API classes and interfaces may be accessed by a 3rd party plug-in [11], none of the internal Eclipse packages is marked as private. This enables plug-ins to fall back to internal non-API segments if this is really required. The current *CPC* implementation tries very hard to limit itself to the official APIs. However, while no part of *CPC* accesses non-API packages, there are a number of cases where additional implementation details, which are not part of the specification, need to be taken into account in order to support some of the *CPC* core requirements. These cases are covered in section 5.3 and 5.4.

Other interesting aspects are an adapter framework which allows plug-ins to dynamically extend existing objects at runtime to add new functionality, the Eclipse Platform’s update manager which can be used to install and update plug-ins (*semi-*)automatically and many more².

3.2 Generic Design Goals and Approaches

*Simple things should be simple,
complex things should be possible.*

[Alan Kay]

The requirements chapter already provided a first glimpse at some of the considerations for the

²The reader is encouraged to refer to the *Eclipse Platform Technical Overview* as well as the available Eclipse online resources [12, 5, 6].

CPC framework design in section 2.3.2. This section will cover some of these points in more detail and highlights the implications for *CPC* and some of the trade-offs required.

Good software design is hard even under the best of circumstances; time pressure, technical uncertainties and emergent requirements make the task all the more daunting. Should *CPC* be adopted by users and developers, *it will fail*. It is important to realise that no matter to what extent one goes, one will *never* be able to conceive *all* the potential future uses and requirements. So while one may try to anticipate as many future needs as possible, one should prepare, from the outset, for the worst.

As Parnas put it years ago, software will ‘age’ and designing for change thus needs to be a central goal during software development [47]. Under the ‘environmental’ conditions which await *CPC*, ‘aging’ will be fast, very fast.

Complexity due to Flexibility

Fortunately the Eclipse Platform, itself a prime example of flexible design, provides the ideal base for such an endeavour. Its plug-in concept provides a ready to use base framework which can be leveraged to provide a very flexible yet concise solution. Advanced features like managed lifecycles and lazy loading of components, dependency management, online installation and updating and many more can be reused with minimal effort.

However, the flexibility comes at a price. The inherent complexity introduced as a side effect is considerable (*see section 5.3.1*). Flexibility versus complexity is just one prominent example of required trade-offs between different conflicting design goals. There existed a constant need for such trade-offs during the design and implementation phases.

One such example is lazy loading. While the Eclipse Platform takes care of the entire lifecycle management, a plug-in needs to take special care when accessing other plug-ins and handling data. In order to receive all the benefits of lazy loading, the introduction of an additional layer in the application design is required. All code sections which do not absolutely need a foreign object’s functionality should make use of descriptor or proxy objects which defer the creation of or access to the object until one of its methods is actually needed³.

Another example is the trade off between including functionality within an existing plug-in and encapsulating it within its own plug-in. While a strict segregation of different features into different plug-ins improves flexibility and reusability it also adds complexity. And while the extra overhead for a loaded plug-in in Eclipse is small, having a number of plug-ins which contain only one or two classes is certainly too extreme. Due to its strong focus on flexibility and future reuse potentials, *CPC* was split into a large number of relatively small plug-ins. The main rationale behind this decision was to provide 3rd parties with a way to highly customise certain parts of *CPC* while still retaining the default versions of all other components of *CPC*. This way contributors are still able to benefit from patches and updates made to the remaining, official components.

³The Eclipse Platform will automatically load a plug-in once any of its classes is accessed.

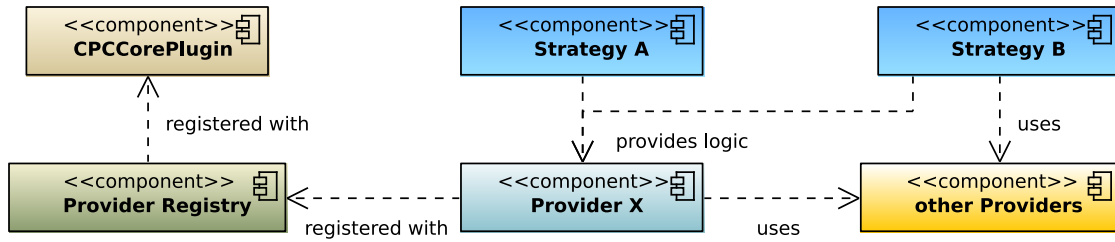


Figure 3.2: COD: Example of design for multi-level reuse in CPC

Visibility Rules

The distribution of functionality over a large number of plug-ins, together with the aim to allow contributors to replace any part of *CPC* results in another problem. *Java* package visibility rules become basically useless. The *Eclipse API rules of engagement* specify that a plug-in should not declare code in a package belonging to another component [11]. In effect this means that all classes and methods which might potentially be of interest to another plug-in need to be declared as public. The new package-level visibility rules introduced by Eclipse can help to offset this, but only on a per package level. There is no way of restricting access to specific methods of a class.

To address this issue, *CPC* adopted a powerful but sometimes complex interface and casting based solution, which is also employed by the Eclipse Platform. Instead of including a large number of methods which must not be called by most clients and documenting this fact in their specification, such methods are distributed over multiple interfaces. The base interface only contains the common methods and a number of specialised sub-interfaces ‘hide’ all additional methods.

The available sub-interfaces and the corresponding access restrictions are documented as part of the API specification. An interested client has to explicitly cast the base interface into one of the specific sub-interfaces in order to access any of the restricted functionality. This approach is evident in most parts of the *CPC* framework. One particular elaborate version can be found in the discussion about the *CPC* clone data objects in section 3.3.3.

This approach also supports the goal of low initial adoption cost for new developers. By default they will only see common functionality which reduces the initial conceptual complexity of the system and spares them from many potential errors which could result from inadvertently using methods which are not meant for public use.

Multi-level Reuse

To achieve the maximal amount of design flexibility and thus increase future reuse potentials even in scenarios which were not considered during the development of *CPC*, a key goal is to enable contributors to reuse *CPC* functionality at different levels of granularity.

A contributor using *CPC* within its intended application domain might just want to extend or modify some heuristic or add a new view to the user interface. However, once the requirements are very different from the original goals of *CPC* a reuse on this level will not be possible. A contributor might need to replace the entire heuristics subsystem or implement a radically different approach to clone creation, tracking or persistence. Even some of *CPC*’s core functions, as described in the

next section, might no longer be suitable.

CPC tries to accommodate such needs by enabling a 3rd party to add, modify or remove functionality at all levels. The typical structure of most *CPC* modules can be seen in figure 3.2. **Provider X** supplies some kind of service to the system. The service is either specified in one of the *CPC Core APIs* or represents a new type of service contributed by a 3rd party⁴. A service provider implementation typically defines a set of APIs which can be used to customise its behaviour and one or more extension points to register implementations of those APIs with the service provider.

A contributor could now add, modify or remove strategies registered with the provider. Some service providers support multiple strategies which are then executed in a specified order. In such a case a contributor could insert new strategies in a specific place and could decide at runtime whether any of the following strategies should still be executed. The behaviour of **Provider X** could also be changed by modifying some of the other providers on which it, or one of its strategies, depend for some of the internal processing.

If this does not suffice to address the concrete requirements at hand, a more drastic approach would be to replace the provider with a new implementation. The implementation could still make use of **Provider X**'s registered strategies, if that seems beneficial. The new implementation could also define a completely different API and provide a new set of strategies.

In the extreme case in which the limitation lies in one of the two central building blocks of *CPC*, the service provider registry or the event hub registry (*see section 3.3*), even their implementations could be replaced.

General Design

In general, common design patterns were used wherever their adoption was deemed to be beneficial to the overall design or the understandability of specific areas [27]. Furthermore, the adoption of certain design patterns in some places is enforced or at least strongly encouraged by some of the Eclipse Platform APIs.

Furthermore, Joshua Bloch's principles of good API design as well as other sources were considered, where applicable [19]. A number of the points highlighted by Bloch and others have already been addressed above, some others are obvious. Of more interest are those points where *CPC* deliberately or perforce deviates from these recommendations.

CPC's quest for an API which enables it to provide useful services for a large, very heterogeneous and so far largely unknown number of future extensions and contributions can be seen as a case of deliberate over engineering. Its APIs are more numerous and detailed than would be strictly necessary for basic copy and paste clone tracking support.

About this Chapter

The following sections contain a number of diagrams and graphics. It is important to note that these are meant as examples which illustrate certain aspects of the *CPC* design and implementation. They do *not* represent authoritative specifications of the underlying systems. In many cases they

⁴It is important to note that multiple external parties may be involved.

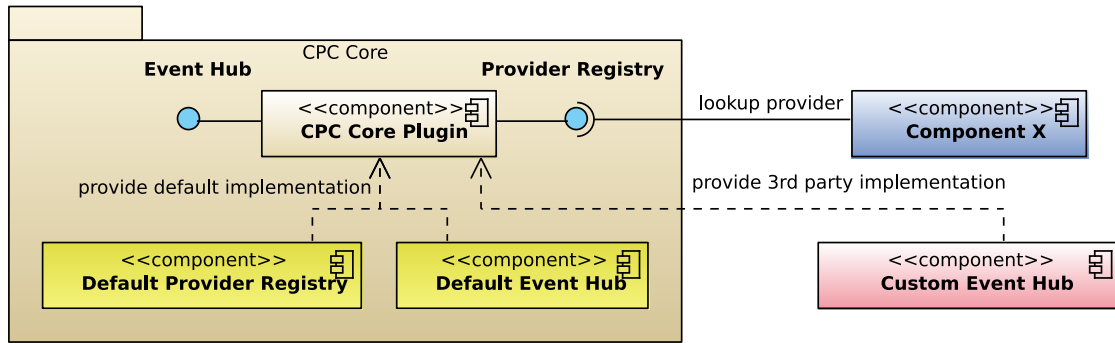


Figure 3.3: COD: The CPC Core module

only display a limited, simplified view. While most diagrams follow the *UML* standard, they may deviate in some aspects. The title of each *UML* diagram starts with a three letter code which specifies the type of diagram. The codes are explained on page 8.

Furthermore, this chapter often uses abbreviated names for classes and components for the sake of readability and brevity. At the end of each section the *CPC Core API Specification* paragraph lists names of relevant interfaces and classes which may be of interest to the reader. More information about these can be found in the provided *CPC Core API Specification* booklet. The *Service Provider Dependencies* paragraph of a section lists service provider APIs which are required by the corresponding module. The *Provider Registry* and *Event Hub* are required by almost all modules and are therefore not explicitly listed.

3.3 The CPC Core

CPC was designed in a highly modular fashion. A large number of independent components and plug-ins surround one central integration point, the *CPC Core* module. The low coupling between the different components is made possible by two of *CPC*'s core concepts, the (*service*) *Provider Registry* and the *Event Hub Registry* as well as a common set of *Clone Data Objects*.

Figure 3.3 displays the basic structure of the *CPC Core* module. It specifies a large number of API interfaces⁵ as well as default implementations of the *Provider Registry* and the *Event Hub Registry*. Two special extension points allow contributors to replace the default implementations with their own custom realisations which are then automatically available to all other components.

The remainder of this section describes the *Provider Registry*, the *Event Hub Registry* and the *Clone Data Objects* in more detail.

CPC Core API Specification:
CPCCorePlugin

⁵The provider and event hub registry are just some of the specified interfaces, please refer to the *CPC Core API Specification* booklet for a complete list.

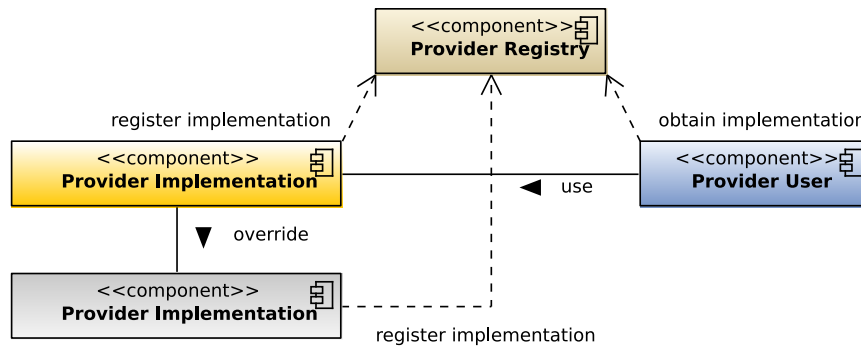


Figure 3.4: COD: The CPC service provider concept

3.3.1 Service Provider API

Within the *CPC* framework many components provide services and in turn require the services provided by other components. The *CPC Core* module defines a large number of service provider API interfaces (see section 3.4) which can be implemented by arbitrary plug-ins. The *Provider Registry* provides a central exchange point between implementers and users of these service provider APIs.

The *OSGi* component model⁶ adopted by the Eclipse Platform contains a built-in service provider concept which could support some of the requirements for such an exchange point. Aside from the additional complexity the *OSGi* approach would entail, there are a couple of issues which made a custom implementation more beneficial. The main rationale for this approach was the fact that the Eclipse *OSGi* implementation would have negatively affected the flexibility of a core *CPC* component and that the effort required for a custom implementation was relatively low. Even the Eclipse Platform itself makes only limited use of the *OSGi* service provider concept.

The basic concepts behind the resulting *Provider Registry* can be seen in figure 3.4. The *Provider Registry*, which is itself registered with the *CPC Core* module, allows arbitrary components to register implementations for a specific service provider API interface. When registering a provider, a plug-in can specify the priority of the provided implementation as well as some life cycle details.

For a given service provider API interface, the *Provider Registry* enables plug-ins to retrieve an instance of the implementation with the highest priority (*the usual case*) or descriptor objects for all registered implementations. This approach enables 3rd party contributions to freely override any existing provider implementation with a custom replacement.

Further life cycle data can be used by a provider contributor to indicate how requests for an instance of the provider should be handled. Most service providers are declared as singletons, all clients requesting an instance will receive a reference to the same provider object. While this is the most economical approach, there may be provider implementations which require a different treatment. Other options are to create a new provider instance for each request or to create multiple instances as part of an instance pool. Special callback methods are used to notify provider implementations about potentially interesting life cycle changes.

Provider implementations can be either registered via specific *CPC Core* extension points or

⁶ *OSGi* – The Dynamic Module System for Java — <http://www.osgi.org>

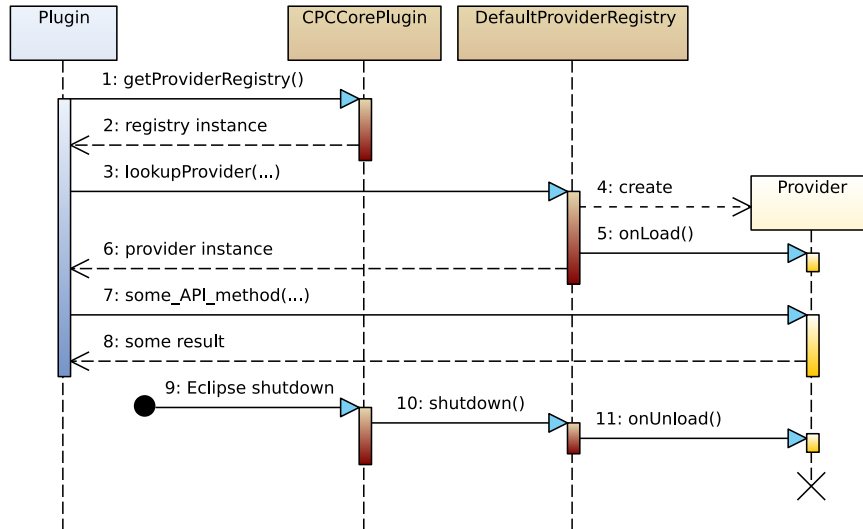


Figure 3.5: SED: Provider registry usage for initial lookup of a singleton provider

via special subscription methods which are part of the *Provider Registry API*. A *Provider Registry* implementation will internally keep track of all registered provider implementations and might also engage in other activities like caching, pre-loading or pooling of provider instances.

Figure 3.5 displays the typical interaction between plug-in code which requires the services specified in a certain service provider API interface and the *CPC* framework. First the plug-in obtains a reference to the *Provider Registry* from the *CPC Core* module⁷. Then an instance of a provider implementing the specified interface is requested from the *Provider Registry*. For a non-singleton or the initial request of a singleton provider, the *Provider Registry* will create a new instance of the provider implementation with the highest priority and will notify the provider about the fact that it will be handed to a client. The client can then interact with the provider as specified by the corresponding API. A singleton provider will also be notified about an imminent shutdown of the Eclipse IDE.

CPC Core API Specification:

`IProviderRegistry`^{*}, `IProvider`^{*}

3.3.2 Event Hub API

The *Event Hub* represents another key concept of the *CPC* framework. It promotes loose coupling between components by providing a centralised, defined way of exchanging arbitrary data between interested components.

Figure 3.6 provides a simplified overview. A component which intends to process events of a specific type or events of a specific category can register an event listener with the *Event Hub*. Listeners are registered either via a corresponding extension point of the *CPC Core* module or via special methods of the *Event Hub API*. When registering a listener, a component can specify its priority and whether it wishes to receive event notifications synchronously or asynchronously.

⁷This might yield a 3rd party *Provider Registry* implementation.

⁸Indicates the existence of further relevant sub-interfaces or sub-classes.

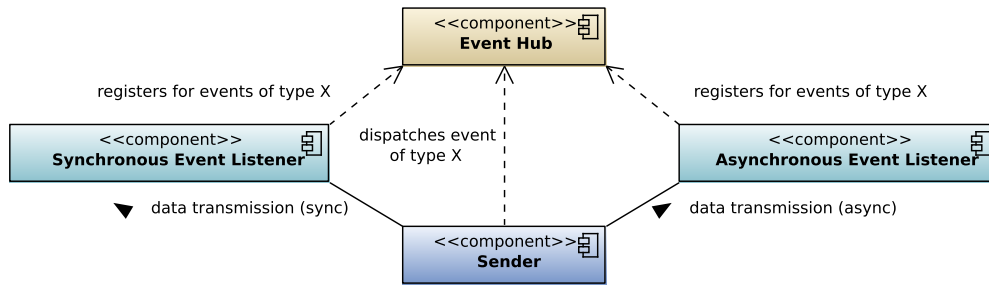


Figure 3.6: COD: The CPC event hub concept

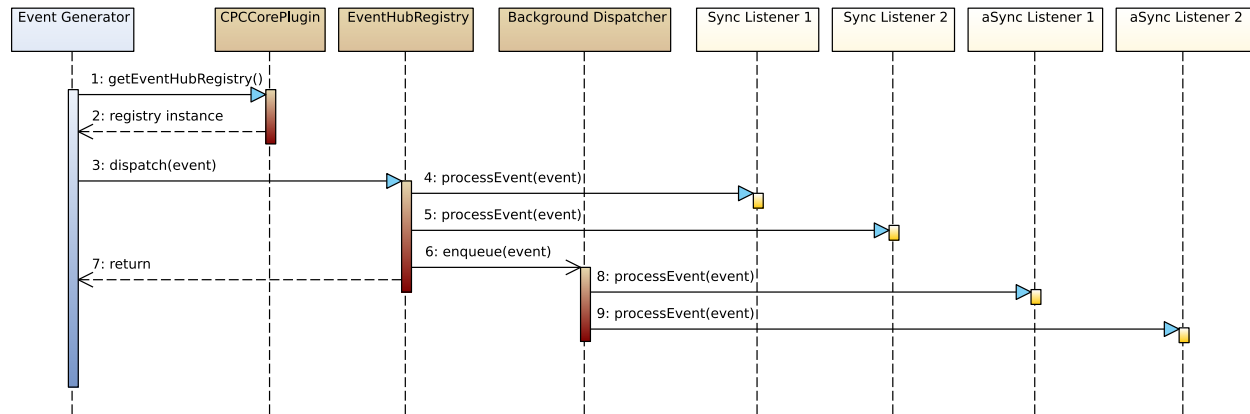


Figure 3.7: SED: Event hub registry usage with synchronous and asynchronous listeners

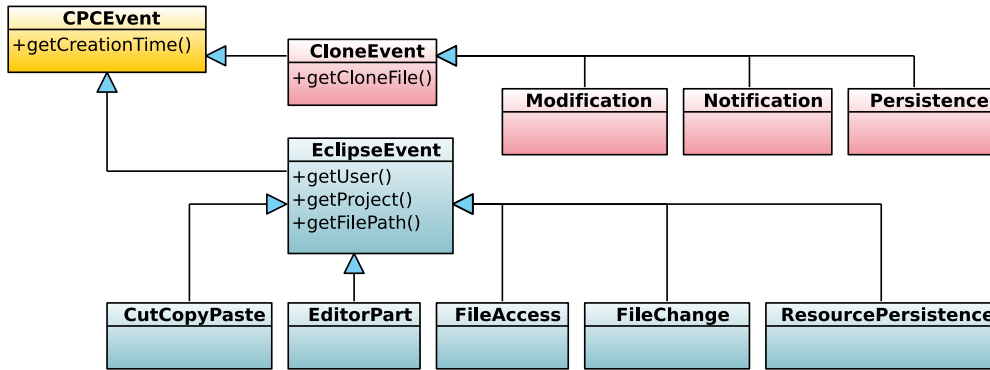
That is, whether the event sender should be blocked for the duration of the event processing by the listener or whether the sender may continue its execution concurrently. Once an event of the specified type or category is created by any component and sent to the *Event Hub*, it will be dispatched to all registered listeners in the order of their priorities⁹.

Figure 3.7 displays the interactions between an event generating plug-in and the *CPC* framework. As with the *Provider Registry*, initially a reference to the current *Event Hub* implementation is retrieved from the *CPC Core* module. This way a different *Event Hub* implementation can easily be plugged in. However, the interesting aspect here is the handling of synchronous and asynchronous listeners.

The *Event Hub* will block the sender of an event until all synchronous listeners have finished processing the event. This allows listeners to partly affect the future execution of the event sender by modifying central clone data elements or sending further events. It also ensures that no further actions of the sender can lead to an illegal state or invalidate any globally accessible data which might be of interest to a listener. One example of such a case is the opening of a source file in an editor. In this case the *CPC Track* module needs to load the clone position data for the new file before the user can modify the file in the editor. By using a synchronous listener, the *CPC Track* module can be sure that the editor window will only open once its listener loaded all the required data.

While the default *CPC* implementation uses a number of synchronous listeners, the majority

⁹The dispatching order for multiple listeners with equal priorities is not specified.

Figure 3.8: CLD: The CPC Event hierarchy (*simplified*)

are asynchronous listeners. Asynchronous listeners have the benefit of not blocking the executing thread. As a large number of events typically originate from the main user interface (*UI*) thread, use of a synchronous listener always entails the risk of making the UI ‘lag’ due to long running processing within the listener.

Asynchronous listeners are notified about an event once all synchronous listeners have finished their processing. It is up to the *Event Hub* implementation whether events are dispatched to asynchronous listeners from one or multiple background threads. A listener can indicate whether it is thread safe or not. Non-thread safe listeners are guaranteed to never receive notifications concurrently from multiple background dispatching threads.

Event Types

The *CPC Core* module defines a number of event types. Figure 3.8 provides a simplified overview of the available events.

Eclipse related Events: Events of this category are directly caused by activities within the Eclipse IDE and are generated primarily by the *CPC Sensor* module (*see section 3.4.1*). Such events are mainly of interest to ‘primary’ event consumers¹⁰ which in turn might generate further ‘Clone Data related Events’ during the processing of these events. All events of this type contain information about the current user, the name of the project and the path of the affected file.

CutCopyPaste: This event is generated whenever the developer executes a cut, copy or paste operation. It provides information about the clipboard content, the selected text and the current document content.

EditorPart: This event is generated whenever an editor window is opened or closed or when an editor window gains or loses the input focus. It does not contain any payload aside of the event type.

FileAccess: This event is generated whenever a file is opened or closed. It lists the affected file, the type of the event and contains a reference to the document object for the file.

¹⁰As opposed to consumers which are only interested in *CPC* related changes like clone additions and modifications.

Events are generated for files opened and closed in an editor as well as for files which are accessed by background tasks.

FileChange: This event is generated whenever a file is moved, renamed or deleted. It contains the type of the event and optionally the new project name and file path.

ResourcePersistence: This event is generated whenever a file is saved or reverted. It contains information about the affected file, the type of operation and whether the file was modified by some background activity or whether it was modified inside a visible editor.

Clone Data related Events: Events of this category are generated by *CPC* components in response to other events or actions which affected the clone data.

Modification: This event is generated by the *Store Provider* (see section 3.4.3) at the end of a transaction which modified the persisted clone data. It contains a detailed listing of the modifications made and provides updated, shared clone objects which can be used by listeners which want to keep a local cache of clone data up to date. This event is one of the key events in the *CPC* framework. Most contributors will be interested in it.

Notification: This event can be generated by any component that detected a potential clone related update anomaly which the user should be notified about. Usually such events are generated by the *CPC Notification* module (see section 3.4.7). The *CPC Notification UI* module is a typical consumer of such events.

Persistence: This event is generated by the *Store Provider* whenever the cached and potentially dirty clone data for a file is written to disk. This typically happens when the user saves a modified file in an editor. Events of this type are of interest to components which need to keep track of the state of the persisted clone data for some files.

The set of events is not limited to these predefined types. A 3rd party can freely add its own event types. Non-standard events can obviously only be used to transmit data between components which are aware of the new event type. However, it is also possible to subclass one of the existing event types to add additional payload to an event. In that case components which are aware of the new sub-type can process all of the payload while other components will still be able to handle the event, but will only see the default payload set.

CPC Core API Specification:

IEventHubRegistry*, IEventHubListener, CPCEvent*

3.3.3 Clone Data Objects

At first glance the design of a data object to represent a clone seems trivial. One approach which comes to mind is a simple *POJO*¹¹ or, similar to *Markers* in Eclipse (see section 5.3.2), a *HashMap*. However, if one carefully examines the requirements for *CPC*, a number of issues surface which can not all be solved with such simple approaches.

¹¹Plain Old Java Object — <http://www.martinfowler.com/bliki/POJO.html>

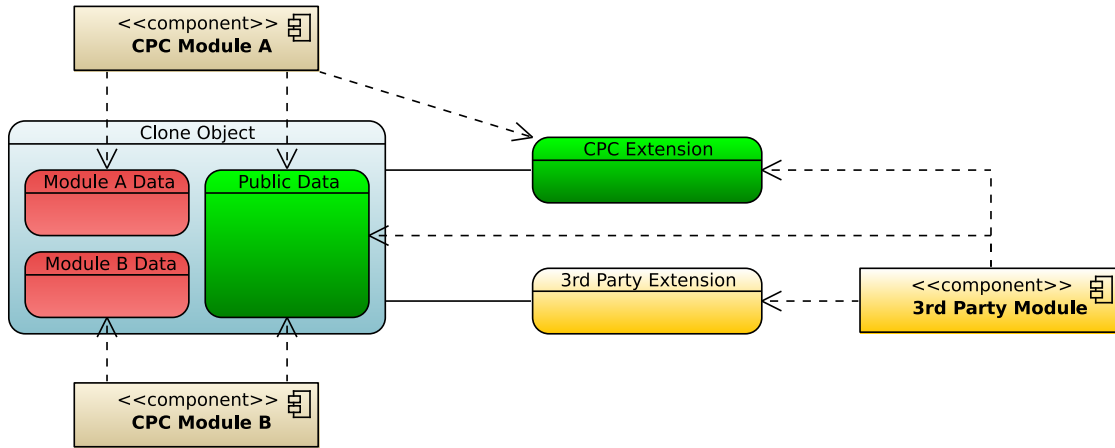


Figure 3.9: The CPC Clone Data Object

Extendible: A clone data object needs to be extendible. There are a large number of applications for *CPC* which are likely to require additional data to be stored together with the clone object.

Multiple Contributors: It is not enough to allow one contributor to supply a new clone data object implementation. There might be multiple parties who need to store additional, custom data within the *same* installation of *CPC*.¹²

Complex Data: The additional data may not always be in a simple, flat form. Some contributors might want to store large element lists. Some of these lists might be large enough to require a lazy loading approach.

Private Data: It should be possible to clearly mark certain data elements as being ‘private’ and only meant for access by a specific component.

Persistence Independence: Extending the clone data object with additional data should not affect the persistence services of *CPC*. The *Store Provider* should not need to be modified in order to persist the new data.

Fallback Option: If the clone data object design really can not accommodate a specific requirement. Entirely replacing the object with a custom implementation should be possible.

The design approach chosen for *CPC* is reflected in figure 3.9. *CPC* defines a *POJO*-like clone data object which contains public and private data elements and which can be extended by additional extension objects. Access restrictions for the default data elements are realised via additional sub-interfaces (see section 3.2). The main interface for a clone data object, *IClone*, only provides getters and setters for the publicly available data elements and only getters for read-only elements. Elements which are only meant for use by specific components can only be accessed by casting the clone data object to the corresponding sub-interface, i.e. *ICreatorClone*.

¹²I.e. if the user installs multiple 3rd party extensions from different sources.

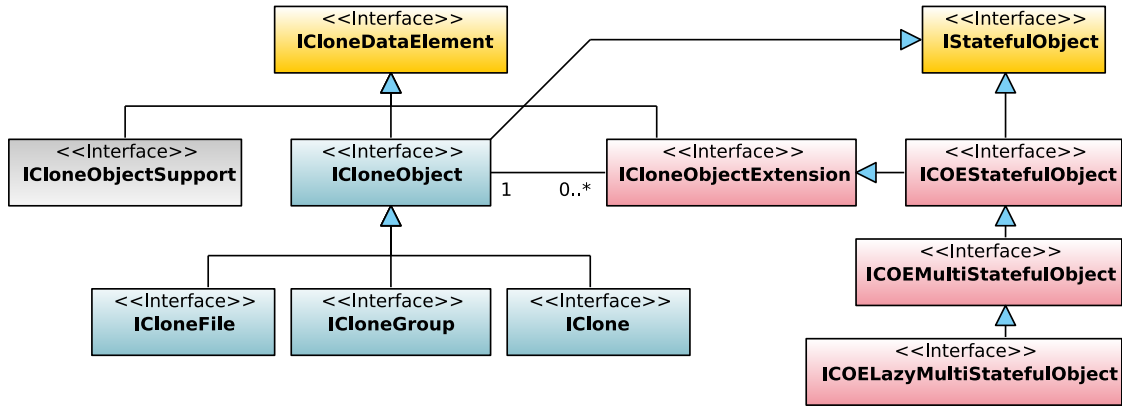
Figure 3.10: CLD: The CPC Clone Data hierarchy (*simplified*)

Figure 3.10 shows the adopted approach in more detail. Besides the core clone data object `IClone` itself, there are additional objects to represent files and clone groups. These `ICloneObjects` can be extended by an arbitrary number of `ICloneObjectExtensions`. While all `ICloneObjects` need persistence, it is up to the implementer of an extension to specify whether it should be persisted together with the corresponding `ICloneObject` or whether it is transient in nature and is only meant to exist for the current session. An extension object can also implement special interfaces to support persistence of non-flat data structures and lazy loading. Persistence independence is achieved by use of the `IStatefulObject` interface. It prevents a persistence provider from requiring any knowledge about the types and internal structure of clone data objects. Some of the design goals for the *Store Provider* also affected the clone data object design (*see section 3.4.3*).

All instances of clone data objects are created by a central *Clone Factory Provider* which can be obtained via the *Provider Registry*. All parts of *CPC* which need to create new clone data object instances exclusively do so via the *Clone Factory Provider*. There is *no* place in the *CPC* framework where the implementation of clone data objects is directly referenced. The *Clone Factory Provider* itself obtains information about the available implementations from the corresponding *CPC Core* module extension point and each registered clone data object has an assigned priority. This allows 3rd parties to contribute custom clone data object implementations and to replace or modify existing data objects, if required. The default *Clone Factory Provider* itself can also be replaced with a custom implementation, if needed.

Clone data objects are also integrated into the Eclipse Platform adapter framework. This enables a tight integration of *CPC* data objects with the Eclipse IDE. One example of such integration is the built-in Eclipse properties view. If a *CPC* clone data object is selected in a *CPC* UI view, the properties view will automatically display the relevant properties for the selected object.

CPC Core API Specification:

`ICloneDataElement*`, `IStatefulObject*`, `ICloneFactoryProvider`

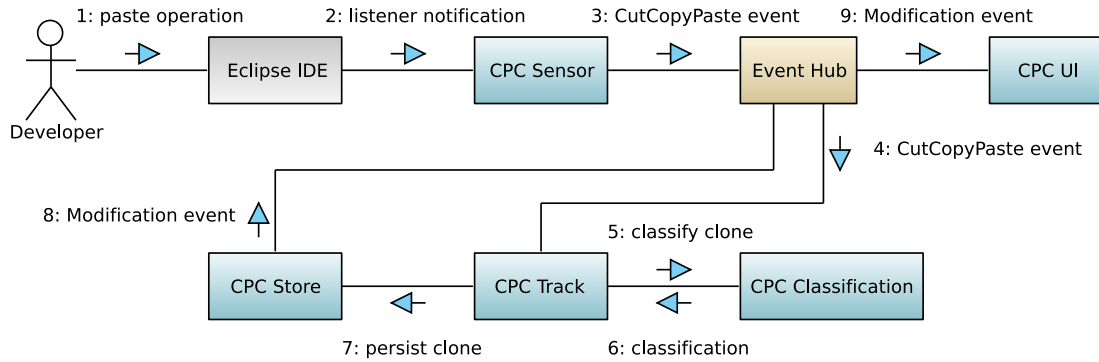


Figure 3.11: CUD: Module collaboration during the handling of a paste operation (*simplified*)

3.4 The CPC Modules

While the *CPC Core* module provides the essential backbone for the entire *CPC framework*, it does not include any functionality in itself. It deliberately limits itself to providing basic communication services and API specifications. A number of loosely coupled components, the *CPC modules*, provide specific functionality which, when orchestrated correctly, yields the *CPC Eclipse plug-in* as it is visible to the end user.

Figure 3.11 displays a simplified example of the collaboration between *CPC* modules. In this case a developer's paste operation triggers multiple events and is processed by a number of *CPC* components before the new clone entry is finally visible in the user interface. It is important to note that the only coupling in this scenario is the dependency of the *CPC Track* module on the *Classification Provider* and *Store Provider* API interfaces. There is no coupling between the *CPC Track*, *CPC Classification* and *CPC Store* modules.

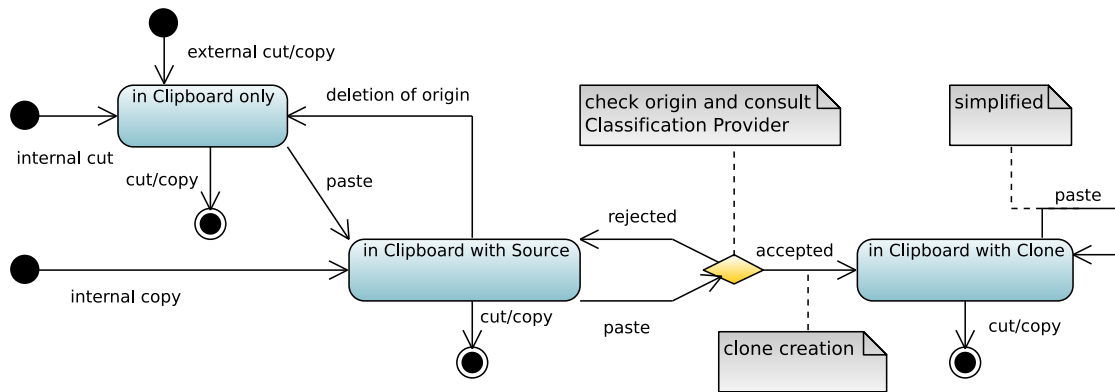
3.4.1 CPC Sensor - Eclipse Event Hooks

CPC is highly dependent on receiving information about actions within the Eclipse IDE. Fortunately, the loosely coupled, modular nature of the Eclipse Platform makes it possible to register listeners, callbacks and other means of observing programmer actions. In order to prevent such, sometimes complex interactions with the Eclipse Platform from spreading through all *CPC* modules, which require notifications about some type of event, they are encapsulated in one central module, the *CPC Sensor*.

The internal workings of the *CPC Sensor* are similar to the *ECG Eclipse Sensor* of the *ECG Lab*. However, it has been considerably trimmed down and extended to other event types which are of interest to *CPC*. Initial versions of *CPC* also made direct use of the *ECG Eclipse Sensor* (see section 5.2).

While registration for specific events within the Eclipse Platform is straight forward for commonly needed events, it can become an extremely time consuming topic for other event types. Section 3.4.9 and 5.3.2 describe some of the problematic cases.

Many of the events covered in the *Event Hub API* section (3.3.2) are generated by this module.

Figure 3.12: SMD: Internal states during cut, copy and paste events (*simplified*)

CPC Core API Specification:

`EclipseEvent*`, `IEventHubRegistry`

3.4.2 CPC Track - Clone Tracking

The *CPC Track* module is where most of the actual copy and paste clone tracking activity takes place. This module listens for copy and paste events from the *CPC Sensor* and potentially creates new clone entries on paste actions of the programmer. The module is furthermore responsible for updating the positions of all known clones during document modifications.

The complexity of this module largely stems from the inadequate Eclipse support for position tracking within documents. Instead of using the built-in, high-level position tracking concept of *Markers*, the *CPC Track* module needs to fall back to low-level *Position* elements. This is discussed in more detail in section 5.3.2.

The main events of interest for the *CPC Track* module are the file-access, cut-copy-paste and resource-persistence events. All actions of the module are triggered by either such events or document modifications by the developer.

Clone Creation

New clone instances are created as a result of cut-copy-paste events. Figure 3.12 displays a simplified overview of the potential internal states.

The first important distinctions are those between cut and copy operations and those between internal and external copy operations¹³. In case of a cut operation or an external copy operation the clipboard content has no corresponding source location within the software system. A paste does thus not create a clone according to *CPC*'s clone definition. Even if the source was originally part of the software system, it might be deleted while its copy resides in the clipboard, resulting in the same situation.

In the event of a paste operation, if the clipboard content has a source, the similarity between the origin and the clipboard content is reevaluated with help of a *Similarity Provider*. In cases where the origin has been modified and the similarity has fallen significantly, the pasted code will

¹³Actions inside or outside of the Eclipse IDE and therefore under or not under *CPC* supervision.

not be considered as a clone. The potential new clone is then passed to a *Classification Provider* for classification. The clone is either accepted and classified or rejected by the provider. If the clone was accepted, it is sent to the *Store Provider* for storage. The *Store Provider* will then automatically notify interested parties about the new clone instance.

If the clipboard content is pasted multiple times, each paste operation creates a new clone instance and all instances share the same clone group. If the origin of the copy operation is already a clone, the new clone instance will become a member of its clone group. Matching of copied source selections and existing clone entries is done via a *Fuzzy Position to Clone Matching Provider*. For a given start and end offset this provider checks whether any existing clone closely fits the specified range. It is up to the provider implementation to decide how ‘loose’ the match may be.

Another special case is the automated source reformatting functionality of the Eclipse IDE. Depending on the workspace configuration, pasted code might be reformatted on insertion, resulting in a situation where the clipboard content during a paste operation does not match the content actually inserted into the document.

A simplified version of the module interaction during a paste event is also shown in figure 3.11.

Position Tracking

The key responsibility of the *CPC Track* module is the continuous tracking of clone positions across document modifications. This is achieved by listening (*synchronously*) for file-access events for newly opened or closed files. When a file is opened the *CPC Track* module retrieves its current clone data from the *Store Provider* and adds it as *Positions*¹⁴ to the internal Eclipse document. The *Positions* are extracted and stored again once the file is closed or after a specific timeout was exceeded. Some of the module interactions during this phase are also shown in figure 3.15.

The *Position* objects, which represent all clones within the file in question, are updated during document modifications by means of a custom *CPC Position Updater*¹⁵ and a document modification listener which are added to each document when it is opened or created. The decision about how a given modification should affect a clone is delegated to a *Position Update Strategy Provider*.

During the clone tracking process, all modifications made to the contents of clone instances are extracted and logged within a *Modification History Extension* of the corresponding clone object. This enables *CPC* to provide a complete history of the long term development of all clone instances within a software system.

Performance

The performance of the clone tracking process is of crucial importance. Early, *ECG Eclipse Sensor* based, versions of *CPC* failed to cope with the large number of document modification events which can be generated during the use of special Eclipse features like in-place rename refactoring. This is discussed in more detail in section 5.2.

The current implementation makes heavy use of caching opportunities. Modified clone positions are not written back to the clone data objects until some other component tries to access the clone

¹⁴In Eclipse *Positions* are small objects, describing a segment of the document to which they are attached.

¹⁵A *Position Updater* is a strategy which can be registered with a document to update *Positions* of a given type.

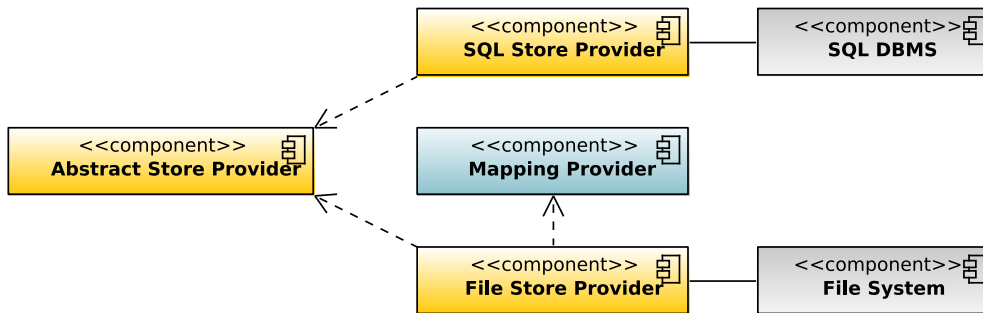


Figure 3.13: COD: The Store Provider concept

data. When this happens the other component is suspended and all cached changes are transmitted back to the *Store Provider* (see section 3.4.3). If no other component actively requests the clone data, the cache is persisted automatically after a certain period of inactivity.

Additional Tasks

Aside of these main responsibilities, the *CPC Track* module also listens for file rename and move operations in order to keep the mapping between file path and clone data up to date. Furthermore, it supervises resource persistence operations like save and revert in order to synchronise source file persistence and clone data persistence events.

CPC Core API Specification:

`EclipseEvent*`, `ICloneModificationHistoryExtension`

Service Provider Dependencies:

`IStoreProvider`, `ICloneFactoryProvider`, `ISimilarityProvider`, `IClassificationProvider`,
`IFuzzyPositionToCloneMatchingProvider`, `IPositionUpdateStrategyProvider`

3.4.3 CPC Store - Data Persistence

The *Store Provider* is a key component of the *CPC* framework. It represents the central persistence provider for all clone data objects. Its API enables other components to retrieve clone data for specific files, file segments or groups and also provides other key services. The clone modification events generated by the *Store Provider*, whenever a component modifies the persisted clone data, are some of the most important events in the *CPC* framework (also shown in figure 3.11).

One of the key design goals for the *Store Provider* API was to achieve the greatest possible independence from the underlying persistence implementation. The API should not limit the set of possible persistence approaches of future *Store Provider* implementations by us or 3rd parties. However, the highly flexible clone data object model chosen for *CPC* (see section 3.3.3) inevitably requires a certain degree of sophistication on the part of the *Store Provider*.

In order to allow a maximum implementation flexibility a number of API simplifications were made. Some of these considerations also affected the clone data object design.

Non-Object Oriented: Some of the potential future persistence providers might be inherently non-object oriented. An API which puts too much emphasis on object oriented approaches might thus prove to be a limitation for the implementation of persistence services based on such providers.

Concurrency and Transactions: A potential persistence provider might not allow concurrent access to stored data or it might not support transactions or guarantee transactional properties such as *ACID* ¹⁶. An example would be a simple file based persistence implementation.

Performance: Accessing data stored with some persistence providers might be time consuming. The API should thus provide enough information to allow effective caching approaches.

Simple: The effort required to implement a *Store Provider* wrapper for a new persistence implementation should be relatively low.

A number of steps were taken to address these main issues. Some of these steps address more than one of the points mentioned above.

Passive Data Objects: To reduce the complexity of *Store Provider* implementations, clone data objects are passive, shallow and detached from their persisted instances. Abstaining from lazy loading object graphs which can be traversed at will, reduces the complexity considerably. This way a *Store Provider* is freed from the obligation of providing or supporting special proxy data objects which would need to be kept in sync with the main clone data objects.

On the other hand this may somewhat reduce the usability of clone data objects as an additional, explicit round trip to the *Store Provider* is required, whenever extra data is needed. I.e. a clone file object contains no direct references to the contained clone objects and a clone object has no direct reference to the corresponding clone file object.

The `IStatefulObject` interface, which is implemented by all clone data objects, enables a *Store Provider* to persist and restore objects without any knowledge of their internal structure.

Locking: To address the concurrency and transaction problem, the *Store Provider* API adopts a read/write locking approach. The greatly reduced internal complexity, which is gained by employing exclusive write locks, clearly outweighs the potential loss in concurrency. Especially if the specific situation within the Eclipse Platform is taken into account. Most activities, which are of interest to *CPC*, originate from a single thread, the main UI thread.

An exclusive write lock ‘session’ represents *CPC*’s concept of a unit of work. The *Store Provider* will enqueue all clone data modifications until the end of the session, before the corresponding clone modification events are generated. However, the *Store Provider* API makes no guarantees about transactional properties.

To improve overall performance, one component can assume a special role in relation to the *Store Provider*. It is granted an implicit write lock and may freely modify its cached clone data. Once another component tries to acquire an exclusive write lock, the implicit write lock

¹⁶ Atomicity, Consistency, Isolation and Durability

is withdrawn and the component is given a chance to copy its modified clone data back into the *Store Provider* before the exclusive write lock request is granted. This is usually done by the *CPC Track* module.

Abstract Super-Class: To simplify future *Store Provider* implementations, an abstract super-class for store providers is provided. It addresses most of the not directly persistence provider related tasks like locking, caching and event generation. The implementor of a new *Store Provider* can therefore concentrate entirely on the persistence provider specific issues.

The *Store Provider* API thus provides a flexible base for the implementation of arbitrary persistence provider wrappers.

Default Implementation

Initially, a number of different persistence approaches for the default *Store Provider* implementation were evaluated. While all of the examined persistence solutions could have been used to implement the *Store Provider* API, an *HSQL DB* based approach was finally adopted [7].

File Based: The initial idea was to store clone data in *XML* files within the plug-in state directory. And while this entails the lowest runtime overhead in terms of libraries and additional persistence applications, creation and maintenance of the required caching structures, to allow fast clone data access even within large projects, would have required a considerable implementation and testing effort.

Other evaluated approaches which fall into this category are use of the *Java Serializable* infrastructure and the *Castor* persistence framework [3].

OO DB Based: The evaluated open source object database *db4o* provided some very interesting capabilities. However, its lack of built-in uniqueness and integrity constraints was considered a severe shortcoming [4].

SQL DB Based: A *SQL* based approach has the interesting property of being able to support internal and external databases without additional effort. *HSQL DB* emerged as the most promising candidate for an integrated *DBMS* with a footprint actually small enough to be run inside the Eclipse IDE. Furthermore, *PostgreSQL* was evaluated as an external *DBMS* [9].

However, the object-relational mapping of *CPC* clone data objects to an *SQL DBMS* proved to be problematic. The evaluated solutions were either too heavy weight for use within the Eclipse IDE (*i.e.* *Hibernate*) or did not provide the required flexibility to allow a truly clone data object independent mapping. One of the design goals was to keep the data object and the persistence implementations strictly separated. Any mapping approach which can't be configured to make use of the *ISerializable* interface will not be able to achieve this goal. The implementation of a small, custom mapper for the *ISerializable* interface was thus required.

The current default *Store Provider* can be used to either store clone data in an internal *HSQL DB* database or an external *PostgreSQL* database. By default the internal *HSQL DB* database is used.

However, there may be scenarios in which use of an external database provides additional benefits. A locally running, external database may provide performance improvements for very large projects and a central database may offer very interesting, real time cloning information which could be helpful in experimental setups. Support for other *SQL* databases could be added with minimal effort.

CPC Core API Specification:

`IStoreProvider*`, `IStoreProviderWriteLockHook`, `IStatefulObject`

3.4.4 CPC Mapping - Data Mapping

A *Mapping Provider* enables other components to map *CPC* clone data objects into a string representation and to convert such representations back into clone data objects. Each *Mapping Provider* implementation can freely define its own string representation format and multiple such implementations can coexist within the *CPC* framework.

The *Mapping Registry* enables a component to parse an arbitrary string representation as long as it is supported by at least one of the installed *Mapping Providers*. This enables free transitions from one *Mapping Provider* implementation to another without a need for explicitly converting any of the old data.

CPC ships with a simple *XML Mapping Provider* which can be used to convert *CPC* clone data into a *XML* representation. *Mapping Providers* are used by a number of components within the *CPC* framework. Most notably by the *CPC Store Remote* module (see section 3.4.9) and the *CPC Exports* module (see section 3.4.10).

CPC Core API Specification:

`IMappingProvider`, `IMappingRegistry`

3.4.5 CPC Classification - Clone Categorisation

CPC classifies each clone on creation and potentially re-classifies clones at specific points during their life time, i.e. after a major modification of the content of a clone. A *Classification Provider* provides the necessary services to all interested parties within the *CPC* framework.

It serves two main purposes. First, it can reject a clone instance during evaluation. This can be used to filter out specific types of clones. A typical example would be to reject all clones which fail to reach a specific minimal size or complexity. And second, it can attach an arbitrary number of classifications to each clone which are then persisted as part of the clone object and can from there on be used by other components in their decision making.

Classifications are simple string values and while the *Classification Provider* API specifies some predefined classifications, contributors are free to add their own. A *Classification Provider* receives additional context information on each classification request. The provider and its strategies can thus also alter their processing dependent on factors other than the content of a clone, like the type of the classification (*initial*, *incremental*, *re-classification*) or the properties of its origin. The default classifications as well as the implemented classification strategies are covered in section 4.1.

CPC Core API Specification:

`IClassificationProvider, IClassificationStrategy`

3.4.6 CPC Similarity - Semantic Equivalence and Differences

Determining the similarity between two clones is a very common problem in the *CPC* framework. A *Similarity Provider* enables other components to easily calculate the percentage of similarity between two clone instances.

The API specification defines neither how similarity is to be calculated nor the exact meaning of the returned similarity value. Only a similarity of 100% is clearly defined. It requires guaranteed semantic equivalence between two clones.

The key users of *Similarity Providers* are the *CPC Track* and *CPC Notification* modules. The supplied default *Similarity Provider* implementation and its employed strategies for similarity evaluations are described in section 4.2.

CPC Core API Specification:

`ISimilarityProvider, ISimilarityStrategy, ISimilarityStrategyTask`

3.4.7 CPC Notification - Clone Modification Warnings

Providing the developer with relevant warnings about potential update anomalies is one of the main goals of *CPC*. However, so far the general understanding of the underlying processes and problems is very limited. For the short and medium term, a flexible, heuristics based approach seems most likely to be able to provide tangible benefits.

The *CPC* framework tries to address the high uncertainty about future developments in this area with a multi-tiered approach. The *CPC Notification* module provides a generic harness for notification handling. The actual decisions about clone modifications are delegated to a *Notification Evaluation Provider* and a *Notification Delay Provider* is used to provide support for delayed warnings.

Figure 3.14 displays the interactions between the components after the modification of a clone. If a developer modifies the contents of a clone, the *CPC Notification* module will, after some intermediate stages, receive a clone modification event. It will then ask a *Notification Evaluation Provider* to evaluate the modification. The provider has a number of options. A typical response to an initial evaluation might be to delay it until it has become clear that the developer has finished his current task. In this case, the *CPC Notification* module will relay the information to a *Notification Delay Provider* which will enqueue it for future processing.

It is up to the *Notification Delay Provider* implementation how queued events are handled. Typical scenarios might be to reevaluate queued events after a specific amount of time has passed or once the file in question is closed by the developer. Once a *Notification Delay Provider* has determined that a queued event is ready for reevaluation, it will again be relayed to a *Notification Evaluation Provider*. By now actions of the developer might have made the notification unnecessary. If the notification is still appropriate, it will be dispatched to the *CPC Notification UI* module.

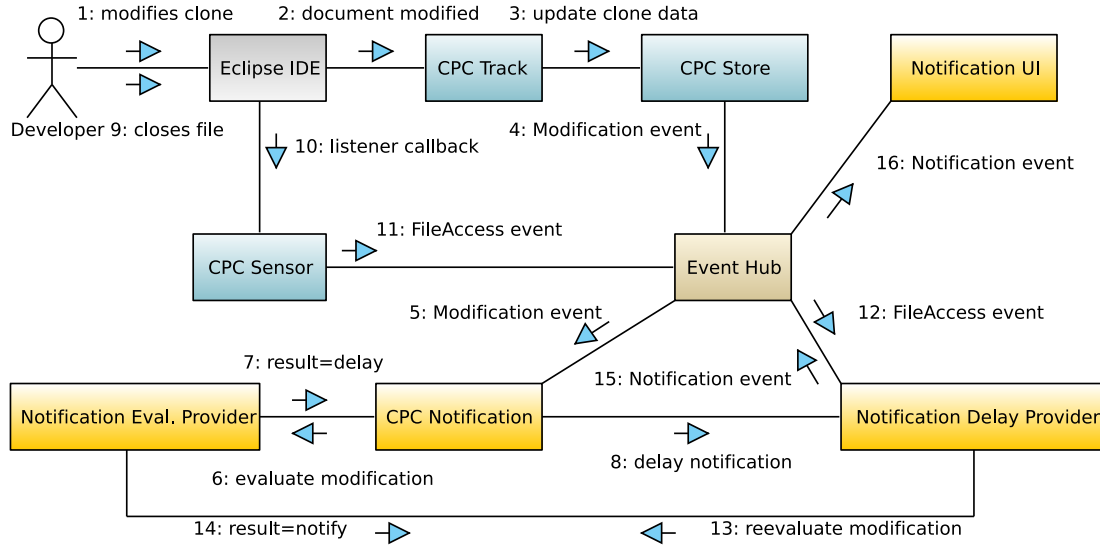


Figure 3.14: CUD: Module collaboration during the handling of a clone modification (*simplified*)

It will display the notifications to the user and it furthermore handles all notification related user interactions (*see section 3.4.11*).

The concrete *Notification Evaluation Provider* and *Notification Delay Provider* implementations provided by *CPC* and their strategies are discussed in detail in section 4.3.

CPC Core API Specification:

`INotificationEvaluationProvider, IEvaluationResult, INotificationDelayProvider`

3.4.8 CPC Reconciler - External Modification Handling

While modification of source code files by external tools outside of the supervision of the Eclipse IDE (*and therefore CPC*) is strongly discouraged, such modifications can never be entirely ruled out. Resilience against such modifications was thus one of the key requirements for *CPC* and constitutes one of the main advantages of the adopted clone data storage mechanisms when compared with the built-in *Marker* support of Eclipse (*see section 5.3.2*).

The *CPC Reconciler* module listens for file access events and checks every opened file for potential external modifications before it is displayed to the user. No other *CPC* module has any notion about ‘reconciliation’ or any dependency on the *CPC Reconciler* module. The high priority of the registered synchronous event listener ensures that the *CPC Reconciler* module can examine and reconcile opened files before the *CPC Track* module, which reacts to the the same event, is notified about them (*shown in figure 3.15, event hub and some events have been omitted for brevity*).

Reconciliation is made possible by the fact that the *Store Provider* keeps a copy of the source file content and clone contents at the time of the last successful save point. The file size and modification date fields which are part of the `ICloneFile` data object furthermore allow fast checking of files for external modifications.

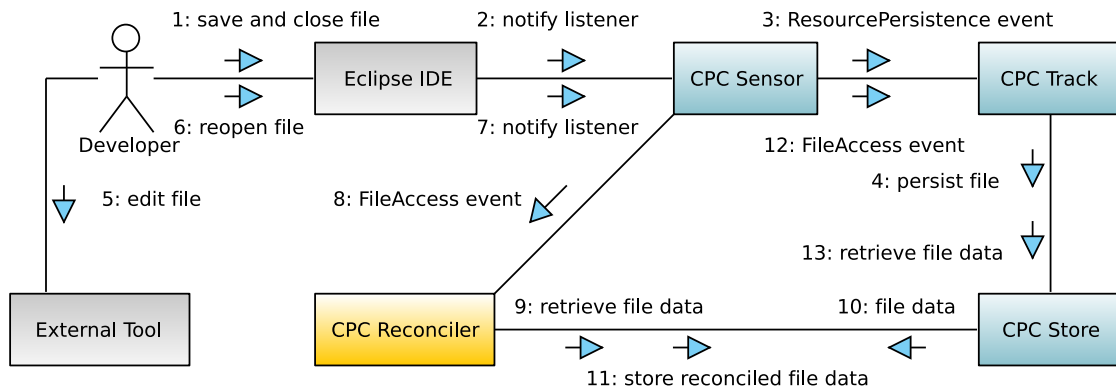


Figure 3.15: CUD: Module interaction during reconciliation of external modifications (*simplified*)

The *Reconciler Provider* API specification does not define how clone positions are to be reconciled. This is left to the implementation and it is believed that this is an area with considerable potential for improvement.

Default Implementation

The default *Reconciler Provider* implementation uses a strategy based approach as outlined in section on multi-level reuse (3.2). Strategies can be contributed via the corresponding *CPC Reconciler* extension point. First the file is checked for external modifications. If the file has not been modified since it has last been saved under supervision of *CPC* nothing needs to be done. Files which do not contain any clones are also ignored.

The *Reconciler Provider* then executes some common calculations and builds up some data structures which are likely to be needed by a number of strategies. This is done to improve performance by allowing strategies to reuse the pre-processed data. The most important pre-processing is the application of a character based diff algorithm provided by a *Diff Provider* in order to approximate the potential nature of the external modification¹⁷.

The actual processing is then passed on to the registered strategies which are executed in order of their priority. The default implementation ships with the following three strategies (*listed in order of execution*).

All Changes After Clones: The first strategy checks whether the modification in the file might be located after all the known clone areas. In this case nothing needs to be done as the positions of the corresponding clone entries will not have been affected in any way.

Whitespace Only Change: The second strategy checks whether the modification affected only whitespace characters. This is typically caused by automated source code reformat operations. If this is the case, the ‘non-whitespace position’ of each clone in the original document is calculated and is then used to extract the new clone positions within the modified document.

¹⁷The current *Diff Provider* uses Google’s implementation of Myer’s diff algorithm: <http://code.google.com/p/google-diff-match-patch>

Diff-based Approach: If all earlier strategies fail to reconcile the modification, the edits extracted by the diff algorithm are used to ‘replay’ all modifications which were made to the document. The position updating is delegated to the normal *CPC* position tracking code (see section 3.4.2).

The three default strategies yield very good reconciliation results for most situations. Though their inherent reliance on the results of a diff algorithm is their weakness. A diff algorithm only returns one of the potential edits which could result in the final document state. There is no guarantee that the result actually represents the real modifications made. As such the reconciliation results obtained are only reliable in situations where the diff algorithm returned sensible output.

Future Improvements

There are a number of possible, future improvements. Of key importance would be the addition of some strategies which do not rely on the output of a diff algorithm and which could thus provide a safety harness for situations in which the diff algorithm could potentially endanger the integrity of the clone positions.

Some starting points could be the fact that the current strategies make no use of the known clone contents, a string based search might be able to reidentify clones which were only moved but not modified. Kim et al. provide some interesting ideas in this area [39]. As well as the application of more robust, approximate position tracking schemes like *Clone Region Descriptors* [23].

CPC Core API Specification:

IReconcilerProvider, IReconcilerStrategy

Service Provider Dependencies:

IDiffProvider, IPositionUpdateStrategyProvider

3.4.9 CPC Store Remote - Remote Synchronisation

Automated synchronisation of clone data across all workstations of a development team was one of the key requirements for *CPC*. Such synchronisation can be achieved in a number of ways. After some consideration, a synchronisation scheme which is aligned with repository operations of the developer and which exchanges *CPC* clone data either via a *source code repository* or via a central *SQL DBMS* emerged as the most interesting approach.

In such a scheme the *CPC* clone data for a file would need to be written to the central *CPC* data storage whenever the file is committed and data would need to be read from the central storage whenever a local file is updated. The same goes for all other repository operations like checkouts, rollbacks or branch/tag changes. The correct alignment of these operations at all times is crucial.

When using a *source code repository* for synchronisation, a central question emerges, should the *CPC* data be stored within the same repository as the source of the software application or in a different repository? The reduced configuration and administration overhead of using a shared repository would be likely to make this approach attractive for a large user base. On the other

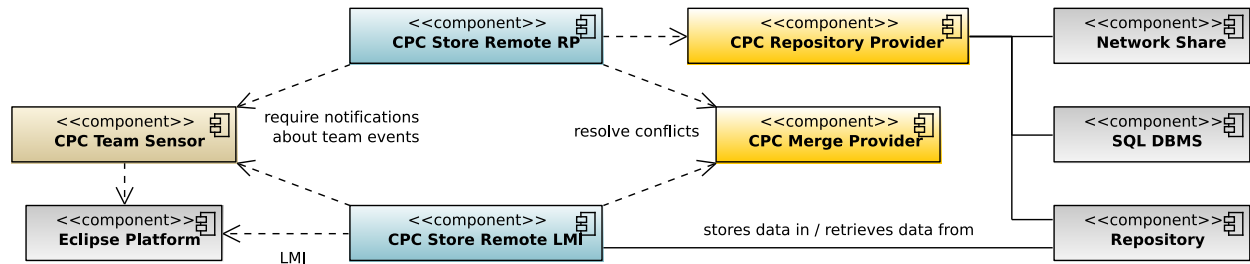


Figure 3.16: COD: Structure of the CPC Remote Synchronisation framework (*simplified*)

hand *CPC* data files would clutter the repository and might accidentally be modified or deleted by a user.

The *CPC* remote synchronisation framework strives to support most interesting scenarios. Unfortunately, clone data storage within the same repository as the source code requires the use of a specific set of Eclipse APIs which don't integrate well with the remaining approaches. *CPC* thus offers two separate remote synchronisation modules which share some common code but adapt to different Eclipse APIs.

Figure 3.16 shows a simplified version of the resulting remote synchronisation framework. Of key importance is the *CPC Team Sensor* which detects repository operations of the developer and generates corresponding *CPC* events. The *CPC Store Remote LMI* module provides synchronisation support based on storing *CPC* data within the same repository as the source code. The *CPC Store Remote RP* module provides the harness for synchronisation approaches which store the clone data in other locations and can use all available *CPC Repository Providers* as adapters for the actual storage backends. Both modules make use of some common functionality like the *CPC Merge Provider*.

The following sections provide more details about the design and implementation of these components. Most sections refer to the *Challenges and Setbacks – Team Providers* section 5.4 for more details about related problems and limitations.

CPC Team Sensor

The *CPC Team Sensor* has emerged as one of the most problematic aspects of the remote synchronisation framework. Neither Eclipse nor the common team provider plug-ins provide an API for registering team operation listeners. This seemingly simple but absolutely crucial module thus became the main obstacle for the successful completion of *CPC*. Section 5.4 provides an extensive overview of the problems faced and the current status.

Once this sensor detects a repository operation it generates a team event which includes information about the type of operation, the affected resource as well as the old and new revision identifiers of the resource as provided by the repository. With the current Eclipse team APIs, a separate team sensor for each *source code repository* team provider plug-in (*CVS*, *Subversive SVN*, *Subclipse SVN*, ...) is required. This module would become superfluous, should the ongoing discussion about potential listener extensions to the Eclipse team API result in corresponding API changes (see section 5.4), as the team event generation could then be included in the normal *CPC*

Sensor module.

CPC Core API Specification:

EclipseTeamEvent

CPC Store Remote LMI

Eclipse 3.2 introduced the notion of *Logical Model Integration (LMI)*, a special set of APIs which allow model providers to participate in team operations [10]. The *LMI* API is aimed at plug-ins which contribute new data models which can not be directly mapped to a specific file resource. One example would be a new *UML* editor which persists the data of one *UML* model in multiple files. The *LMI* API allows a plug-in to specify that certain sets of files should always be grouped together for repository operations. It furthermore allows a plug-in to participate in the handling of merge conflicts.

While this API was not designed for the needs of *CPC*, it does provide a good starting point. The *CPC Store Remote LMI* module ensures that the latest *CPC* clone data is always stored in a hidden ‘.cpc’ directory and contributes a new *LMI* model which groups each source file with its corresponding *CPC* clone data file. The clone data files are kept up to date by listening for persistence events generated by the *Store Provider* and are created by using a *Mapping Provider* to convert the *CPC* clone data objects into a string representation which can be written to the *CPC* clone data file. This way the *CPC* data will automatically be transmitted to the repository whenever the user commits a file.

Unfortunately, there are some important aspects of the *LMI* API which do not align well with some of the needs of *CPC*. This leads to inconveniences and serious problems. This is also covered in section 5.4.

Service Provider Dependencies:

IMappingProvider, IMergeProvider

CPC Store Remote RP

As the Eclipse *LMI* API is currently only implemented by a single team provider, the *CVS* provider which ships with Eclipse, the *CPC Store Remote RP* module takes a different approach and completely avoids the *LMI* API. Instead it is based on *CPC Repository Providers* which act as adapters to arbitrary storage backends. This also includes source repositories and in theory this approach could also be used to commit *CPC* data files to the same repository as the source files, though this could lead to problems as the data would be committed in two separate transactions¹⁸.

The *CPC Store Remote RP* module listens for team events generated by the *CPC Team Sensor* module and sends or retrieves the corresponding data via the configured *CPC Repository Provider*. In case a source file needs to be merged, the *CPC Store Remote RP* module will use the available Eclipse team APIs and the *CPC Repository Provider* to collect a full set of source and clone data for the base, local and remote revisions and will delegate the merge to a *Merge Provider*.

¹⁸For *source code repositories* which don’t support transactions, like *CVS*, there would be little difference.

One of the potential problems of such a separated approach are concurrent commit and update operations for the same file. By always accessing or updating the *CPC Repository Provider* based data *after* the corresponding repository operation has finished and by using the unique revision identifiers returned by the repository as lookup and storage keys the repository itself can be used as a synchronising central entity. This way, most timing issues can be addressed by reexecuting failed *CPC Repository Provider* lookups after a specific delay.

However, one central problem remains. Due to a network connectivity failure or other similar events it is possible for *CPC* clone data and source files to reach a non-synchronised state. I.e. the network connection could drop right after a source file was committed to the repository but before *CPC* could send the corresponding clone data or the connection might drop during an update operation, right after the local source file was updated but before the corresponding *CPC* clone data could be retrieved. As one of the *CPC* requirements explicitly stated that *CPC* should not affect the normal programming practice of its users, any approach which would prevent the developer from working on the non-synchronised file is clearly out of question. In such situations *CPC* can thus only fall back to the available clone data and treat the changes in the source file like external file modifications.

Service Provider Dependencies:

ICPCRepositoryProvider, IMergeProvider

CPC Repository Provider

A *CPC Repository Provider* represents an adapter for a specific storage back end. A *SQL DBMS*, a *source code repository* or a central network share are just some examples of potential storage back ends which might be of interest. The *CPC Repository Provider* API specifies only a small number of simple operations to store and retrieve a set of clone data objects by revision identifier and file identifier. It does not require an implementation to provide full transactional properties and only specifies that concurrent store and retrieve operations must not return partial or corrupt data. Most *CPC Repository Providers* are likely to internally use a *Mapping Provider* in order to convert the clone data objects into a string representation which can be written to a file.

Currently only a simple, *SQL DBMS* based *CPC Repository Provider* is available, but further providers could easily be added due to the limited requirements of the *CPC Repository Provider* API. Even implementations based on very simple protocols like *FTP* would be possible as long as a test-and-set operation or some other means of locking resources exist.

CPC Core API Specification:

ICPCRepositoryProvider

Service Provider Dependencies:

IMappingProvider

CPC Merge Provider

A *CPC Merge Provider* uses the clone and content data for the base, local and remote revisions of a source file and tries to reconcile any conflicts by merging the clone data to correctly reflect the new

contents of the merged source file. The merging of the source file itself is left to the Eclipse team repository plug-in implementation¹⁹ and has to occur before the *CPC Merge Provider* is executed. The presence of *CPC* thus does not affect the merging of source files.

Like many other *CPC* providers the *CPC Merge Provider* delegates the merge task to the registered *Merge Strategies*. The following list provides an overview of the currently implemented strategies in order of their execution.

Locally Unmodified: This is the simplest case. If the content of the local source file prior to the merge equals the content of the base revision, the remote clone data can simply be copied over as the resulting content of the ‘merged’ source file will correspond to the remote source content.

Clone Deletions: This strategy represents part of a three-way-merge. It compares the local and remote clone data with the clone data of the base revision and detects clone instances which were locally or remotely deleted. In this simple approach any clone instance which was deleted locally or remotely is filtered out and will not be part of the final merge result.

The rare situation in which a clone instance was deleted locally or remotely but where a merge conflict between the local and remote source files resulted in a merged file which still contains the removed clone’s code is a special case which is not covered by the current implementation.

Full Merge: This strategy implements the remaining part of a three-way-merge. First the new clone positions in the merged source file for the local and remote clone data are extracted with help of a *Reconciler Provider*²⁰. Clones which were locally or remotely added form part of the final merge result. This leaves only clones which exist locally and remotely. Such clones are added to the merge result as long as either the local or the remote clone could still be identified within the merged source file.

Clones which can not be found within the merged source file represent a special case which is currently not handled and are discarded. This also applies to clones for which the reconciliation yields different positions according to the local and remote clone data.

3rd parties can contribute additional *Merge Strategies* as well as special *Clone Object Extension Merge Strategies* which can be used to provide special merge semantics for 3rd party clone data object extensions. Overall the currently implemented strategies should cover most of the common situations but fail to take care of certain special cases.

CPC Core API Specification:

IMergeProvider, IMergeStrategy, ICloneObjectExtensionMerger,
ICloneObjectExtensionMergeStrategy

3.4.10 CPC Imports and Exports

Clone data import and export are an important functionality within the *CPC* framework. Reuse possibilities, due to the obvious commonalities between import and export activities, are leveraged

¹⁹I.e. CVS, Subversive SVN, Subclipse SVN and others

²⁰The merged source file is treated similar to an external modification.

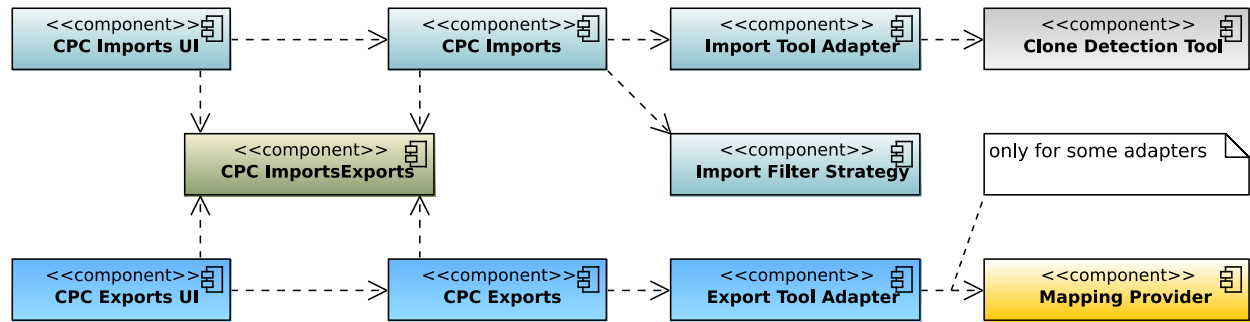


Figure 3.17: COD: Structure of the CPC Imports/Exports framework (*simplified*)

by means of a common imports/exports core. This core is then extended by flexible imports and exports subsystems which in turn delegate most of the key functionality to a set of registered adapters and strategies (*see figure 3.17*).

The Common Core

Most of the imports and exports related codebase is part of the generic imports/exports core. A typical example of the reuse possibilities are the UI wizards used during im- and export, they are highly similar. The overall processing and the employed extension strategies are other points of high similarity. The common core helps to lower the complexity of the imports and exports subsystems considerably.

Both, imports and exports subsystem are strictly separated into a UI and a backend part. This allows other components of the *CPC* framework to employ the services of these subsystem backends as well, if they are required.

The Imports Subsystem

One of the major, potential obstacles for the adoption of *CPC* is the fact that most cloning activities will already have happened if *CPC* is introduced into a running project. One potential remedy for this limitation could be the use of static clone detection techniques to ‘jump start’ the *CPC* clone database. The *CPC Imports* module thus provides a generic framework for all kinds of clone data import approaches.

The *CPC Imports* module introduces the notion of *Import Tool Adapters*, special modules which act as a bridge between the interfaces required by the *CPC Imports* module and any potential clone detection tool. *Import Filter Strategies* furthermore support the post-processing of static clone detection results to prune out potential false positives and clones which are not of interest to *CPC*. Arbitrary *Import Tool Adapters* and *Import Filter Strategies* can be registered via the corresponding extension point of the *CPC Imports* module.

The import wizard UI of the *CPC Imports UI* module offers a selection of all registered *Import Tool Adapters* and provides the user with all available configuration options for the selected adapter. Simple configuration scenarios can be realised purely declaratively and do not require any code contribution by an *Import Tool Adapter*. More advanced adapters can contribute their own wizard

pages and dialogs.

To emphasise the fact that a considerable number of the imported clones may represent false positives or accidental clones [13], each imported clone is clearly marked and other *CPC* components can take this fact into account during their processing of an imported clone instance.

While the legacy clone data import itself was not part of the goals of this thesis, a proof of concept implementation was required to ensure the adequateness of the imports APIs. To this end three static clone detectors were evaluated. *CCFinder*, a token based clone detector developed at Osaka University, well known for its high recall and speed [36]. *CloneMiner*, another token based clone detection tool developed at the National University of Singapore [15, 16]. As well as, *SDD for Eclipse*, a small Eclipse plug-in using the SDD clone detection algorithm [34, 35].

Even though *CCFinder* and *CloneMiner* produced better clone detection results other limitations made their adoption as a proof of concept implementation impractical. Both tools are platform dependent and only run on *Windows* systems. This limits the potential userbase for such an approach and also implies that a tight integration into a *Java* based *Import Tool Adapter* is not easy. Distribution rights are another problematic aspect. *CCFinder* may not be distributed, every user of a *CCFinder* based *Import Tool Adapter* would thus have to manually request a free licence via the *CCFinder* website. *CloneMiner* is still under development and is currently not publicly available.

SDD for Eclipse was thus chosen as base for the proof of concept *Import Tool Adapter* implementation. It could be tightly integrated into the *CPC* system and its use requires no additional software installation or configuration by the end user. In the long run *Import Tool Adapters* for better clone detection tools and sophisticated *Import Filter Strategies* would be needed to provide a good starting point for the introduction of *CPC* into running projects.

CPC Core API Specification:

IImportController, IImportToolAdapter, IImportFilterStrategy

The Exports Subsystem

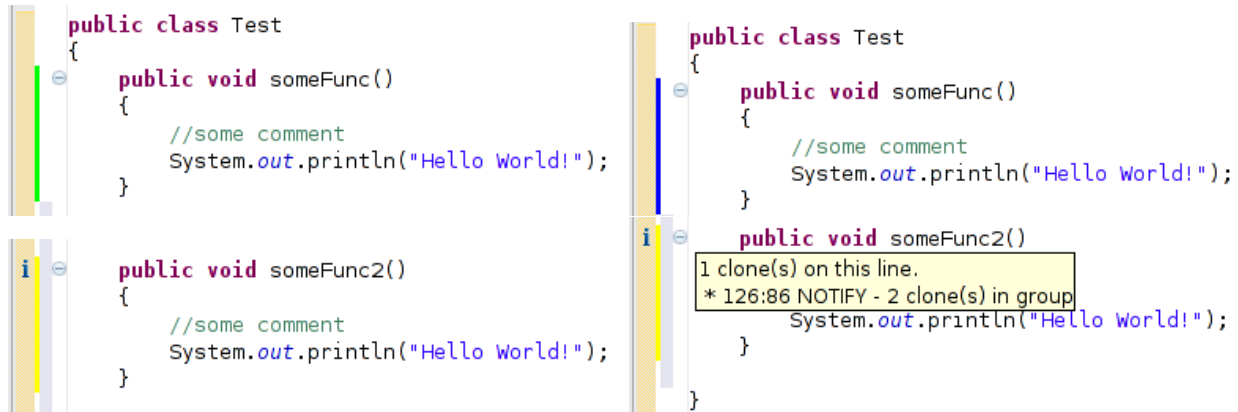
In order to facilitate data collection for future improvements of *CPC* a way of exporting clone data in a *Store Provider* independent²¹ way was required. The *CPC Exports* module provides the required functionality and is very similar in structure to the *CPC Imports* module.

The actual export functionality is encapsulated by an *Export Tool Adapter*. Currently *CPC* ships with an adapter which provides file based exports and uses one of the available *Mapping Providers* to determine the internal format of the export data files²².

However, an *Export Tool Adapter* is not limited to exporting clone data into files. Other future possibilities could be adapters for export of clone data into databases or adapters for transmission of clone data to special clone data servers. Such adapters could be used as a base for future, automated online clone data submissions by interested parties.

²¹Simply copying the internal data files of a *Store Provider* would be a possibility.

²²As there is currently only a *XML* based *Mapping Provider* shipped with *CPC*, *XML* data files are the only export format available.

Figure 3.18: CPC UI clone marking via coloured bars (*rulers*)

CPC Core API Specification:

`IExportController, IExportToolAdapter`

Service Provider Dependencies:

`IMappingProvider`

3.4.11 CPC UI and Notifications UI

The *CPC UI* and *CPC Notification UI* as well as the *CPC Imports UI* and *CPC Exports UI* contribute a number of user interface additions to the Eclipse IDE. Overall the current *CPC* user-interface clearly falls short of our long-term vision. The provided visualisations are relatively simple and fail to convey a good overview of the cloning situation on a project level. Extendibility and flexibility were thus, once again, the most important factors in the design phase.

All the UI modules are strictly separated from the underlying program logic of *CPC* and can easily be extended or replaced. A 3rd party contributor has access to all data and the full range of APIs and can thus implement completely new clone data visualisations. Specific extension points also allow addition of new functionalities to the existing *CPC* views, i.e. a new context menu option. This enables future UI plug-ins to tightly integrate with the existing *CPC* views.

The remainder of this section will give a very brief overview over some of the current *CPC* UI elements. Further screenshots and descriptions can be found on the *CPC* website²³.

The CPC User Interface

One of the key goals of *CPC* is the improvement of the overall awareness of the cloning situation within a software system. It was thus decided that one of the best initial approaches towards this long-term goal would be a non-intrusive but constant clone information feedback for the developer. This is achieved by adding a small coloured bar at the left side of the screen as shown in figure 3.18, these so called **rulers** are displayed next to each clone instance.

²³Official *CPC* website: <http://cpc.anetwork.de>

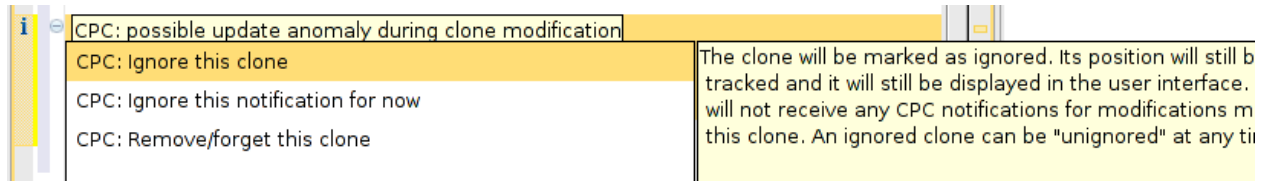


Figure 3.19: Actions available via the notification icon in the editor ruler

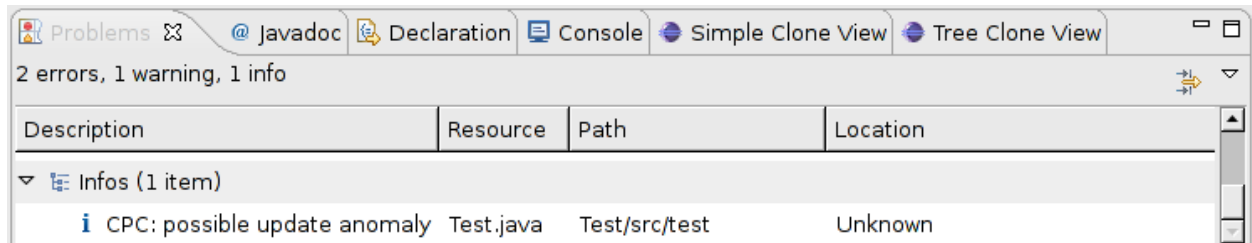


Figure 3.20: CPC notification in problems view

The colour provides immediate feedback about the state of the corresponding clone. A green ruler marks clones which are semantically equal²⁴ with all other members of their respective clone group. Once one of the group members was modified, the ruler switches to a blue colour. Clones for which the notification framework has issued an update anomaly notification are highlighted with a yellow ruler or, in case of warnings, a red ruler. Further ruler colours are grey for orphans, light grey for ignored clones and black for lines which contain more than one clone. If a contributor requires more colours, the ruler implementation could easily be extended.

A click on the ruler will highlight the exact extent of the clone²⁵ and selects the clone in all *CPC* clone data views. Hovering over the ruler furthermore displays a small tooltip with additional information like the size of the clone, its state and a listing of all files which contain members of the clone's group.

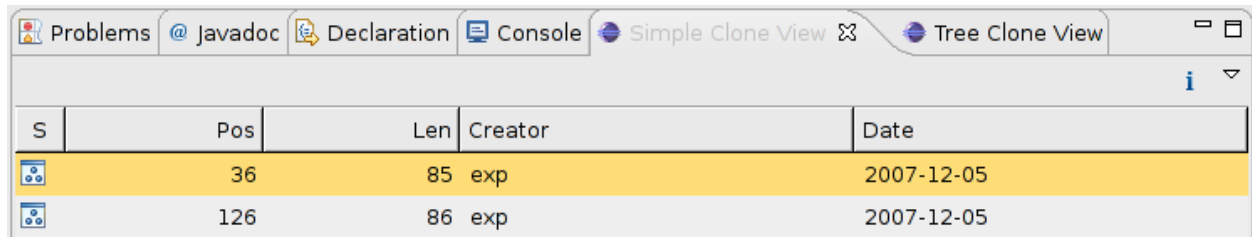
Update anomaly notifications are additionally marked with an Eclipse information marker. Clicking on such a marker will reveal the *CPC* quick fix context menu which provides the developer with a number of options as seen in figure 3.19. By default *CPC* offers three choices which can easily be extended with contributions by 3rd party plug-ins. A clone can be marked as ignored, no more notifications will be generated but the clone's position is still tracked. The notification itself can be dismissed in which case the notification is removed and the clone enters the modified state until a new modification of the clone content triggers a new notification. The third option removes the clone entirely.

As can be seen in figure 3.20, update anomaly notifications will also be visible in the Eclipse problems view. The visible description in the problems view as well as the marker tooltips corresponds to the message returned by the *Notification Evaluation Provider*. Notifications are also indicated in the overview ruler at the right side of all Eclipse editors.

Figure 3.21 and figure 3.22 show the two main clone data views available in the current version

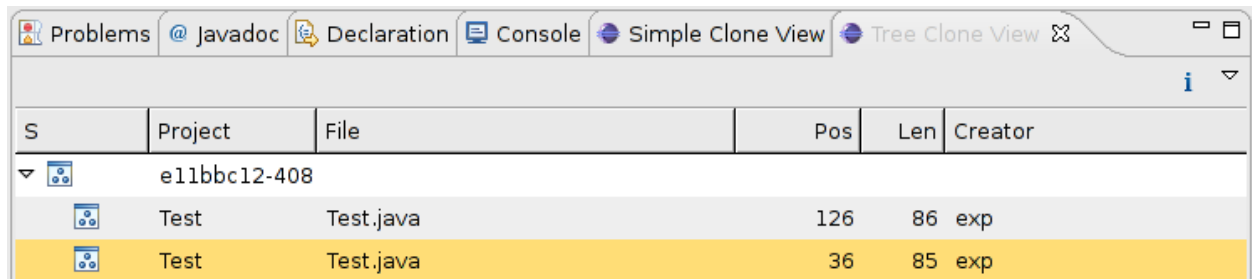
²⁴According to the installed *Similarity Provider* and *Notification Evaluation Provider*.

²⁵A ruler can only convey line range information. By clicking on the ruler the user can also see the exact start and end positions within the first and last line.



S	Pos	Len	Creator	Date
	36	85	exp	2007-12-05
	126	86	exp	2007-12-05

Figure 3.21: Simple CPC clone data in list viewer



S	Project	File	Pos	Len	Creator
▼	e11bbc12-408				
	Test	Test.java	126	86	exp
	Test	Test.java	36	85	exp

Figure 3.22: CPC clone data tree viewer

of *CPC*. A list based, simple view lists all clones within the current file whereas a tree based view can be used to obtain an overview of clone groups. The *Tree Clone View* displays all clone groups for which at least one member is located within the currently open source editor. Aside of the ruler tooltips this is another way of obtaining an overview of all files which contain members of a specific clone group.

The large amount of information about clone modifications which *CPC* needs to collect in order to provide sophisticated notification strategies and other potential future extensions like refactoring support with a set of data to work on can also be accessed by the developer via a special *Clone Replay View* shown in figure 3.23. The view is based on prior work at the Freie Universität Berlin [41] and enables the user to examine the evolution of a clone step by step. Specific modifications can be selected individually or the modification process can be ‘replayed’ like a movie, either in real time or at a specified event rate.

Figure 3.24 displays a part of the *CPC* preferences dialog. By tightly integrating with the Eclipse preferences system, *CPC* allows 3rd parties to contribute additional preferences pages with minimal effort. The adopted, loose structure of the preferences dialog leaves ample room for future expansion. A new plug-in would typically add a new subtree to the configuration tree whereas a new strategy would add its new pages below the entry of the plug-in which it is extending. If needed all preferences settings can also be directly specified via configuration files or programmatically by means of the Eclipse preferences API.

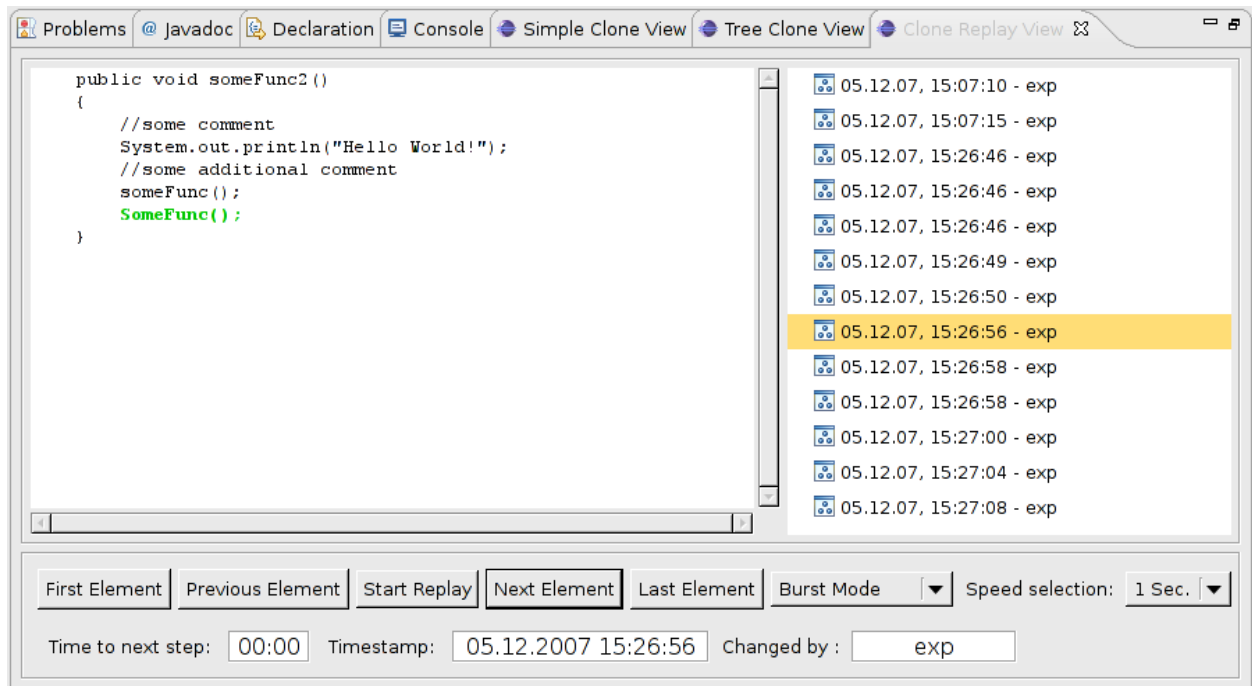


Figure 3.23: CPC clone history replay view

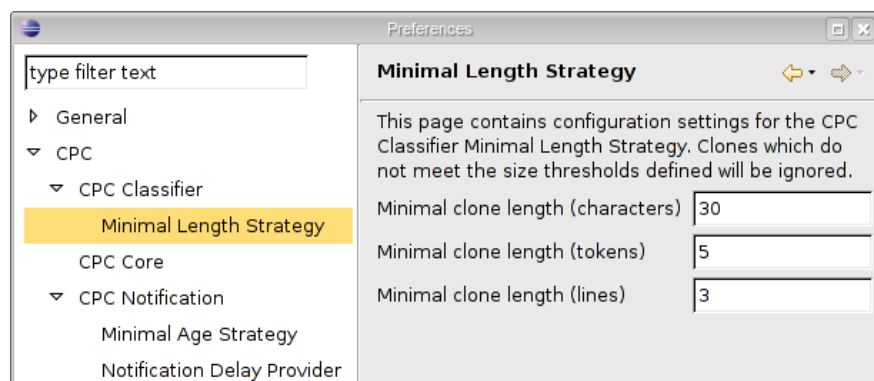


Figure 3.24: Extendible CPC preferences dialogs

Chapter 4

Heuristics

It's not at all important to get it right the first time.

It's vitally important to get it right the last time.

[Andrew Hunt and David Thomas]

It was clear from the very outset of this project that the first version of *CPC* would only be able to provide very basic heuristics as neither the copy-and-paste and cloning data nor the time needed for the development of sophisticated and sound heuristics was available. This chapter describes the basic heuristics which are currently part of *CPC* and provides an outlook on potential future possibilities.

4.1 Clone Classification

The ‘correct’ classification of a clone instance is the first step towards the automated detection of potential clone related update anomalies. However, even after more than a decade of cloning research, this first step still holds unsolved problems.

Related Work

Past efforts in the field of cloning research have produced a multitude of classification systems focusing on all kinds of aspects of code duplication [51]. Yet, most of these approaches are focusing on commonalities and differences between clones and are thus more suited for discussion in the following clone similarity section.

The classification of clones into *syntactic templates* and *semantic templates* introduced by Kim et al. focuses more on the intention of the programmer [38]. They argue that a programmer either wishes to reuse the syntax or the semantics of a copied code section. They further subdivide *semantic templates* into four subcategories: *design pattern*, *usage of a module*, *implementation of a module* and *control structure*. It remains unclear whether a classification according to these subcategories, assuming that it would be feasible, would be beneficial for *CPC*. However, a classification into *syntactic templates* and *semantic templates* would be very interesting. Unfortunately the distinction between these two classifications poses a real challenge. The *CPC* classification API

by default contains a ‘template’ category for use in cases where a clone most likely represents a *syntactic template*.

Another classification presented focuses on the type of region the clone is located in. I.e. whether it represents an entire class, an entire method, the beginning or the end of a method, parts within a method, a loop, parts within a loop or conditional statements.

A separate area of discussion within the static clone detection research community is the minimal size or complexity of a clone [51, 13, 45]. It is argued that knowledge about trivial clones provides no tangible benefits and increases the risk of false positives. False positives are of no concern for the copy and paste based approach of *CPC*. However, a suitable minimal threshold for clone sizes provides a good filter which can reduce the number of tracked clones significantly. Typical thresholds found in literature are 20 to 30 tokens for token based approaches and 5 to 50 lines for textual matching based approaches. It remains unclear what the best threshold for *CPC* would be.

Current Approach

The *CPC Classifier* module is structured like most other *CPC* modules (*see section 3.2*) and delegates all processing and decision making to a number of registered strategies. The classification strategies currently shipped with *CPC* are very basic and leave a lot of room for improvement.

Minimal Length: The first strategy checks whether the clone satisfies the minimal size requirement. Clones which are too small or too simple are likely to be of no or only very little interest and will be rejected by this strategy. Size is measured in characters and lines. Complexity is measured in number of tokens. A clone needs to pass the configured thresholds for all three metrics to be accepted.

Classification of Origin: The *Classification Provider Strategy* API provides context information for each classification request. This strategy only applies to initial classifications of new clone instances for which the origin clone is available.

In this case the strategy checks whether the content of the clone matches that of its origin and if it does, all classifications of the origin are copied over to the new clone instance. The processing is aborted at this point and no further strategies will be executed.

The rationale for this approach is twofold. First, some classification strategies may depend on syntactic validity of a clone for their processing and the pasted code may not be valid in its new location whereas the likelihood of validity at the origin is much higher. And second, if the contents of the pasted clone and its origin match, their classifications will usually match too. In this case copying the classifications of the origin yields a performance improvement.

Classification by Content: If none of the previous classifications rejected the clone or aborted the classification process, this strategy will make use of the internal AST of Eclipse to classify the clone according to the *Java* language elements it contains.

The resulting classifications indicate whether the clone contains a class, one or more methods, loops or conditional expressions. Furthermore, some checks for very simple cases are

performed and matching clones are classified accordingly. Namely clones which exactly represent an identifier and clones which contain only comments and/or whitespaces.

Additionally, all language elements which are part of the clone are assigned specific weights and an overall complexity value for the entire clone content is calculated. If this value exceeds a configured threshold, the clone is marked as ‘complex’.

The *Classification Provider* API also specifies the classification ‘template’, however simple, experimental strategy implementations for classifications of this type have failed to produce any significant results and are therefore currently not part of *CPC*.

Future Improvements

A first step toward improving the classification results could be to devise additional classifications which might be of interest. The currently defined classifications only cover complete language constructs. While this information is certainly important, it may not be enough for a broad classification of typical clones (*see section 6.2.2*).

A reliable detection of *syntactic templates* would provide a considerable improvement in the overall accuracy of the general clone modification evaluation of *CPC*. If a clone was never meant to copy the semantics of its origin, all potential modification warnings issued by *CPC* are by definition false positives. Unfortunately, a general solution of this classification problem seems to be unrealistic, at least in the short term.

Two more promising approaches might be to base the classification on specific examples of *syntactic templates* which could be either collected in experiments and observations or could be extracted from the copy and paste behaviour of the programmer. I.e. if a programmer repeatedly copies a specific, short source fragment from seemingly random locations¹, there might be a high likelihood that the fragment represents a *syntactic template*.

Overall the copy and paste practices are likely to differ considerably between programmers. Approaches which can be tailored to the practices of a specific developer might thus be the most promising solution in the long run. Employment of artificial intelligence based solutions might be an interesting option in this area.

4.2 Clone Similarity

One of the key questions when considering clone relations are the commonalities and differences between the clone instances of a clone group. The degree of similarity between two clone instances lies at the root of this problem.

Related Work

Roy and Cordy have tried to combine the large number of different classification approaches for the differences between two clone instances, which can be found in the clone research literature,

¹Simply the first location where the programmer found a very common snippet which was needed in a given situation.

into a common classification [51]. They define four basic types for the relation between two clone instances. The first three are based on textual similarity and the fourth on functional similarity.

Type I: Identical code fragments except for variations in whitespace, layout and comments.

Type II: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.

Type III: Code fragments with modified, added or removed statements in addition to variations in identifiers, literals, types, layout and comments.

Type IV: Code fragments which perform the same computation but are implemented through different syntactic variants.

Of these types, *Type I* and *Type II* are most likely to be of importance to *CPC*. As they represent situations which can be automatically detected and indicate that there is no ‘real’ difference between two given clones. Whether *CPC* should also ignore changes to literals and types is questionable though. Reliable detection of some cases of *Type IV* clones, while of less importance to *CPC* compared to static clone detection approaches, would also be beneficial.

Another interesting area are so called metrics based approaches. As they are inherently based on numeric representations of similarity, they seem to be predestined for the task. A potential problem might be the high degree of approximation in most approaches. In general, most static clone detectors make heavy use of different source code normalisation techniques. I.e. a common approach is to replace all identifiers² with a single placeholder, effectively ignoring all differences between identifiers.

Current Approach

Similar to the *CPC Classifier* module, the *CPC Similarity* module also delegates all processing to the registered strategies. A strategy can contribute pre-parsing and/or similarity evaluation functionality. The default *CPC* package ships with the following strategies.

Pre-Parser Strategies: A pre-parser evaluates the contents of the clones in question and may create a special normalised, temporary representation which can then be used by the similarity strategies to gauge clone similarity without interference of irrelevant details like whitespace changes.

Whitespace Normalisation: The first pre-parser will normalise whitespace in all clones, unless the content type has been specified as plain text. Changes in spaces, tabs or newlines are filtered out.

Java Tokeniser: If the clones contain *Java* code, they are ‘tokenised’ with a *Java* tokeniser. The results are then used to generate a normalised version of the clone contents. In this process all comments are filtered out. Changes which only affect comments are thus ignored during all following stages of the similarity evaluation.

²The same is often done for literals too.

Similarity Strategies: A similarity strategy tries to evaluate the similarity and differences between two clones. The overall result is built up incrementally by the joint work of all registered strategies.

Levenshtein Distance: The only currently included strategy is based on the Levenshtein distance, the number of simple, one character operations needed to transform one clone into the other. This metric thus mimics the way in which the programmer modifies source code and was also used as a similarity metric by other clone detection tools [23]. The Levenshtein distance is converted into a similarity percentage by evaluating it in relation to the length of the longer of the two clones.

CPC does not currently ship with a strategy which detects renamed identifiers. While it is relatively easy to ignore all identifiers during comparison or to replace each identifier with a special, unique placeholder, such a simple approach would not be enough. It is crucial to ensure that the renaming is truly semantically equivalent. Otherwise developers which rely on *CPC*'s assertions that a clone is synchronised with its clone group might overlook critical modifications³.

Future Improvements

Currently the *Similarity Provider* does only handle *Type I* clones completely. Support for renamed identifiers and thus coverage of the most important aspect of *Type II* clones would be the next important step. However, as was already outlined, the implementation of a corresponding strategy is not easy. It is important to keep in mind that there is no guarantee that the clone itself covers a complete construct of the programming language or that the file it is located in is syntactically correct. In this way the problem facing *CPC* differs from those faced by static clone detectors which can usually assume that the underlying source files are all syntactically correct. There may well be no viable approach which produces good results under these circumstances. A strategy implementation which first checks the syntactic correctness of the source file and only normalises renamed identifiers if there are no parsing errors might be the only realistic approach.

Once *Type II* clones are sufficiently covered, the next question would be if there are more appropriate algorithms which could be used for the similarity calculation. As this problem is very similar to those faced in the area of static clone detection, it might be very interesting to experiment with some of the techniques used in that field. Some studies have even applied *CCFinder*, a static clone detector which does not tolerate statement insertions or removals, to the contents of two clones and have used the amount of reported duplicated tokens as a measure of clone similarity [36, 40].

One potential approach would be to use a token based comparison instead of the current character based approach. Even combinations of both approaches might yield good results. It might be beneficial to treat multiple changes in different locations differently from one big modification in one location. One could also handle statement modifications, insertions and removals in different ways.

³A potential problem for consistent identifier renaming, which has been described as one of the most common update anomalies [30, 42].

Approaches of a completely different nature, which are also very prominent in the field of static clone detection, would be metrics or control flow based evaluations of similarities. Past research has produced a vast amount of findings in this area.

Another, *CPC* specific, possibility would be to leverage the fact that the complete modification histories of all clone entries are available. The knowledge about all past modifications and the detailed evolution of the clone since it was originally copied from its origin might prove to be helpful.

4.3 Clone Modification Notifications

The detection of potential, clone related update anomalies represents the main purpose of *CPC*. The corresponding heuristics are therefore of key importance for future versions of *CPC* and providing a base for such heuristics was thus a key requirement for this project.

The *CPC* notification framework is subdivided into multiple, loosely coupled parts on multiple levels, which enable contributors to provide replacements for parts of the implementation. It is furthermore completely separated from the remaining *CPC* components which also makes it possible to replace the *entire* notification framework with a completely different approach if desired. The individual components of the notification framework and their interactions were also covered in section 3.4.7.

Related Work

The available work which specifically covers the problem of update anomalies in relation to clone characteristics is very limited. However, a number of general considerations found in the research area of static clone detection may be helpful.

Some publications have noted that the proximity of clone instances has an effect on the likelihood of update anomalies [14, 51]. If all members of a clone group are located within one class there is a higher probability that they will always be modified consistently. Further distinctions made are clones within the same region⁴, clones in files within the same directory and clones in files in different directories.

The timing of modifications may also be relevant. Kim et al. and others argue that programmers tend to propagate modifications correctly most of the time [38, 51]. Programmers regularly use their own memories about copy and paste operations to guide their actions. Issuing modification notifications about a clone which was just recently created is therefore likely to be of little use.

4.3.1 Modification Evaluation

Once the contents of a clone are modified, the question arises whether these modifications should be propagated to other members of the clone group. The modifications might also be of no consequence or the clone instance might be evolving separately from the rest of its group.

⁴Within the same method or within the same code block within a method.

Current Approach

The current *Notification Evaluation Provider* is, like other providers, constructed in a very flexible and loosely coupled way. Each registered strategy is executed in order of its priority and the final evaluation result is based on the joint work of all strategies.

Each strategy can ‘vote’ for one or more evaluation results and can assign different weights to each of its ‘votes’. A strategy can also attach a human readable message to its decision which will be displayed to the user, if the strategy is not overruled by other strategies. The following listing contains some of the common evaluation results.

Ignore: The modification will be silently ignored.

Synchronised: The modification either returned all members of the clone group into a synchronised state or did not compromise an existing synchronisation. All members of the clone group will be marked as synchronised.

Modified: The modification was not significant enough to warrant a notification of the user. However, it nevertheless led to a non-synchronised state. All members of the clone group will be marked as modified.

Notification: The modification should be propagated to other members of the clone group. A notification for the modified clone will be generated and all other group members will be marked as modified⁵. A notification can be marked as either instant or delayed.

Warning: The modification was deemed to be especially critical and the developer should be notified more prominently about the problem. Otherwise this state is similar to ‘Notification’.

Leave Group: The modification may have changed the contents of the clone to a degree where it no longer has any significant commonalities with the other members of its clone group. In this case the clone should no longer be considered to be a part of the group. Depending on the configuration settings, such clones are either marked as orphans or are deleted.

The *Notification Evaluation Provider* currently ships with the following limited set of simple strategies. Strategies are executed in order of priority⁶ and each strategy can specify whether the processing should continue with the next strategy or whether it should be aborted.

Classification Check: First the classifications of the modified clone are evaluated. Clones which are classified as ‘template’ will be filtered out⁷. There is no need to notify the developer about such modifications as template clones have no semantic commonalities.

Whitespace only Modification: Additionally all modifications which exclusively consist of whitespace changes are also filtered out.

⁵Group members which already have a notification pending are ignored.

⁶The following listing is sorted by priority and represents the actual execution order.

⁷An ‘ignore’ result is returned and further processing of strategies is aborted.

Self Similarity: As the modification evaluation process is potentially expensive, this strategy checks whether the current clone content significantly differs from the clone content prior to the modification. This is done by calculating the similarity between the old and the new content by use of a *Similarity Provider*. If the similarity is still 100%, the modification is filtered out.

Depending on the capabilities and configuration of the *Similarity Provider* implementation, this will filter out modifications which only affected comments and modifications which modified the code in other, semantically irrelevant, ways (*i.e. consistently renamed local identifiers*).

Full Similarity: This strategy evaluates the similarity (*via a Similarity Provider*) between the content of the modified clone and the contents of all members of its clone group. Special emphasis is placed on the synchronisation state between the modified clone and its origin.

If the clone is fully synchronised with all of its group members the strategy returns a ‘synchronised’ result and prevents the processing of further strategies. If the group is not synchronised but the modified clone still matches its origin, a ‘modified’ result is returned. The handling of potential update anomalies in the cases where the origin does not match the modified clone is left to follow-up strategies.

Minimal Age: This strategy suppresses notifications for newly created, modified clones with an age below a specific threshold as the developer is likely to still remember the other clone instances [38, 51].

Clone Location: This strategy suppresses notifications for modified clones if all group members are located within the same file as these are less likely to result in update anomalies [14, 51].

Default Notify: By default the current *Notification Evaluation Provider* implementation would ‘ignore’ all clone modifications which are not handled by one of the registered strategies. This strategy is a fallback strategy which returns a ‘notify’ result with a generic message for every modified clone.

Future Improvements

Nowhere else is the need for improvement as visible as in the area of notification generation. Other, much needed improvements in the *Classification Provider* and *Similarity Provider* have the potential of greatly supporting the notification generation. The first three strategies described here will automatically benefit from most improvements in these providers. A strategy for removing highly modified clones from their clone groups is currently even suspended, as missing handling of identifier renaming by the *Similarity Provider* makes use of the similarity result for this purpose problematic. However, the key to better update anomaly notifications definitely lies in sophisticated notification strategies.

The most obvious improvement would be to leverage the available modification history of the clone instances. Modifications which are made right after the creation of a clone are likely to

represent the ‘parameterisation’ of the clone and do not constitute changes which need to be propagated to other clone instances. Such parameterised clones could also be highlighted differently as they are neither synchronised nor truly modified. The *CPC* framework already includes the necessary support and is currently only missing the corresponding notification strategy.

It might also be worthwhile to investigate the effects of other clone location relations besides the ‘same file’ approach taken so far. Aversano et al. argue that other aspects like the distance in the package or class hierarchy have a direct influence too [14]. They also argue that there are specific types of changes which usually don’t need to be propagated right away as a delay does not introduce risk. One of their examples are code restructuring efforts which do not directly affect the functionality of the code. It remains open whether some of these cases could be reliably detected in an automated way.

In the long term, it might be interesting to reconsider the appropriateness of the current ‘black list’ approach⁸. If it emerges that a number of specific cases are typical causes for update anomalies, strategies could specifically target these cases and the default action could be to ignore a modification. Realising this would be no problem with the current *CPC* notification framework.

Overall, really tangible improvements in this area are likely to require an insight into copy and paste activities and clone modifications which can only be obtained by evaluating considerable amounts of real world data about cloning activities.

4.3.2 Delayed Notifications

If the registered modification evaluation strategies come to the conclusion that a recent modification should be propagated to other members of the corresponding clone group it might be prudent to delay the notification of the developer until it is certain that he is unlikely to do these modifications on his own. Instant notifications, while possible⁹, may often be counter productive as past studies have shown that developers tend to maintain clone groups consistently [14, 38, 40, 51].

Current Approach

The current, rather simple *Notification Delay Provider* implementation enqueues notification events which were not marked as instant notifications and reevaluates them once one of the following conditions are met.

Timeout: A configurable amount of time has passed since the last modification of the clone.

File Closure: The file which contains the clone was closed by the programmer.

After the reevaluation was triggered, the reevaluation result of a notification event decides whether the event is displayed to the developer or whether it is silently dropped. The high inhomogeneity of the interface requirements of potential extensions made a strategy based approach problematic. Contributors are therefore encouraged to provide their own *Notification Delay Provider* implementations. The current implementation can be extended by a contributor and provides a good base to start from.

⁸The current strategies only filter out cases where a notification is likely to be unneeded.

⁹A modification evaluation strategy can indicate that a specific modification should trigger an instant warning.

Future Improvements

The best approach to delayed notifications remains unclear. If they are displayed in a very unintrusive way, as is currently the case, delaying of notifications might not even be required as any notification would automatically be removed once the developer propagates the modification to the remaining clone group members. However, the presentation of notifications to the user is not within the control of the *CPC Notification* module and UI contributors might opt for a more aggressive notification style.

A number of additional conditions seem to be interesting candidates for inclusion in the *Notification Delay Provider*. The currently visible area of the source code might provide an indication about the state of a modification. Once the developer starts to work in an area where neither the modified clone nor any of its group members are visible, the modification might be complete. It would furthermore be interesting to check whether any of the currently opened files contains group members of the modified clone or when the last modification to any group member has taken place.

Chapter 5

Challenges and Setbacks

This chapter describes some of the challenges faced during the course of this thesis and highlights areas which have proven to be much more work intensive than one might initially expect.

5.1 Planning and Risk Assessment

*The perfect project plan is possible
if one first documents a list of all the unknowns.*

[Bill Langley]

Right at the outset of this thesis, during the preliminary prototyping phase a large number of potential problems became apparent. These problems could be attributed mainly to two factors; *CPC*'s needs to interact with the Eclipse IDE in very uncommon ways and the general complexity of the Eclipse platform.

It is interesting to note that *all* of the problems faced during the course of this thesis were already identified in the initial topic proposal¹. However, even though these risks were known and were continuously tracked and reassessed during the course of this thesis, most could not be completely mitigated.

One of overarching challenges posed during this thesis was the futility of any long term planning. The size and complexity of the Eclipse API made it all but impossible to adequately estimate the work effort required for a particular task. Without in-depth knowledge of the corresponding APIs, it often remained highly uncertain whether hours or weeks would be needed for the implementation a specific feature.

In many cases the only way to achieve the required indepth knowledge was the implementation of a prototype which already encompassed most of the functionality of the final implementation. This resulted in a number of cases where the required effort remained uncertain until the implementation of the specific feature itself was almost completed. This also meant that for some design and implementation decisions two time consuming prototypes needed to be constructed before any conclusion could be reached.

¹Included in the *Appendix and Recommended Readings* booklet.

5.2 Reuse and Performance

Two of the initial design goals for *CPC* were the reuse of *ECG* functionality as well as highly decoupled operation from Eclipse. While both aspects are desirable, it later became apparent that they are not, or only partly, feasible due to their potentially negative effect on overall system performance.

Reusing the ECG Sensor

The existing *ECG Sensor*, which was developed as part of the *ECG Lab*, was able to provide information about a large set of activities inside the Eclipse IDE [52]. Among these activities were copy and paste actions by the developer as well as document modifications. While the sensor did not provide all the required data, it seemed like a good base to start from. Some of the additional event types required for *CPC* were also deemed to be useful for future experimental setups with the *ECG Lab*.

Thus one of the first steps during the early prototyping and initial implementation phases was a complete restructuring and modularisation of the existing *ECG Sensor* in order to allow its use by *CPC* and *ECG Lab*. Initially the tight coupling of the *ECG Sensor* to the *HackyStat Sensor*, on which it was based, obstructed the restructuring effort. Its core design only supported the communication of programmer activities via *SOAP* or *Java RMI*. And the only communication format, even for *Java RMI*, was *XML*.

The resulting, improved version of the *ECG Sensor* creates internal event objects instead of directly producing *XML* output. These events are then handled by an internal event dispatching registry which, depending on the registered listeners, dispatches the events via *SOAP*, *Java RMI* or directly as a callback to a registered listener object. Multiple listeners of different types are supported, allowing *CPC* and *ECG Lab* to use the same *ECG Sensor* instance. This fact also allowed *CPC* to optionally output *ECG Lab* style log files, if requested.

However, the *ECG Sensor* had some short comings which required further modifications to allow its use as part of *CPC*. The original version of the sensor did not provide detailed information about document modifications. Instead it simply resent the entire document after two seconds of inactivity. The *ECG Lab* extracted modification information from these document events by applying a line based *diff*-style algorithm. But the resulting approximation of the real document modification was too inaccurate for use in *CPC*. Thus new event types with fine grained modification information and better caching/purging strategies were added and some of the existing events were extended.

Furthermore a couple of additional improvements and bug fixes were introduced. Though a couple of problems could not be solved during the initial restructuring effort. The *ECG Sensor* depends on a *HackyStat* library which is no longer supported, contains known bugs and for which the source code is no longer available. This can result in index out of bounds and null pointer exceptions within the library which are very hard to work around in the *ECG Sensor* code.

The resulting version of the *ECG Sensor* was used by *CPC* for roughly two months before it had to be replaced for performance reasons.

Decoupled Operation

Decoupling as much of the *CPC* code as possible from the specifics of the current Eclipse API was one of the key design goals. The necessity for this becomes very apparent when looking at some of the older Eclipse plugins. While the Eclipse community tries to limit the number of API breaking changes between releases, it is very common for older, no longer maintained plugins to be incompatible with newer Eclipse versions. This especially affects plugins which use a lot of low level APIs like *CPC* or *CbR* [28].

While decoupling is desirable, it does impose potential performance penalties and may be impossible in some areas due to other constraints. The initial *CPC* design tried very hard to limit most of the interaction between Eclipse and *CPC* to the *ECG Sensor*. An approach which was later partly reconsidered due to its considerable performance impact (*see next section*).

Performance Impact

The initial prototyping phase and tests with early versions of *CPC* did not uncover any major performance issues. An *ECG Sensor* based architecture was thus adopted and it was believed that any potential future performance issues would be addressable by introducing additional optimisations and a degree of caching.

However, the Eclipse 3.3 Europa release² added a new in-place rename refactoring feature which ‘renames’ Java identifiers on the fly. During such a rename operation, all occurrences of an identifier within the current file are updated with each key press. The actual refactoring is still performed by the old rename refactoring backend. Basically the initial modifications to the editor are only a preview. Once the programmer confirms the rename operation, Eclipse undoes the modification of the document and then performs the normal refactoring operation.

For a large Java class with an often referenced member variable this could result in hundreds or even thousands of document updates for each key press of the developer. Each document update potentially moving or modifying existing clones within the file. In some artificial tests carried out it was easily possible to block Eclipse for seconds up to minutes on each key press if *CPC* was activated. Even without *CPC* this feature has been reported to be very slow in certain situations³.

A redesign of some of the *CPC* core components was thus required to provide adequate performance even for situations with very high rates of document updates. The strong modularisation of the *CPC* architecture meant that this switch affected only a very small part of the overall *CPC* code base. The resulting design was presented in chapter 3.

5.3 The Eclipse API

5.3.1 General Complexity and Documentation

While the Eclipse IDE undoubtedly represents a hallmark of modular design and extensibility, its complexity and sheer size pose serious challenges. With a total of over 17 million lines of code, the

²2007-06-27 — http://www.eclipse.org/org/press-release/20070627_europarelease.php

³Eclipse Bug: 185050 — <http://bugs.eclipse.org>

Eclipse IDE easily outclasses many major open source projects. Without good documentation an API of this size is very hard to use. In general the Eclipse API is documented on three levels:

1. All API interfaces and classes contain *JavaDoc* documentation and all extension points contain basic documentation as part of their schema definition.
2. The major APIs are described in the Eclipse online help [5]. In most cases special example applications demonstrating the correct use of these APIs can be found in the Eclipse source repository⁴.
3. A small selection of key API topics is presented in special articles on the Eclipse website [6].

As long as the problem to be solved follows one of the common extension use cases, the Eclipse documentation is very good. The online help provides a broad overview and examples, articles on the website provide step by step guides and exhaustive example applications can easily be obtained. Other articles improve the understanding of key areas and often cover even the small details which might prove to be potential pitfalls for new developers. Furthermore, a number of good books covering the internals of the Eclipse platform are available.

However, projects like *CPC* which venture off the beaten track are an entirely different story. In many cases parts of the required API are only covered by the *JavaDoc* documentation and even that may leave many questions unanswered. So far the Eclipse community has adopted the practice of specifying as little as possible in the API documentation to retain as much implementation freedom as possible. While this simplifies the development of the Eclipse workbench and reduces the risk of breaking specified API behaviour it may force plug-in developers to rely on aspects of the implementation which are not officially part of the API.

Another serious problem which arises in these situations is the discovery of API sections which can be used to address a specific requirement. The search for the right API to solve a given task may require reading large sections of Eclipse source code to determine which packages and classes take part in the processing of the data of interest, followed by examination of all API classes and interfaces found.

The situation is escalated even further by the fact that there are often multiple ways of achieving a certain goal but not all of them might be equally well suited for the task. This implies that even after finding an API which can be used to solve the issue at hand, there remains a possibility that a better API might have been overlooked.

This becomes even more probable due to the fact that the naming of an API package or interface is usually related to its primary use case. However, many parts of the API can be used for more than their main purpose. This results in a situation where simply looking at the package and class names of the Eclipse API is not enough.

Thus the question of whether there exists any API for a particular task and where to find it can easily become very time consuming to answer. And even once an answer is found, the confidence in its correctness is often limited.

⁴Eclipse CVS Repository: `:pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse`

5.3.2 Inconsistent, Inappropriate or Missing APIs

Unfortunately there are a number of cases where core *CPC* requirements are not well aligned with the existing Eclipse APIs. For some areas API coverage is simply non existent. For other parts an API is provided but does not behave consistently or is so ill matched to the task at hand that a lot of additional work is required for its use. This section provides a non exhaustive list of examples of such cases. Team provider related issues will be covered in section 5.4.

Rename Refactoring

Refactoring operations overall and rename refactorings specifically are of high importance to *CPC*. It is vital for *CPC* to be able to participate in all types of document modifications which could potentially move or modify clone instances. As was already partly covered in section 3.4.2, this participation is made possible by tracking low level file buffer changes inside the Eclipse workbench.

While this approach works very well for most refactoring operations and could in theory work for all such operations, it fails for rename refactorings. For some reason the current *JDT*⁵ *Java* class rename refactoring partly circumvents the Eclipse file buffer framework. All modifications to other affected classes⁶ are executed through the buffer framework as expected. However, for some reason modifications to the renamed class itself are done in a manner which circumvents the file buffer framework and thus prevents interested parties from being notified about these modifications.

As there is no specification which explicitly states that all modifications have to be made through the buffer framework, the observed behaviour does not constitute a failure. However, it clearly is an inconsistency which should be addressed. This matter was therefore filed as a bugreport by the author of this thesis⁷. Even though no alternative API or workaround exists, it remains unclear whether this issue will be addressed anytime soon.

Another important task relating to rename refactorings is the detection of rename and move operations and the correct updating of the corresponding clone data to reflect the change. While Eclipse offers a special notification API for file resource changes which also provides information about file moves, correct handling of rename and move operations is still problematic.

The high concurrency present within the Eclipse workbench can cause situations in which the different notifications required may reach an interested party out of order. I.e. the file open event for the new file location may be received before the file move notification.

There are furthermore situations in which other plug-ins can cause problems. The *Subversive SVN* provider, which is bound to become the official Eclipse *SVN* team provider in the future, hooks into the resource modification process in a way which prevents file resource change notifications from being generated correctly. If a file is under supervision of this team provider, rename and move operations will be reported as deletions and additions, without any context information which could be used to identify the underlying file move. A bugreport for this behaviour was also filed by the author⁸.

⁵Java Development Toolkit

⁶Renaming all occurrences of the refactored class name in classes where it is referenced.

⁷Eclipse Bug: 213984 — <http://bugs.eclipse.org>

⁸Eclipse Bug: 213991 — <http://bugs.eclipse.org>

Multiple steps were taken to support file move and rename operations in *CPC* despite these problems. The concurrency issue is bound to remain, even if the two ‘defects’ mentioned above are fixed in a newer version of Eclipse. *CPC* addresses this by checking for file location changes in multiple locations and by keeping track of already processed changes. This way it does not matter whether the information about the location change reaches *CPC* first as a file move notification, a file open event with the new location or another event. It will be correctly processed.

A workaround for the fact that the rename refactoring is partly bypassing the file buffer framework was also implemented. These situations are automatically detected once the file is first opened in the new location. The potentially out of sync clone data is then handed to the current default *Reconciliation Provider* for reconciliation. As file rename or move operations typically only change the package and/or class name the reconciliation should in most cases be successful without any loss of clone data.

Special handling for the missing file move notification for files under *Subversive SVN* control is currently not implemented as the required information is not easy to obtain at the point in time where these notifications are received. Instead these situations will currently cause a graceful fallback which will invalidate all clone data for the affected file to prevent inconsistencies. Use of a rename refactoring within *Subversive SVN* projects will thus lead to controlled *clone data loss* until the corresponding defect in the team provider is fixed.

Marker Support

The Eclipse workbench provides support for persistent annotations on file system resources and on sections within files. In theory, these so called **Markers** are exactly what *CPC* would need to store the clone data for each file. However, the initial evaluation uncovered a number of serious problems of the current marker implementation.

First and foremost is the fact that the exact behaviour of markers is neither specified nor documented. It is thus impossible to make any decisions based on API specifications. Instead examination of relevant sections of the Eclipse source code and experimentation are the only means of building grounded assumptions about their expected behaviour. A time consuming, potentially error prone process which provides no guarantee that the observed behaviour will remain unchanged in future Eclipse versions.

Another aspect is reliability. Inside the Eclipse workbench markers are used primarily for storage of position data which can easily be recomputed. I.e. compiler warning and error markers which are dropped and recreated on each compilation. This typical use case of using markers mainly for ‘caching’ purposes puts only a low priority on the long term correctness of marker position updates.

‘There’s no built-in concept of markers always being up to date: anyone can change a file and there’s no guarantee that this client correctly updates the attached markers.’⁹

The results of the missing emphasis on reliability can be observed in many areas.

⁹Daniel Megert <daniel_megert@ch.ibm.com>, official maintainer of Eclipse marker support, *Eclipse platform newsgroup*, in reply to an inquiry by the author

A large number of open bug reports for Eclipse marker support can be found on the bug tracker. Some entries dating back to 2001. The remainder of this section will highlight some of the issues which would prove to be problematic for *CPC*.

The Eclipse markers framework has no notion of external modifications to a file. This will lead to corrupted and lost markers if a resource is modified outside of Eclipse¹⁰. Even modifications inside the Eclipse workbench can be problematic if a plug-in directly accesses the underlying file.

Furthermore, the control over marker position updates is limited. Positions are updated by the default Eclipse position updater and customising the position updating strategy would be problematic. In certain situations positions may be updated incorrectly too¹¹.

Markers are also not part of the Eclipse undo/redo framework and undo/redo actions by the programmer can thus lead to the loss of marker position data. While this is also true for the current *CPC* implementation, a marker based approach would be fundamentally unable to support undo/redo, whereas the current *CPC* implementation could in theory support undo/redo (*see next section*).

It also remains unclear how well a marker based approach would scale as *CPC* clone data entries are likely to considerably outnumber the common Eclipse markers. The existing marker APIs do not allow marker lookups over the entire project at an acceptable cost. As such even with a marker based approach some additional backend storage infrastructure would be needed to support fast global access to clone related data.

All in all the current marker implementation was deemed to be unfit for use in *CPC*. In order to make it suitable for the requirements of *CPC*, extensive modifications to the current implementation, affecting a number of low level Eclipse components, would be required. The work effort entailed in designing a patch which addresses these issues and the expected duration of the acceptance process made any marker based approach infeasible.

However, in the long run, once these problems have been addressed, such a marker based approach to clone tracking would seem highly attractive. The author is convinced that, should clone tracking functionalities ever be introduced as part of the official Eclipse distribution, they would most likely be marker based.

Undo and Redo

The Eclipse API provides an extensive undo/redo framework which tries to encompass all activities of a programmer. Plug-in contributors are encouraged to use these APIs to provide a seamless undo/redo experience and it is considered bad practice to circumvent them. However, maintaining undo/redo stacks adds additional overhead to operations. For very common, small operations the overhead can exceed the cost of the operation itself.

As a result, the Eclipse workbench does not maintain undo/redo information for document markers and position updates. These areas have been identified as being especially performance critical due to their typically very high update rates. Position objects are even updated using direct access to the member values, circumventing the getters and setters, to increase performance.

¹⁰Eclipse Bug: 35696, 3551 — <http://bugs.eclipse.org>

¹¹Eclipse Bug: 186118 — <http://bugs.eclipse.org>

CPC is based on the Eclipse document position APIs and all modifications made to positions are thus not reflected in the undo/redo stack. Experimental prototypes have shown that undo/redo support would be possible. However, the lack of applicable Eclipse undo/redo APIs implies that such support would require a custom undo/redo implementation which would have to rely on certain aspects of the Eclipse position and document update handling which are not part of the API specification and documentation. Furthermore, the entailed performance overhead would be potentially considerable.

The current implementation thus does *not* provide explicit undo/redo support. However, as undo/redo actions produce normal document update events, their use will only lead to clone data loss in very specific situations. One example would be the deletion of a section in a source file which encloses a clone. The deletion will remove the clone data for the enclosed clone and a potentially following undo action will not restore it.

Concurrency and Document Locking

Another area of concern is the high concurrency within the Eclipse platform and the large number of potential points of lock contention. More than 30 active threads at any given time are a common situation. Once a plug-in contributes functionality which requires its own internal locking, extreme care needs to be taken to avoid potential dead locks.

It is crucial to ensure that all parties acquire locks in the same order. This is made more complicated by the fact that some listener APIs do not clearly specify which locks the caller might be holding. The reliance on the **synchronized** locking scheme of *Java* in certain areas, furthermore makes it impossible to partly circumvent this problem by acquiring multiple locks in one atomic operation.

The Eclipse platform addresses this issue by introducing the concept of a *Job*. Jobs are bundles of work which can be scheduled with a job execution framework for asynchronous execution. Each job specifies its locking requirements as a **scheduling rule** and the job execution framework takes care of the correct ordering and scheduling of all waiting jobs. A job will automatically be queued until all of its resource requirements can be met. The strict limitation to upfront lock acquisition ensures that no deadlocks can occur.

Unfortunately, such an asynchronous approach is not suitable for many areas of *CPC* where access to synchronised resources is needed directly and where the result of such operations has immediate effects on further processing within that thread. Due to the large number of listener callbacks which provide *CPC* with detailed information about the current state of the Eclipse IDE, the same section of *CPC* code could be executed within a variety of different threads which might potentially be holding a number of locks.

The main UI thread is another typical point of lock contention as access to *SWT* UI elements is restricted exclusively to the UI thread. Any background process which wishes to display data to the user in a synchronous fashion thus has to take extreme care. Any lock it holds could potentially cause a dead lock scenario. The same applies to code sections which are already holding some kind of lock and are calling Eclipse API methods as some of these may internally acquire specific locks.

Situations where the caller of an API listener is always holding a specific lock can furthermore

result in another curious situation, if the called listener method also requires some locks. As the locking order can not be changed in this scenario, *all* other places need to be modified to match this particular locking order. In *CPC* this resulted in one case where the lock on a document needs to be acquired before a lock on a document registry which is needed to retrieve the document.

The locking problem has also increased the complexity of some parts of *CPC*. In very rare situations it may happen that a required resource can not be acquired without causing a deadlock. The current *CPC* implementation handles such special cases by enqueueing information about all operations and automatically ‘replaying’ them, once the resource becomes available again. I.e. the processing of document modifications happens synchronously and requires the availability of the clone data for the modified document. If the clone data can not be obtained without potentially causing a dead lock, all modifications are enqueued until the clone data becomes available.

Minor Issues

During the course of this project, a number of smaller issues surfaced. While these are not critical, they may still represent worthwhile candidates for future Eclipse API improvements.

Aside of the general file buffer related problems which were already addressed in earlier sections, a somewhat peculiar fact is that Eclipse offers no straight forward means of receiving notifications about the execution of a revert operation. While monitoring of the file modification date and some file buffer API methods yields a very good approximation, there may still be cases where such a makeshift approach breaks. A clear distinction between save and revert actions within the file buffer framework would thus be very helpful.

With its strong emphasis on tight integration and interoperability of contributions, the very limited data persistence services offered by the Eclipse Platform simply don’t fit. As a result every somewhat sophisticated plug-in, which is not solely file based, has its own persistence approach.

5.3.3 Conservative Development

Public APIs, like diamonds, are forever.

You have one chance to get it right so give it your best.

[Joshua Bloch]

The overall style of development in the Eclipse community is rather conservative. This fact has already been partly addressed in multiple other sections within this thesis. It is thus covered only very shortly here.

When looking at the Eclipse bug tracker¹² a bit more indepth, the first thing that strikes the eye are a number of very old and still pending bug reports and feature requests. These fit well with the author’s general impression to date. As is to be expected, the time required for a bug fix or feature request to become part of the next release is in most cases closely related to the controversiality of the change.

If a submitted patch addresses an issue which is undisputedly a defect and if the suggested fix is the only viable solution, patches are usually integrated into the upstream repository within a

¹²<http://bugs.eclipse.org>

very short time frame. Often hours rather than days. However, due to the adopted release cycle it may still take more than half a year for a fix to appear in a stable release.

If, on the other hand, the issue addressed is not clearly a defect or if there are multiple possible approaches to fixing an acknowledged issue the adoption of a final fix is likely to take much longer. For cases which affect the API, even in non-breaking ways, discussions can stretch over multiple releases and can extend over years. There are even cases where official Eclipse projects opted for permanently dropping functionality which required internal API access rather than waiting for the required extension of the official Eclipse API¹³.

Another aspect responsible for the slow adoption of some of the, often very reasonable, suggestions found on the bug tracker is a general shortage of staff. While Eclipse is an open source project, a considerable part of the development still seems to be done by IBM employees. It is quite common for acknowledged feature requests to be postponed to the next release multiple times and to finally end up being delayed indefinitely due to a lack of ‘cycles’.

5.4 Team Providers

From the outset of this project it was clearly evident that the remote synchronisation aspects of *CPC* would pose one of the biggest technical challenges. Experimentation with initial prototypes quickly showed that the available Eclipse and team provider APIs were ill suited for the requirements of *CPC*. This section highlights the corresponding key problems and limitations.

Team Operation Listeners — The History

One of the basic requirements for *CPC* support of distributed development teams and source repositories is a way to detect repository operations like commits, updates, merges, checkouts and others. While a listener API for such events might seem straight forward, there is currently no way to obtain such notifications in Eclipse.

*‘There currently isn’t any repository independent API for commit/update/merge notification and there is no way to do what you are asking with the CVS client.’*¹⁴

This short coming was first reported in 2002 and support for team provider operation notifications has since been requested again and again¹⁵. The same holds for the *Subversive SVN* team provider which has recently obtained the status of an Eclipse project in incubation and is poised to become the official Eclipse *SVN* team provider in the future.

*‘At the current moment Subversive does not provide corresponding interfaces.’*¹⁶

¹³TPTP banned internal API access: http://wiki.eclipse.org/index.php/TPTP_Jan_07_face_to_face

¹⁴Michael Valenta <michael_valenta@ca.ibm.com>, official maintainer of Eclipse CVS team provider, *Eclipse platform newsgroup*, in reply to an inquiry by the author

¹⁵Eclipse Bug: 24882, 26634, 40623, 44923, 78133 — <http://bugs.eclipse.org>

¹⁶Alexander Gurov <alexander.gurov@polarion.org>, official maintainer of the *Subversive SVN* team provider, *Subversive forum* — <http://forums.polarion.org/viewtopic.php?t=2661>, in reply to an inquiry by the author

One interesting approach, which would also benefit a large number of other users of the team provider APIs, would thus be the development of such a notification API. The author has submitted an API proposal¹⁷ and has spent some time investigating potential implementation paths.

However, in light of the complexity of the Eclipse team provider framework and the currently ongoing transition of the *Subversive SVN* team provider into this framework, the required effort was estimated to be considerable. The calls for a notification API which have remained unanswered for almost five years, furthermore fuelled doubts about the interest of the team support maintainers. The risk of spending weeks on the development of a notification API patch to finally see it denied was perceived as being considerable.

After much consideration, it was decided that the only viable approach which could potentially provide a working solution within the timeframe of this thesis was a custom sensor for repository provider operations which does not require any modifications of the Eclipse code base.

By its very nature such a sensor is team provider specific and is likely to rely on internal, non-API aspects of the team provider and potentially the Eclipse platform. The decision for this approach thus made it necessary to focus the development effort on a specific team provider. After an initial evaluation, the Eclipse internal *CVS* team provider was chosen as the primary target as it represents the only team provider which implements all Eclipse team APIs. Strict modularisation was applied to ensure that additional sensors for other team providers could easily be added.

Team Operation Listeners — The Implementation

The *CPC Team Sensor* was already briefly described in section 3.4.9. It tries to detect repository operations of the Eclipse *CVS* team provider by listening for callbacks via multiple other related APIs and by inspecting the internal state of the system. The Eclipse platform provides two APIs which can be exploited to this end, *Subscriber Change Listeners* and *Job Change Listeners*.

A *Subscriber Change Listener* provides Eclipse UI elements with team provider specific resource decoration information such as the current revision identifier of the resource and a number of status flags. This fact can be used to detect potential team provider actions as a commit or update operation always modifies the revision identifier of the corresponding file. Unfortunately, Eclipse generates these *Subscriber Change Events* at more or less arbitrary times. By keeping its own storage of revision identifier data for each resource¹⁸, the *CPC Team Sensor* can filter out all *Subscriber Change Events* which did not change the revision identifier of the underlying resource.

However, for *CPC* it is crucial to distinguish between commit and update operations which is impossible with just the data available in *Subscriber Change Events*. The *CPC Team Sensor* therefore also registers a *Job Change Listener*. *Job Change Listeners* provide information about scheduled and currently running background tasks within the Eclipse platform. By making use of the *unspecified* fact that the *CVS* team provider uses background *Jobs* to execute its operations, the *CPC Team Sensor* can gain additional insight into the current team provider state. Unfortunately, the corresponding *Job Change Events* do not provide enough information to reliably identify and distinguish the different *CVS* team provider operations. The *CPC Team Sensor* therefore has to

¹⁷Part of the *Appendix and Recommended Readings* booklet and submitted under Eclipse Bug: 78133

¹⁸In Eclipse arbitrary string data can be ‘attached’ to a resource by means of so called *persistent properties*.

fall back to the *Java Reflection* API in order to extract additional information from private fields and classes.

Overall, the *CPC Team Sensor* relies heavily on a number of implementation details of the *CVS* team provider which are *not* part of its specification. This leads to a number of problems. In the long term, any new Eclipse platform or *CVS* team provider version might change some crucial part of the implementation which could break the *CPC Team Sensor*. While this might be an acceptable limitation in light of the lack of viable alternatives, the following short term problem is much more significant.

As the internal behaviour of the *CVS* team provider and its use of background *Jobs* is not specified, experimentation and source code reviews are the only means by which the required understanding for the internals can be gained. This is considerably complicated by the fact that different ways of executing team operations often result in a different internal processing and that the timing of concurrently executed operations may change. I.e. it makes a difference whether an update operation is started from the context menu of a resource or from the synchronisation view. The resulting heterogeneity makes it all but impossible to ensure that all potential paths are correctly handled by the *CPC Team Sensor*.

Furthermore, the work on a *CPC Team Sensor* implementation for the *Subversive SVN* team provider has been stalled because the team provider does not implement one of the Eclipse team APIs correctly. A simple yet crucial method of the API currently only contains an empty body and an autogenerated ‘todo/stub’ comment. The author has reported this fact on the *Subversive SVN* forum and the Eclipse bug tracker¹⁹. So far the issue remains unsolved.

Logical Model Integration

The *CPC Store Remote LMI* module which was covered in section 3.4.9 also relies heavily on the *CPC Team Sensor* for notifications about team provider operations. Additionally there are a number of other *LMI* API related issues worth noting.

The default *CVS* team provider configuration explicitly disables *LMI* API support. Any *LMI* API based approach thus needs to inform the user about the required configuration settings. This is especially critical as incorrect settings could lead to *CPC* clone data loss. The corresponding configuration setting is furthermore not ‘side effect’ free as enabling *LMI* API support also requires the selection of a specific, non-default *CVS* team provider stack²⁰ which handles team operations and especially merge conflicts very different from the default stack.

Another usability problem is the default commit dialog of the *CVS* team provider. If any plug-in contributes *LMI* models which add files to the scope of a repository operation, the user will always be presented with a special confirmation dialog which lists all the resources which are to be included in the operation. This additional dialog can not be disabled and it furthermore contains a small check box at the bottom which indicates whether the changes should be displayed in the synchronisation view. By default this option is checked as soon as any *LMI* model contribution is

¹⁹Eclipse Bug: 211251 — <http://bugs.eclipse.org>

²⁰The *CVS* team provider offers two modes of operation, a compatibility mode with the *CVS* command line utility and an Eclipse specific mode. The compatibility mode is the default.

present and the state of the checkbox is not persisted. This requires a developer who is used to the default *CVS* team provider behaviour to always disable the corresponding checkbox on each team operation in order to retain the old behaviour. Both aspects are likely to be irritating for users of *CPC*.

Of real concern are two other aspects of the current *LMI* API specification itself and their implementation within the Eclipse platform. The basic design of the *LMI* API assumes that all files of one model are always under the control of the same model provider. The approach taken by *CPC* to combine a *CPC* clone data file and a *Java* source file in one *LMI* mapping does not fit this assumption as the *JDT* plug-in provides special merge handling semantics for *Java* source files. When contributing a *LMI* model, a plug-in can specify that its model contribution should be given a higher priority than specific other *LMI* models. This enables a plug-in to ensure that its model contributions are executed before specific contributions of other plug-ins. However, it is not possible to specify the need to be executed *after* another model and it is also not possible to delegate the handling of a specific file to another model.

This poses a serious problem for the *CPC Store Remote LMI* module in the case of a merge operation. The clone data for the merged file can only be calculated *after* the merge of the source file was executed. The *LMI* API does not allow *CPC* to specify its need to be executed after the *JDT* plug-in's *Java* merge handling. And the other option, to handle the *CPC* model first, would require *CPC* to handle the source file merging. An approach which would violate one of the key requirements of *CPC* as the presence of *CPC* could then affect the merge results of source files.

This was addressed by storing clone data in a temporary directory and processing it once the team operation was completed. While this approach works, it increases the complexity of the solution considerably and may also slow down team operations as some of the required data may need to be retrieved from the repository multiple times. Together with the fact that the contributed *LMI* model code is never executed if a simple update does not produce any conflict, this represents the main reason for the critical dependency of the *CPC Store Remote LMI* module on the *CPC Team Sensor*.

Another point of concern is the separation of headless and UI based *LMI* model operations practised by the Eclipse platform. While a *LMI* model contribution can enforce that specific sets of files may only be checked in and checked out together, this restriction does not hold for actions issued from within the synchronisation view. Instead a *LMI* model contributor has to provide new UI elements for the synchronisation view which enforce these restrictions. While this approach leaves plug-in authors a large degree of freedom, it is seriously encumbered by the *LMI* APIs underlying assumption that all parts of a model originate from the same contributor. There is no straight forward way of enforcing the combined handling of *CPC* data files and *Java* source files without circumventing the *Java* merging for the *JDT* plug-in in the process.

Current Status

All in all two serious problems with both remote synchronisation approaches of *CPC* remain. The inherent low reliability of the *CPC Team Sensor* and the limitations of the Eclipse *LMI* APIs. As a result the current remote synchronisation functionality of *CPC* should be considered as a prototype

implementation and remote synchronisation support is thus currently *not* shipped with *CPC*.

Future work in this area would be additional, extensive testing of the *CPC Team Sensor* and its extension to team operation execution scenarios which might currently not be handled correctly. However, the much more sensible long term approach, would be to further push for the introduction of an official Eclipse team operation listener API²¹. While the *LMI* model execution order problem can be circumvented by delayed processing and heavy reliance on the *CPC Team Sensor*, there is currently no solution which would address the problem of team operations issued from the synchronisation view not honouring model constraints. A reliable *LMI* based remote synchronisation approach might require changes to the Eclipse *LMI* APIs. The author will continue to pursue these issues.

5.5 Failures, Defects and Solutions

*There are two ways to write error-free programs;
only the third works.*

[Alan J. Perlis]

During the course of this thesis a number of Eclipse failures were observed and some defects could be located within the Eclipse source code. Some of these were already covered in other sections. For the sake of brevity, this part thus only addresses the remaining failures and defects.

While investigating potential approaches for undo/redo support in *CPC*, it became apparent that the Eclipse undo/redo framework did not generate redo events correctly in certain situations. The defect was located within the Eclipse source and a patch was submitted²². The issue has now been fixed in the upstream repository and the fix will be part of Eclipse 3.4.

Furthermore, a large number of seemingly random crashes of Eclipse text editors during editing of large text files was observed. The issue, though happening frequently, proved hard to reproduce. After a lengthy exchange on the bug tracker, the author submitted a JUnit test which managed to reproduce the problem with a high probability and provided further details about the nature of the problem²³. The issue is now fixed in the upstream repository.

Twice, extended amounts of time were wasted on the investigation of strange crashes and misbehaviours of *CPC*. In both cases the issues turned out to be entirely Eclipse related.

During the initial prototyping phases strange problems with the copy action became apparent. While working without problems under *Windows*, use of a copy hotkey sometimes had no effect under *Linux*. It finally emerged that these problems were caused by an incompatibility between the Eclipse IDE and the *Linux* clipboard management application *Klipper*²⁴.

Another time consuming problem resulted in seemingly random out of memory crashes of the entire Eclipse IDE. Exactly one day after a large and complex modification to the *CPC* core internals. Multiple days of debugging and testing effort, aimed at *CPC*, remained fruitless. It later emerged that an automatic update feature of the Eclipse IDE had updated the Eclipse installation

²¹See also: corresponding API proposal in the *Appendix and Recommended Readings* booklet.

²²Eclipse Bug: 206305 — <http://bugs.eclipse.org>

²³Eclipse Bug: 150934 — <http://bugs.eclipse.org>

²⁴Eclipse Bug: 153809 — <http://bugs.eclipse.org>

from version 3.3 to version 3.3.1. Unfortunately, 3.3.1 had a crucial defect which prevented Eclipse from setting the memory limits correctly at startup²⁵. The later released hotfix version 3.3.1.1 solved the problem, after all the debugging and testing effort had been wasted.

²⁵Eclipse Bug: 203325, 206775 — <http://bugs.eclipse.org>

Chapter 6

Testing and Evaluation

6.1 Testing

Any sufficiently advanced bug is indistinguishable from a feature.

[Bruce Brown]

In order to support the testing and debugging effort of the *CPC* framework a number of best practices have been adopted during the development of *CPC*. This section tries to provide an overview about the main points.

6.1.1 Testing and Debugging Support

Logging

Error and debug logging plays a crucial supporting role within the *CPC* framework. Its contribution is twofold.

First, during the exploratory prototyping phases it provided much needed support for the understanding of the internal workings of the Eclipse IDE. While the use of a debugger is possible, single stepping through large amounts of code represents a very inefficient way of promoting comprehension of the interactions between multiple components in a complex and highly concurrent environment like the Eclipse platform. In such an environment, the concurrency aspects are bound to especially pose a problem.

And second, logging provides much needed information which can help in locating a defect once a failure of *CPC* was observed. This is especially crucial as some failures may be very hard to reproduce and may only occur very infrequently. Care has been taken to include all potentially important state information in the debug and trace logging messages. A *CPC* trace log file contains all the information required to precisely reconstruct the events and interactions which led to a specific situation and can thus provide the same insights as a debugger session in almost all cases. This is especially important as *CPC* failures within a production environment may not be reproducible without access to the entire workspace and clone data of the developer. However, in a commercial environment these are likely to be confidential.

The *CPC* codebase is largely kept free of dependencies on any particular logging implementation by thorough use of the *Apache Commons Logging* framework [1]. A special *CPC* logging module takes care of most of the logging internals and also provides further logging related services to the rest of the *CPC* framework. The current implementation is based on *log4j* [2].

A comprehensive configuration file allows the behaviour of the *CPC* logging subsystem to be modified as needed¹. The underlying *log4j* implementation permits the realisation of even unconventional logging setups.

The degree to which logging penetrates all layers of the *CPC* framework is best emphasised with a number. The current implementation contains around three thousand logging statements² and can, with full trace logging, produce very large amounts of logging output in a short amount of time.

API Pre-/Post-Condition and Integrity Checking

In line with good programming practices and API guidelines, the *CPC* framework uses a large number of assertion checks to ensure that pre- and postconditions defined in the *CPC* API specifications hold at all times. The resulting seven hundred assertion checks³ enable the current implementation to fail fast in many typical cases of incorrect API usage.

The *CPC* framework furthermore includes a large number of internal integrity checks which can detect a large variety of potential error states. In many cases these integrity checks can successfully prevent corrupted data from spreading between *CPC* components or between processing stages of one component. This approach has proven to be invaluable in supporting the testing and debugging efforts as it ensures that most internal failures are detected at an early stage and do not propagate through multiple layers of processing first, before they cause a visible failure.

Due to their potential, negative impact on overall system performance the processing intensive integrity checks can be switched on and off via a central configuration option. By default they are switched off.

6.1.2 Unit Testing

In addition to extensive manual testing a number of automated *JUnit* unit tests have been implemented. As the implementation of unit tests for GUI based applications is a very time consuming task, the focus was placed on the development of automated tests for components which are either critical or for which manual testing is very tedious. *CPC* currently includes 53 unit tests with a total runtime of roughly 17 minutes.

Most of the unit tests simulate actions of a developer and verify their effects on the internal clone data. A typical example is the simulation of a user who opens a *Java* file and starts to make a number of modifications to the document while at the same time issuing copy and paste actions every now and then. The unit test verifies the correct creation, modification and position updating

¹All logging aspects can be fine tuned. *CPC* currently ships with a 300 line logging configuration file.

²Each statement potentially producing one or multiple lines of log output.

³The number of `assert` statements in the code base. One statement typically covers multiple checks.

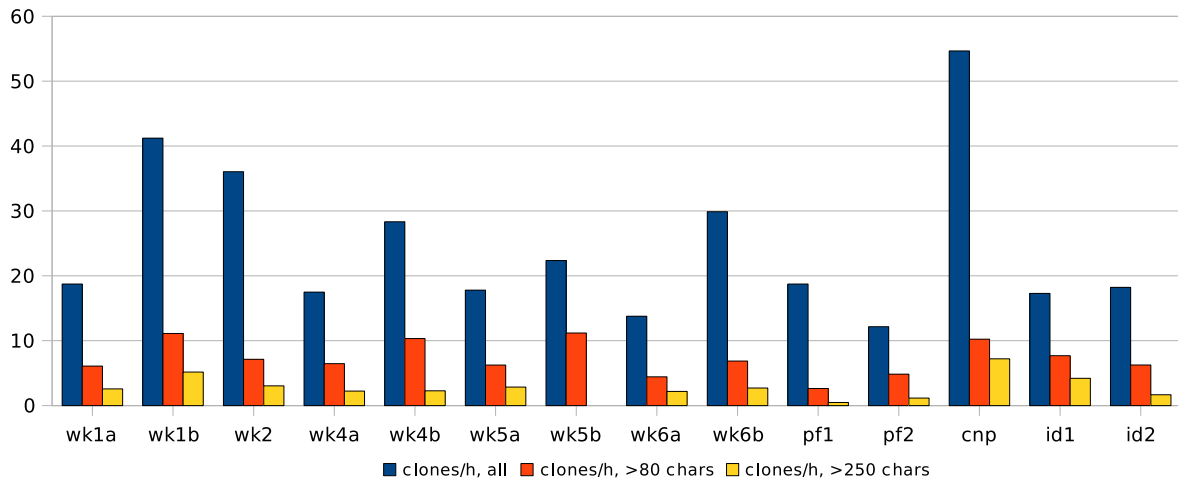


Figure 6.1: Comparison of average clone creation rate per hour

of all clone entries after each simulated user action. Some of the other user simulated actions are external file modifications, refactoring operations and source code reformatting.

The large number of internal integrity checks mentioned in section 6.1.1 furthermore allows the use of random data for some tests. While the correct position data is not hard coded in these cases, these tests still offer the possibility of uncovering internal defects which lead to a data corruption that can be detected by one of the internal integrity checks.

A number of tests also target critical, internal APIs like the *Store Provider* directly. The test suite also includes a few load tests.

6.2 Evaluation

*Good judgment comes from experience,
and experience comes from bad judgment.*

[Frederick P. Brooks]

6.2.1 Survey of Existing Data

To increase the general understanding of copy and paste practices of different developers across different projects and tasks, some of the data collected in earlier experiments and case studies was analysed anew with an emphasis on copy and paste actions. The main focus was placed on the frequency of cloning activities and the size distribution of typical clones in order to obtain additional support for the claim of Kim et al. concerning their observed high rates of large copy and paste clones [38].

This yielded 8,004 copy and paste actions by 15 developers, creating 4,363 clones over a total of 247 programming hours⁴. The following section gives a short description of the basic settings and highlights some of the interesting similarities and differences.

⁴Working time is measured from the first to the last copy and paste operation within a programming session. Breaks with a duration of one hour or longer are subtracted.

Tapestry Workshop (*wk*)

In 2007 an experiment aimed at investigating the differences between *pair programming* and *side by side programming* required five programming teams of two students each to solve a specific web-development task [54]. The five teams were given a three day introduction into the *Tapestry* web-application framework prior to the experiment which lasted one day.

The experiment consisted of a morning and an afternoon session. During the morning session the participants were allowed to work in either a *pair programming* or a *side by side programming* mode and could freely switch between programming modes. During the afternoon session all teams were limited to one computer per team and had to adopt a *pair programming* approach.

It is important to note that this setup makes it impossible to distinguish between the two developers in each team as they could freely change computers and spent prolonged periods of time in front of the same computer. The members of team 2 even spent all of their time in the *pair programming* mode.

During the experiment the teams had to implement a new database entity and a number of additional functionalities for an existing web-application which each team had implemented during the initial three day workshop. The new requirements were very similar to other problems which were already covered during the workshop. The experimental setup thus strongly encouraged copying code segments from the existing parts of the web-application.

On average the participants created 25.07 new clones per hour. The majority of these clones were very short and were unlikely to introduce any adverse effects. However, among these clones were 7.76 clones per hour which were larger than 80 characters and 2.56 clones per hour which exceeded 250 characters in size (*median: 22.36, 6.86 and 2.57 respectively*). This is also shown in figure 6.1.

Plat_Forms (*pf*)

In 2007 the Freie Universität Berlin hosted an international programming contest aimed at comparing different technological platforms for developing web-based applications. Nine professional teams of three developers each had 30 hours to complete a specific task. Each team was free to use its preferred programming language and tools to solve the task [48, 49].

Two members of one of the *Perl* teams used an instrumented version of the Eclipse IDE which logged all programmer actions. The 1,033 copy and paste actions recorded during the 19 hours of development yielded a total of 587 clones.

It is interesting to note that, while this observation is the only one which was not based on *Java* programming, the average cloning rates and the clone size distribution are still very similar to those obtained in other observations.

ECG Clone Tracking Evaluation (*cnp*)

Prior copy and paste related work at the Freie Universität Berlin provided an additional data set [46]. As part of an older, copy and paste related thesis a developer was observed for five hours

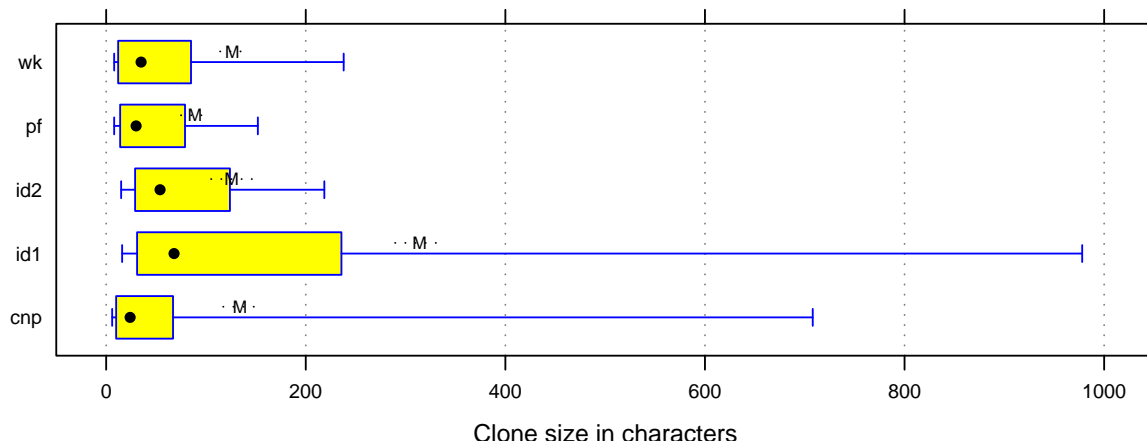


Figure 6.2: Comparison of clone size distribution

while trying to solve the *Rubik's Cube Solver* problem, as specified for the *ACM South Central Region 2002 Programming Contest*⁵.

The task was specifically selected because it was believed that the structure of the problem would result in a large number of copy and paste actions. Knowledge about the copy and paste related nature of the experiment is another factor which may have increased the number of copy and paste operations by the observed programmer.

This experiment displayed the highest cloning rate observed so far. The developer executed 326 copy and paste actions which created a total of 235 clones. This translates to 54.65 new clones per hour of which 10.23 and 7.21 per hour were larger than 80 or 250 characters.

Individual Developers (*id1/id2*)

The three experimental setups outlined above have a potentially crucial short coming. Considering the artificial nature of the tasks and their limited scope, it seems plausible that the copy and paste behaviour of the developers might have been affected by the experiments. Especially the tight time constraints and the prospect of not having to maintain the created software afterwards could entice programmers to adopt a more cloning intensive development style than they normally would.

To address this limitation, two additional programmers were observed during their normal programming work in their usual work environment. This resulted in 5,067 copy and paste actions and 2,688 clones over a total working time of 153 hours.

Results of Interest

While the small sample of only two developers does not allow any general conclusions, the results do at least provide some support for our doubts about the validity of copy and paste related results obtained from short, artificial experimental setups. Figure 6.2 displays the clone size distribution for the three experiments and the two observed developers. There seems to be a notable shift

⁵<http://acm2002.csc.lsu.edu/problems/>

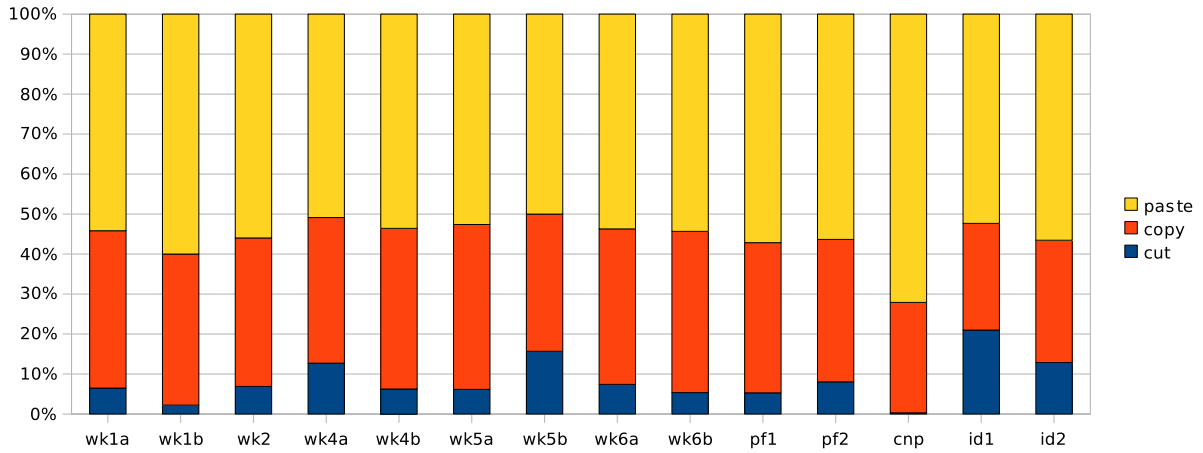


Figure 6.3: Comparison of cut, copy and paste event distribution

towards larger clone sizes when looking at long term programming efforts on normal projects, compared to the three experiments.

During these experiments, the clone size distribution was fairly similar with a large number of small clones. Between 63 and 69 percent of all clones were smaller than 50 characters. However, only 41 to 47 percent of the clones created by the two observed programmers were in this size range.

A potential cause for this trend could be the perceived time pressure by the developers during the three experiments with fixed ‘deadlines’, leading to a lot of copy and paste actions for identifiers and other small code segments which might otherwise be typed in manually.

However, it remains unclear whether the observed effect might be mainly related to the nature of the tasks chosen for the experiments and the characteristics of the project work done by the observed programmers, to the characteristics of the individual developers or whether this constitutes a general effect. It is to be expected that all three factors are involved. Though their individual weights are unknown.

It is furthermore interesting to note that the highly copy and paste intensive *Rubik’s Cube Solver* task (*cnp*) also displayed a different distribution of cut, copy and paste actions. The developer tended to paste the same code part multiple times and almost never executed a cut operation. This might be an indication for use of *syntactic templates* as defined by Kim et. al [38].

Conclusion

Overall, the obtained results confirm the observations of Kim et al. [38]. Programmers seem to produce a large number of copy and paste clones during software development tasks and while the large majority of these clones are too small to be of interest, the number of larger clones is still considerable.

During the analysed experiments and observations, programmers created an average of 24.76 new clones per hour. On average, 7.25 of these were larger than 80 characters and 2.7 were even larger than 250 characters. This means that an 8 hour working day would result in more than 50

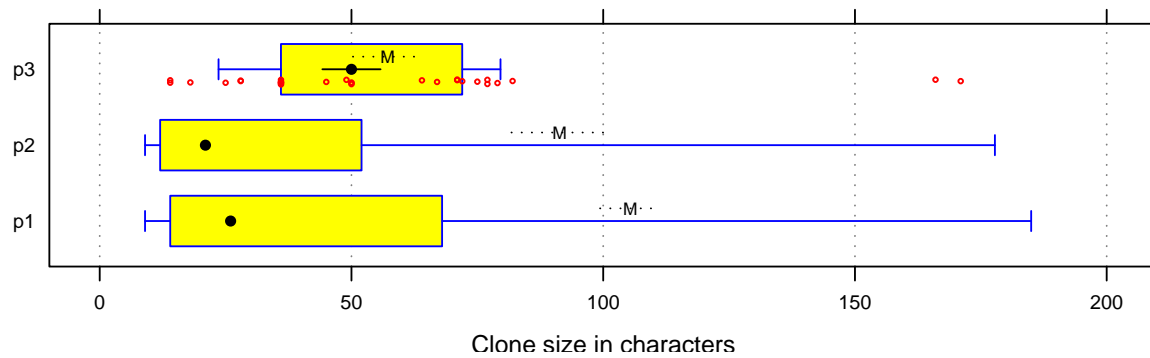


Figure 6.4: Comparison of clone size distribution

large clones (*more than 250 a week and 1000 a month*).

These findings hint at the potential seriousness of copy and paste cloning during software development and thus provide additional justification for this thesis.

6.2.2 CPC

Practical Application

Early prototype versions of *CPC* have been distributed to a number of people to facilitate the collection of initial feedback. This initial prototyping phase uncovered a couple of problems which were addressed in subsequent prototype versions. Unfortunately, a reliable collection of clone data was not possible during this phase. A number of changes made to the way clones are tracked and the clone data storage format required a number of resets of the clone database.

Starting in November a pre-release of the *CPC* plug-in entered its active testing phase. Since then *CPC* has been used by the author and two other programmers during their normal day to day programming work under *Linux*, *MacOS* and *Windows*. Thus covering many weeks of full time software development work⁶.

Aside of the rename refactoring problem described in section 5.3.2, no critical problem was reported and no further reset of the clone database was required. The clone data available for evaluation thus begins on the 14. November 2007, roughly two months before the deadline for this thesis.

Collected Data

During the two month long testing phase with three developers a total of 6,464 copy and paste clones were created under the supervision of *CPC*. The different data collection approach means that the obtained results can not be directly compared to the *ECG Lab* based data covered in section 6.2.1.

⁶The third programmer was not currently engaged in implementation tasks and thus created only a very small number of clones. All values for *p3* should therefore be taken with a grain of salt.

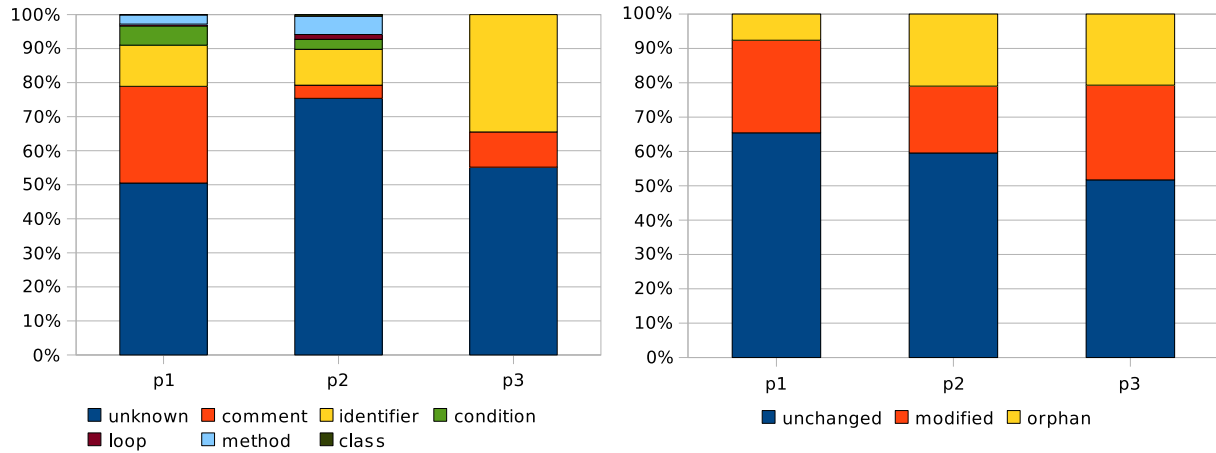


Figure 6.5: Clone content classifications and modification states

Figure 6.4 displays the size distribution of the created clones. It shows an expected trend, most clones created by developers during their day to day work were very small. For programmer 1 the median of the clone length in characters was 26 and for programmer 2 it was even only 21. Programmer 3’s relatively ‘high’ median of 50 needs to be considered with some caution, as was already outlined before. Even though large clones up to ten thousand characters in size did occur, they were very rare.

While the values may not be directly comparable, it is interesting to note that these values seem to contradict our assessment in section 6.2.1 in part. When compared with the size distribution in figure 6.2, the size distribution of the clones collected by *CPC* seems to resemble that of the clones collected during the three experiments more than it resembles the one of the clones produced by the two observed developers. More data will be required to provide a clear answer to our initial doubts about the validity of copy and paste data collected in short, artificial experimental setups.

Figure 6.5 shows the distribution of clone classifications as produced by the current *Classification Provider* implementation (see section 4.1). The high percentage of clones without classification makes it obvious that additional classification approaches are needed. Typical clones which do currently not receive any classification are collections of one or more lines that do not contain any control structures or clones which only cover a part of a programming construct, i.e. a method header and the first three lines of the method.

Aside from this obvious fact, the sample of three developers was too small and the variation between developers too high for any general conclusions. The high degree of cloning activities within comments observed for programmer 1 compared to programmer 2 might be related to their corresponding tasks. Programmer 1 primarily worked on a final implementation while programmer 2 mostly worked on a prototype implementation. Source documentation might thus have had more importance for programmer 1.

Figure 6.5 also displays the modification state distribution for all three developers. The majority

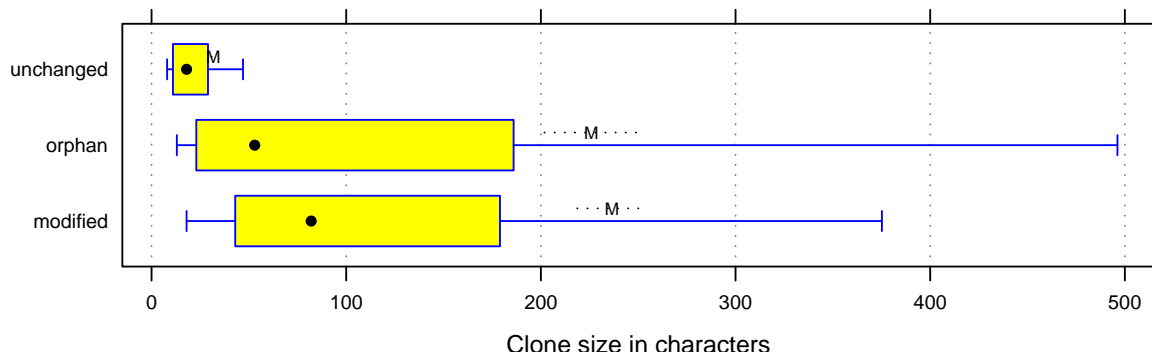


Figure 6.6: Size distribution of clones by clone state

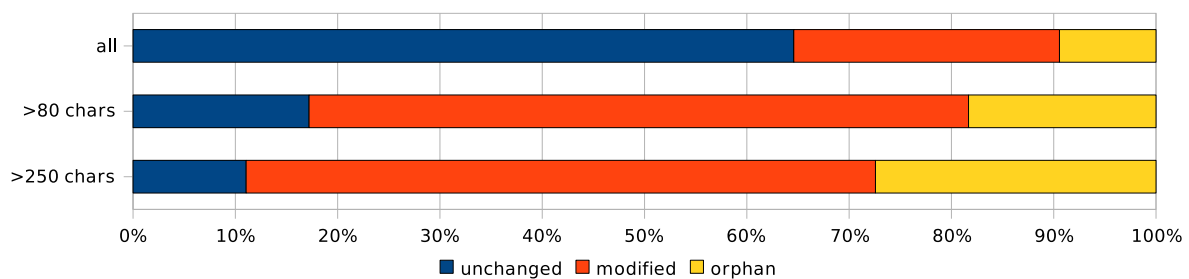


Figure 6.7: Clone state distribution by clone size category

of all created clones was never modified⁷ and a notable percentage entered the orphan state⁸.

It is furthermore interesting to look at the the modification state in relation to the size of the clones. Figure 6.6 shows quite clearly that most clones which remained unchanged were very small and that the average size of clones which actually ‘evolved’ in one way or another, be it a modification or the transition into the orphan state, was much larger. Which was to be expected as it seems generally very plausible that the chance of modification is strongly related to the size of a clone. This is also visible in figure 6.7, the majority of larger clones have been modified in one way or another.

The modified clones have been examined in a bit more detail. Figure 6.8 visualises the changes made to the modified clones in two metrics. On the left it shows the distribution of differences between the clone sizes before and after all modifications. It is interesting to see that the overall size of most modified clones changed by less than 10 characters during their lifetime. On the other hand, even ten characters can already represent a size change of more than ten percent for many clones (*the median length for modified clones is 82 characters, see also figure 6.6*).

However, a number of modifications may change a clone without affecting its size considerably. The left side of figure 6.8 displays the difference distribution by means of the *Levenshtein distance* between the contents prior and after all modifications⁹. Even though the difference between simply

⁷Orphaned clones can either be unchanged or modified. The total unchanged percentage is thus likely to be even higher than displayed here.

⁸A clone becomes an orphan once all other members of its clone group have been removed.

⁹Plain *Levenshtein distance* without any prior source code normalisation.

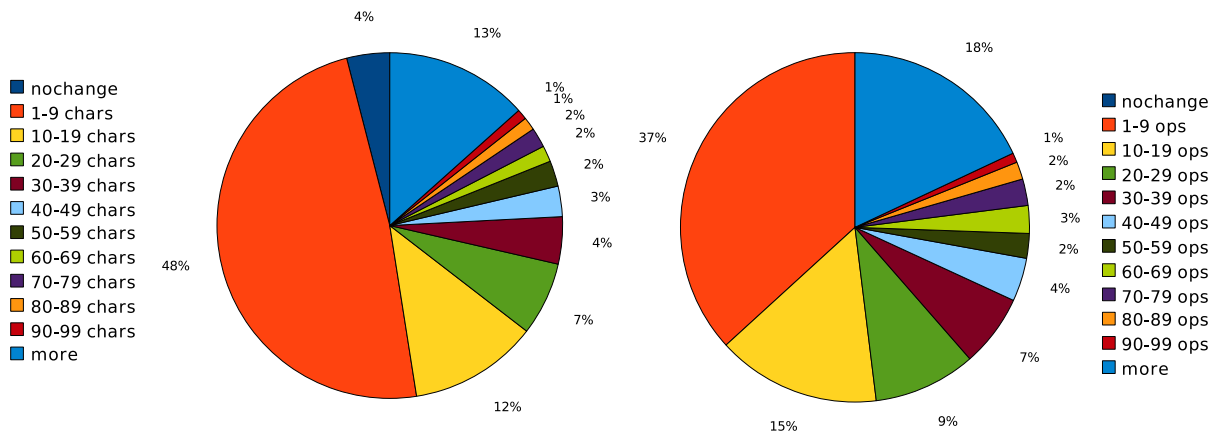


Figure 6.8: Size of clone modifications in content length difference and Levenshtein distance

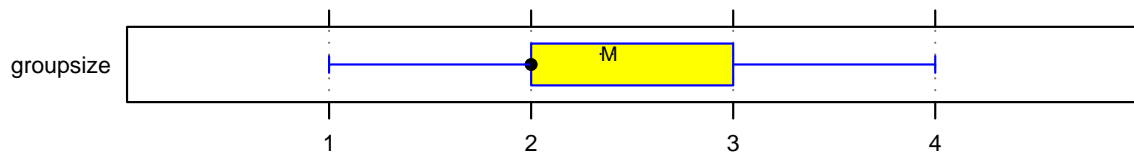


Figure 6.9: Size distribution of clone groups

using the clone length change and the *Levenshtein distance* is noticeable, the average modification made to each clone still remains relatively low.

By keeping track of the entire modification history of each clone *CPC* provides a new granularity of clone data which has so far not been available as existing static clone detection approaches are limited to the use of source code repository version histories which do not provide information about individual modifications. The remainder of this section lists some examples of such data.

While some clone groups had up to 72 members, most clone groups remained very small in size as can be seen in figure 6.9. Most clone groups, even those with more than two members, were entirely created shortly after the group itself was created. The median delay between the creation of the first and last member of a clone group is just 19 seconds which indicates that most groups

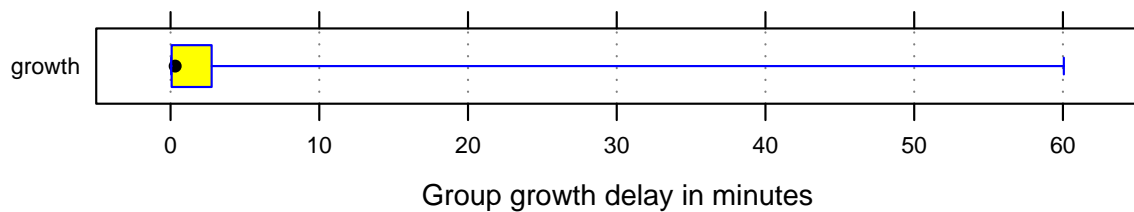


Figure 6.10: Delay between creation of first and last clone in group

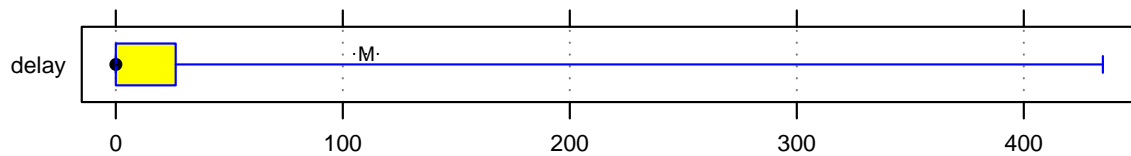


Figure 6.11: Delay in hours between clone creation and last modification

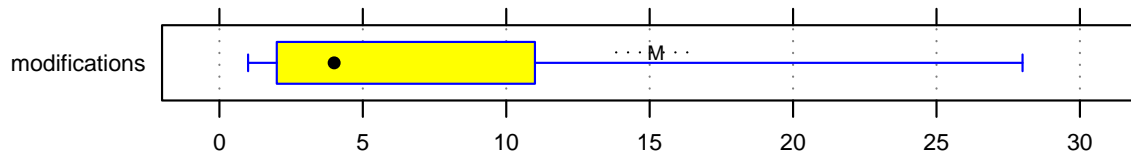


Figure 6.12: Number of modifications made to a clone's content

of a size larger than two were created by pasting the same clipboard content multiple times. This is also visible in figure 6.10. It should be noted that these values would be likely to increase if a more lenient *Fuzzy Position to Clone Matching Provider* would be employed (see section 3.4.2).

Figure 6.11 displays the distribution of the delay between clone creation and last modification. As expected a large number of clones were modified shortly after their creation and remained static from that point onwards. The median lies at two minutes and 37 seconds. Figure 6.12 represents another example. It shows the number of modifications *CPC* has collected for modified clones. While the number of modifications per clone tended to be low, most clones underwent multiple modifications before reaching their ‘final’ state. In most cases these modifications were small and happened in rapid succession. Figure 6.13 furthermore shows that the individual modifications tended to be rather small.

Conclusion

Considering the small developer sample and the time frame the collected data has already proven to be quite interesting. So far the data has mostly been used as means of feedback for areas of the *CPC* framework which need improvement but the author is quite confident that *CPC* based large scale data collection is likely to provide much better insights.

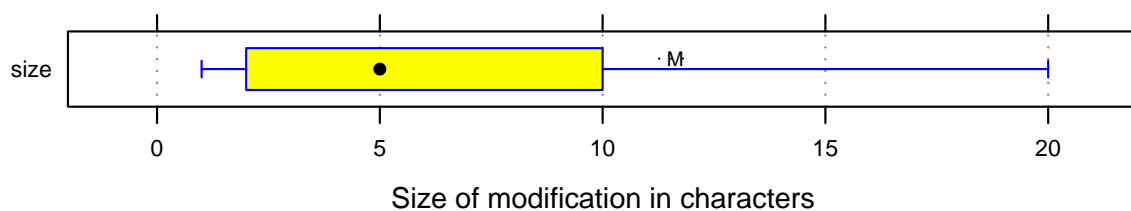


Figure 6.13: Size distribution of modifications made to a clone's content

However, development of tools and approaches to handle the resulting wealth of data is a crucial aspect of future work. Even now a large part of the information collected by *CPC* could only be examined very superficially. The 6,464 collected clones are accompanied by detailed information about 40,606 modifications which are likely to hold a number of interesting insights.

Chapter 7

Conclusion and Future Work

7.1 Looking back

The enormous complexity of the Eclipse platform posed one of the main challenges. Together with the fact that seldomly used APIs tend to be less well documented this made the adoption of a structured well planned approach very hard. A large part of the time invested during this project was spent on exploring and understanding documented and undocumented parts of the Eclipse platform and its APIs.

As was already expected at the outset of this thesis, missing and inappropriate APIs emerged as the key problems for *CPC*, especially in the area of remote synchronisation. The integrated *CVS* team provider of Eclipse is the only team provider which implements all Eclipse team APIs. I.e. the *Logical Model Integration* API, a major Eclipse team API, is not implemented by any of the available *SVN* team providers. And even when APIs are supported by a team plug-in, they are not always implemented completely. At other times existing APIs are implemented but do not align well with key requirements of *CPC* or crucially needed APIs, like a Team operation listener API, are missing completely.

Another point of serious concern were implementation details which, though important, were not part of the official specification. The Eclipse file buffer framework is a good example. While all plug-in contributors are encouraged to access documents only via this framework, direct access to the underlying file is still acceptable. Plug-ins like *CPC* which rely on observing all document modifications thus need to spend considerable effort on detecting and recovering from such situations.

Defects and failures in Eclipse components and other plug-ins represented another time consuming area as time spent on identifying the responsible parties and in some cases even debugging and patching them could have been put to better use. It was furthermore unfortunate that a reuse of the existing *ECG Sensor* proved to be infeasible rather late in the development cycle after considerable effort had been put into its restructuring and extending. In later stages of the project, the remote synchronisation aspects of *CPC* emerged as the main hindrance to *CPC*'s progress. Even with a lot of effort spent, a good reliable solution could not be achieved without requiring modifications or extensions to some of the Eclipse team APIs.

7.2 Looking ahead

In the short run, the way ahead is clear. Due to the time constraints for this thesis many parts of *CPC* only cover the basic requirements. Heuristics and strategies in general are prime examples. The overall usefulness of *CPC* can be improved considerably by providing additional strategies for the clone classification, similarity and modification notification heuristics. The interesting aspect of better strategy and provider implementations is their tendency to benefit a large number of other components within the *CPC* framework.

Additional clone data visualisations are another area which could greatly benefit *CPC*. The static clone detection research community has published a wealth of ideas on the topic such visualisations. Some available implementations are even based on Eclipse and might provide interesting reuse opportunities.

Once the Eclipse team API situation has improved, further progress on the topic of remote synchronisation would be highly interesting as it would open up a whole new target audience for *CPC*. In other areas Eclipse API improvements could lead to a reduction in complexity of the *CPC* framework if fewer special cases need to be handled. There is moreover no reason why *CPC* could not be extended to support other languages besides *Java*. All language specific areas of *CPC* are located within registered strategies and additional strategies for new languages could thus be easily added.

Section 2.3.1 furthermore introduced a number of potential future extensions which are not covered here again for the sake of brevity. However, an especially interesting aspect of *CPC* will be its potential as a source for large amounts of data on the copy and paste activities of programmers. If *CPC* is adopted by a large user base, it could help provide new insights into the micro-process of software development. In which case an entirely different topic might achieve a high relevance, data privacy. Large scale data collection is likely to not only require an automated data submission system, which *CPC* could easily support, but may also require automated anonymisation and obfuscation approaches, for users and source code. In case of a high adoption rate of *CPC* another interesting option might furthermore be the integration of a trimmed down *ECG Sensor* which could be used to optionally enrich the copy and paste data collected by *CPC* with further micro-activity data.

A very interesting topic for the near future will be the potential reuse possibilities between *CPC* and the copy and paste tracking tool *CnP* which is currently being developed at Clarkson University (see section 2.2.1). The very high overlap between the goals of both tools makes them almost predestined to exchange experiences and very likely to also reuse implementation aspects. Discussion with the author of *CnP* are currently ongoing and *CPC* may well emerge as the base framework for future versions of *CnP*.

Developments at the National University of Singapore (*NUS*) may also prove to be interesting. Some of the potential use cases for *CPC* at the NUS have already been outlined in section 2.2.1 and still others can readily be envisioned. The strong focus on cloning in software applications and on all the associated areas of research have resulted in a large number of tools like *XVCL*, *CloneMiner* and others which clearly hold collaboration possibilities [15, 58].

7.3 Conclusion

This thesis has presented a highly flexible framework for clone tracking within the Eclipse IDE. The strong emphasis on flexibility and future reuse has resulted in a complex but versatile implementation which currently consists of a collection of 28 highly independent plug-ins, 14 service providers, 101 interfaces, 355 classes and 66,573 lines of code¹.

CPC represents the very first copy and paste clone tracking utility available for the Eclipse platform which is ready for general use. It supplies the Eclipse IDE with a central integration point for clone tracking activities and represents an ideal base for all kinds of tools which require clone or position tracking functionality. *CPC* provides such extensions with a degree of resilience against external file modifications which has so far not been available within the Eclipse platform. The very open and loosely coupled nature of the *CPC* framework enables 3rd parties to reuse or exchange arbitrary parts of the implementation easily and ensures that contributions from different parties can coexist within the same *CPC* installation.

This project has furthermore uncovered multiple defects in the Eclipse platform and its team provider plug-ins and has provided patches and valuable information to aid in their removal. A proposal for a strongly needed new Team operation listener API was submitted and a number of other recommendations for improvements in the Eclipse platform and team provider APIs were made.

Feedback received from developers who were involved in the testing phases of *CPC* was overwhelmingly positive. It was reported that even very early versions which did not include any functionality aside of basic clone tracking and marking provided valuable information to the users. This is in line with our initial expectation that the clone tracking and visualisation features of *CPC* might provide improved awareness of the cloning situation within a software application. It is quite plausible that such increased awareness could lead to fewer clone related defects.

The evaluation of existing copy and paste data as well as new data collected during the testing phase of *CPC* yielded results on average cloning rates and the pervasiveness of copy and paste clones which confirmed earlier published findings. The detailed clone data collected by *CPC*, especially the clone modification histories and general clone evolution information, provides a new level of granularity much finer than available before. The employed copy and paste based approach furthermore provides a much higher precision than any static clone detection approach. Application of *CPC* might thus be able to provide new insights into copy and paste operations and the micro-process in general.

The large number of potentially very interesting extensions to *CPC* and the considerable interest at the Clarkson University and the National University of Singapore are likely to ensure continued progress and development. All in all, the future of *CPC* looks quite promising.

¹Figure does not include reused and automatically generated source code or blank lines.

Software and cathedrals are much the same
— first we build them, then we pray.

Bibliography

- [1] Apache commons logging. <http://commons.apache.org/logging/>.
- [2] Apache log4j — logging library. <http://logging.apache.org/log4j/>.
- [3] Castor — XML data binding framework for Java. <http://www.castor.org>.
- [4] db4objects — object database for java. <http://www.db4o.com>.
- [5] Eclipse online help. <http://help.eclipse.org>.
- [6] Eclipse online resources. <http://www.eclipse.org/resources/>.
- [7] Hsqldb — the lightweight 100% java database. <http://www.hsqldb.org/>.
- [8] PMD plug-in for Eclipse. <http://pmd.sourceforge.net>.
- [9] PostgreSQL — SQL DBMS. <http://www.postgresql.org>.
- [10] Team support for logical model integration. http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/guide/team_model.htm.
- [11] Eclipse platform — API rules of engagement. <http://help.eclipse.org/help33/topic/org.eclipse.platform.doc.isv/reference/misc/api-usage-rules.html>, May 2001.
- [12] Eclipse platform overview. <http://www.eclipse.org/platform/overview.php>, 2006.
- [13] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *ISESE*, pages 376–385. IEEE, 2005.
- [14] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In R. L. Krikhaar, C. Verhoef, and G. A. D. Lucca, editors, *CSMR*, pages 81–90. IEEE Computer Society, 2007.
- [15] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 156–165. ACM, 2005.
- [16] H. A. Basit and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In I. Crnkovic and A. Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 513–516. ACM, 2007.

- [17] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 451–459. ACM, 2005.
- [18] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In W. C. Chu, N. J. Juzgado, and W. E. Wong, editors, *SEKE*, pages 109–114, 2005.
- [19] J. Bloch. How to design a good api and why it matters. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 506–507, New York, NY, USA, 2006. ACM Press.
- [20] U. Borkowski. C4d website. <http://www.udo-borkowski.de/C4D/>, 2004.
- [21] B. E. Bulgaria. Simscan for eclipse. http://blue-edge.bg/simscan/simscan_help_r1.htm.
- [22] J. Cordy. Comprehending reality: Practical challenges to software maintenance automation. In *Int'l Workshop on Program Comprehension*, pages 196–206. IEEE Computer Society Press, 2003.
- [23] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167. IEEE Computer Society, 2007.
- [24] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.
- [25] T. Dudziak and J. Wloka. Tool-supported discovery and refactoring of structural weaknesses in code. Master's thesis, Technical University of Berlin, 2002.
- [26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [28] S. Giesecke. Clone-based Reengineering für Java auf der Eclipse-Plattform. Master's thesis, Carl von Ossietzky Universität Oldenburg, 2003.
- [29] P. Jablonski. Dissertation proposal - techniques for detecting and preventing copy-and-paste errors during software development. <http://coppypastecode.googlepages.com/Detecting-and-Preventing-Errors.pdf>, 2007.
- [30] P. Jablonski. Managing the copy-and-paste programming practice in modern ides. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA Companion*, pages 933–934. ACM, 2007.
- [31] P. Jablonski and D. Hou. Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. *OOPSLA - Workshop: Eclipse Technology Exchange*, 2007.

- [32] S. Jekutsch. Der Mikroprozess von Programmierfehlern. In *Software Engineering 2007 - Beiträge zu den Workshops*, Nachwuchsworkshop der Software Engineering Konferenz 2007. Gesellschaft für Informatik.
- [33] S. Jekutsch. Micro-process of software development website. <https://www.inf.fu-berlin.de/w/SE/MicroprocessHome>.
- [34] I. Jeong. Sdd for eclipse. [http://wiki.eclipse.org/index.php/Duplicated_code_detection_tool_\(SDD\)](http://wiki.eclipse.org/index.php/Duplicated_code_detection_tool_(SDD)).
- [35] I. Jeong and S. Lee. Sdd: high performance code clone detection system for large scale source code. In R. Johnson and R. P. Gabriel, editors, *OOPSLA Companion*, pages 140–141. ACM, 2005.
- [36] T. Kamiya, S. Kusumoto, and K. Inoue. Cefinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.
- [37] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, pages 19–28. IEEE Computer Society, 2006.
- [38] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.
- [39] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In S. Diehl, H. Gall, and A. E. Hassan, editors, *MSR*, pages 58–64. ACM, 2006.
- [40] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM.
- [41] M. Kranz. Animation vergangener Codeänderungen von Java-Methoden. <https://www.inf.fu-berlin.de/w/SE/ThesisCodeMetamorphoses>.
- [42] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.
- [43] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *MSR*, page 18. IEEE Computer Society, 2007.
- [44] Z. A. Mann. Three public enemies: Cut, copy, and paste. *Computer*, 39(7):31–35, 2006.
- [45] F. Mitter. Tracking source code propagation in software systems via release history data and code clone detection. Master's thesis, Technische Universität Wien, 2006.

- [46] S. Papadopoulos. Verfolgen von Kodekopien zur Defektvermeidung in Eclipse. Master's thesis, Free University of Berlin, 2007.
- [47] D. L. Parnas. Software aging. In *ICSE*, pages 279–287, 1994.
- [48] L. Prechelt. Plat_forms 2007 task: Pbt. Technical Report B 07-03, Institut für Informatik, Freie Universität Berlin, 2007.
- [49] L. Prechelt. Plat_forms 2007: The web development platform comparison — evaluation and results. Technical Report B 07-10, Institut für Informatik, Freie Universität Berlin, 2007.
- [50] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In D. Lowe and M. Gaedke, editors, *ICWE*, volume 3579 of *Lecture Notes in Computer Science*, pages 252–262. Springer, 2005.
- [51] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University at Kingston, 2007.
- [52] F. Schlesinger. Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung. Master's thesis, Free University of Berlin, 2005.
- [53] F. Schlesinger and S. Jekutsch. ElectroCodeoGram: An Environment for Studying Programming. Workshop on "Ethnographies of Code" at Lancaster University — <https://www.mi.fu-berlin.de/wiki/pub/SE/ElectroCodeoGram/lancaster.pdf>, March 2006.
- [54] U. Stärk. Empirische Untersuchungen des Side-by-Side Programming. <https://www.inf.fu-berlin.de/w/SE/ThesisSBSP>.
- [55] R. Tairas. Clone detection literature overview. <http://www.cis.uab.edu/tairasr/clones/literature/>.
- [56] R. Tairas, J. Gray, and I. Baxter. Visualization of clone detection results. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 50–54, New York, NY, USA, 2006. ACM.
- [57] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VL/HCC*, pages 173–180. IEEE Computer Society, 2004.
- [58] H. Zhang and S. Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, 53(3):381–407, 2004.

Unless otherwise stated all provided URLs were valid on 2008-01-23.