Generic modeling of code clones

Simon Giesecke*

January 18, 2006

Abstract

Code clones, i.e. instances of duplicated code, can be found in many software systems. They adversely affect the software systems' quality, in particular their maintainability and comprehensibility. Thus, this aspect is particularly important to consider in software maintenance and reengineering. Many different algorithms detecting code clones have been developed. For various reasons, it is difficult to compare the results of different algorithms. Most notable among these reasons is that there is no conceptual model allowing description of code clones determined by different algorithms. Much more, each algorithm uses its specific concept of code clones, which is rarely made explicit.

To overcome these problems, we have developed a generic model for describing clones. The model is generic in that it is independent of the programming language examined and of the clone detection algorithm used. It is flexible enough to facilitate various granularities of artifacts employed for selection and comparison, including inexact clones. The model allows separation of concerns between clone detection, description and management, which reduces the effort for the implementation of tools supporting these activities. On the basis of the model, we have implemented a prototype tool supporting these activities, tightly integrated into the Eclipse environment.

1 Introduction

One important goal of software engineering efforts is dealing with the inherent complexity of software systems, and the reduction of this complexity. Software systems can be made free of redundancy in theory. This is in contrast to other engineering products where repeated elements are commonly seen [Bro87], e.g. many occurrences of one particular screw in a machine. Muchmore it is desirable to avoid duplications in software systems to improve maintainability.

Looking at real industrial software systems, however, one can find that between 5% and 50% of the source code lines are duplicates [BYM+98, DPML02, MLH96]. There are many causes of this phenomenon. But one cannot expect a reduction in code duplication ratios, unless the duplication problem is explicitly focused during the development process. We consider two aspects important to the solution of the duplication problem: Firstly, there are *algorithms* that perform automatic clone detection, and, for another thing, there are modifications

*Carl von Ossietzky University of Oldenburg, Software Engineering Group, 26111 Oldenburg, Germany, simon.giesecke@acm.org

Dagstuhl Seminar Proceedings 06301
Duplication, Redundancy, and Similarity in Software
http://drops.dagstuhl.de/opus/volltexte/2007/960

of the software development *process* which enable practical use of the results of a clone detection algorithm. The first aspect is more of technical nature, the second of organizational nature. It is desirable for a software developer to have tool support at hand supporting both dimensions.

It is currently unclear if any of the currently available clone detection algorithms is superior [Bel02], much less if there is a particular granularity on which clones are considered that is universally adequate. For these reasons, it should be possible to choose from a variety of clone detection algorithms within one tool. However, prior work in the area of code clones is either directly concerned with a new detection algorithm, or presents work based on one such particular algorithm. Due to the lack of uniformity in concepts and implementations of clone detection algorithms, it is impossible to easily apply the results described to a different clone detection algorithm. This slows down scientific progress without necessity, since it is rarely the case that the concepts really depend on specific characteristics of the chosen algorithms.

The contributions of this paper are

- that a model for describing clones is made explicit,
- that an approach to clone description that is uniformly applicable, regardless of the clone detection algorithm used and of the level of granularity examined, incorporating represenation of inexact clones,
- that an application of the clone model is presented, providing a framework for developing clone management tools, thereby enabling effective separation of concerns regarding clone detection, clone description and clone management.

A prototype tool implementation has been developed which provides elementary clone management facilities for the Eclipse Java Development Tools IDE.

The paper is structured as follows: First, in section 2, we explain our understanding of the notion of code clones. In section 4 we propose a model to represent code clones in a generic way, i.e. the model is independent of the programming languages and clone detection algorithms used, and illustrate the model using a concrete code example. Then, possible uses of the model are discussed and our tool prototype is briefly presented (section 5). Finally, related work is discussed in section 6 and conclusions are drawn in section 7.

2 Notion of code clones

Simply put, a *clone* is a manifestation of redundancy in a representation of a software system. By *cloning* we refer to the—intentional or unintentional—process leading to the emergence of clones.

Of particular interest to us are clones in those representations which are under direct control of the developers. Among these are concept and architecture diagrams as well as the source code. In the latter case, we speak of *code clones*. A *concept clone*, in general, is a clone already apparent in a more abstract representation of the system than that under inspection, which may propagate into duplicated artifacts in the inspected representation. However, such a propagation of clones is not imperative: Let us consider documenting

requirements by a set of use cases, each of which is annotated by multiple scenarios. If these scenarios are written down independently of each other, partial redundancies between scenarios are inevitable. These redundancies, however, will not commonly cause redundancies in the system's architecture.

We are interested in detecting function clones, so we focus on a special case. In general, it is impossible to automatically detect arbitrary function clones as an implication of Rice's theorem [Ric53]. Thus, we regard functional specifications as concepts, and their implementations as lower-level artifacts. While two identical source code fragments will most probably refer to the same functional concept, one concept may be implemented in arbitrarily deviating ways. In practice, however, this problem will not be as hard, since a finite set of developers will follow a similar style and is therefore likely to adhere to a small number of implementation variants of each concept. Summarizing this discussion, we assume, that functional similarity often results in structural similarity of the source code.

While a central concern of software analysis and design is the systematic avoidance of redundancies, this is not the case in the implementation phase. Probably, one reason for this is that the lower the level of abstraction of a software system's representation is, the greater is the representation's size and complexity. When no single person knows all the code of the project, no one can determine if a specific fragment might be a clone.

2.1 Effects of cloning on software qualities

We now consider the effects that cloning has on the qualities of a software system, i.e. the difference of the qualities of the software system f containing clones and those of a supposed equivalent software system $\bar{f}$ differing from f only in that $\bar{f}$ contains no clones.

First of all, clones will directly affect software *metrics* like lines of code, number of statements and size of executable code, making the system f larger than $\bar{f}$. Kontogiannis et al. [KDM+96] found that cloning negatively affects maintainability and comprehensibility. *Comprehensibility* is affected primarily because the principle of locality of functionality is violated, i.e. one specific aspect of functionality is implemented by each instance of a clone.

Maintainability is related to comprehensibility [Ghe03], in that better comprehensibility increases maintainability (and vice versa). In addition, a problem affecting maintainability more specifically is introduced by cloning: When a developer modifies one clone instance, probably the *changes* must be *propagated* into the other clone instances—at least they should verify if this is necessary. However, a check for propagation is only possible if they know that a code fragment being modified is a clone instance at all. Even if this is the case, it is possible that only a subset of all instances is known. Furthermore, the propagation may be performed erroneously. This poses a particular problem if the clones are not exact, since it may not be obvious if and how a change has to be propagated. A *systematic* approach to *clone management*, involving detection and analysis of clones and the handling of resulting information, is required to overcome these problems.

In contrast to silently avoiding clone introduction during development, clone analysis provides an opportunity in itself: Cloning of the implementation of a certain functionality indicates that this functionality might provide a candidate

for a unit of reuse [KDM+96]. Clone analysis may be used to identify such functionality and trigger transformation into explicit units of reuse (modules, components, etc.), consequently increasing reusability.

2.2 Emergence of code clones

There is a variety of circumstances under which code clones may emerge. We will analyze these circumstances with respect to the technical means that are used to create clones, and to the reasons for which developers create clones. These dimensions are not always clearly separable.

2.2.1 Technical means of clone creation

To start with, we consider which technical means are used to create clones. Later, in section 2.3, we will turn back to these techniques and analyze how they relate to the degree of exactness of the resulting clones. We call a clone *exact* if its instances are textually identical; several degrees of lesser exactness will be considered.

The most obvious technique that leads to duplicated code is to simply *copy* certain code fragments ("copy and paste programming").

Certain *idioms* (also "mental macros" [BYM+98]) specific to a programming language or framework, which are used as a matter of routine, may also lead to clones. An idiomatic programming style certainly has its own right, since it makes code easier to read and understand. But if such idioms are used at too large a scale, they might better be replaced by more explicit abstractions.

When *merging* two software systems, it is clear that certain functionality will be duplicated among the constituting systems. The amount of duplicated functionality will differ depending on the similarity of the overall purpose of the merged systems.

Finally, there might be no technical relation between clone instance, apart from their intrinsic relation of implementing similar functions.

2.2.2 Reasons leading to the emergence of clones

Every single clone apparent in a software system will have been created at some point in time during the software development process. The reasons leading to the creation of clones may be of technical, psychological or organizational nature. Some of these reasons apply only to specific means discussed in section 2.2.1, others are more general.

Developers may introduce clones out of *laziness*. At first sight, it may appear to them that cloning by copying or implementation as a matter of routine takes less time than to proceed under avoidance of cloning. When a developer clones by copying, obviously at least one other clone instance is known to them. This might not even be the case when they repeatedly implement an idiom as a matter of routine.

Cloning may also be done expressly and intentiously with a similar rationale. If a certain code fragment is known to work well, developers might intent to *reduce the risk* of breaking it by copying it and adapting only the copy for use in a different context. In the light of change propagation problems it is questionable if this really reduces risks of overall errors.

Thus, besides blaming the developers for not putting enough effort into avoiding clones, we have to consider *deficiencies in knowledge* about the software system. The larger the subject software system is, the less probably the developers know in detail which functions may have been implemented by others. Then, communication deficiencies form another aspect of the same problem. Each single developer might not even know what they have implemented. This is most notably a problem for artifacts too fine-grained to be reflected on the architectural level.

A code clone may result from too strict an adherence to the waterfall model of software development [PC86]: Before it is known in full which functionality is needed by which parts of a software system, an architecture is defined without the intent to be subjected to later modification. Since dependencies between modules are fixed at the *architectural level* in an early phase of development, it may become necessary to reimplement functionality of one module in another if they are not declared to depend on each other. The easiest way to overcome this problem is to copy the concerned code. Of course, this problem may also be the result of intentional architectural decisions after weighing advantages and disadvantages. Another reason leading to similar problems is the use of a framework which requires or recommends a certain architectural style that can or should not be changed.

One specific intention for introducing clones which is not related to structural issues is explicitly cloning for *performance* reasons during an optimization phase. In spite of tremendous advances in available computing power, this is still relevant today, particularly in the development of embedded real-time systems.

Clones caused by using idioms might be a consequence of a *lack of abstraction* mechanisms of the language—or poor use of such mechanisms. In the Java language, up to version 1.4, such a typical idiom is the iteration of a collection, during which certain actions were performed on the individual elements (cf. figure 1). A more general example—applicable to most object-oriented languages—usually considered bad style in itself is the use of conditional constructs based on dynamic type checks [DDN02, ch. 10].

2.3 Exactness of clones

In this section we will consider the question of exactness of clones. In particular, we will analyze how the circumstances of clone creation relate to whether a clone will be exact or inexact. We will show that it is not sufficient to restrict clone management to the consideration of exact clones.

2.3.1 Degrees of exactness

The exactness of the clone property of a pair of code fragments may be considered in different terms. Considering the code itself, textual identity may be viewed as the most apparent notion of exactness. Considering the code's function, behavioral equivalence might be the ideal notion of exactness. Obviously, these notions are not equivalent, but related: in an informal sense, textual identity will imply behavioral equivalence, but not vice versa. Various other degrees of equivalence can be defined between these two extremes, including lexical and structural equivalence.

No matter which notion of equivalence one chooses to use, maximal exactly cloned code fragments are surrounded by code differing from instance to instance: considering an exact clone relation $\equiv$, given code fragments $A = \text{concat}(a_1, a_2, a_3)$ and $B = \text{concat}(b_1, b_2, b_3)$ where $a_2 \equiv b_2$ but $a_1 \not\equiv b_1 \wedge a_3 \not\equiv b_3$, A and B form a clone which is *partially exact* with respect to $\equiv$.

2.3.2 Relation to clone creation

Copy-and-paste-programming results in code clones that are exact in the beginning. However, modifications are often performed afterwards to adapt the copied code fragment to a new context, finally resulting in arbitrarily inexact clones. This reflects the fact that cloning is an informal mode of reuse, in contrast to formal reuse of functionality properly encapsulated in a component.

The degree of exactness of clones created through the use of idioms may vary depending on the strictness of style guidelines, number of developers and similar organizational parameters.

Considering the merging of systems, if the developers of the merged systems are distinct, a lower grade of similarity between implementations of similar functions may be expected.

Finally, for clones whose creation was technically unrelated, no assumption may be made about their exactness, despite that the more similar the implemented functions the more exact the resulting clones will be.

2.3.3 Risks

In addition to the effects of clones on a software system described in section 2.1, the inexactness of clones introduces further risks to the software system. Inexact clones may violate the *principle of regularity*, which means that interfaces and behavior of similar functions within one software system should vary as little as possible. Violations of this principle may include deviations in the order of parameter lists, handling of borderline cases and of error conditions.

2.3.4 Detection

Unfortunately, inexact clones do not only pose additional risks, but they are harder to detect as well. This applies to both manual and automatic clone detection. Especially for the latter, the problems are

- to define in a generic way which differences between potential clone instances should be allowed,
- to construct decidable criteria or algorithms based on this definition,
- and to evaluate such criteria.

In principle, these problems have to be considered for manual detection as well, but they may be neglected more easily without apparent consequences.

Based on the differences between instances of inexact clones, an additional classification of clones may be useful [BMD+99a, Bel02, LPM+97, MLM96].

2.4 Avoiding and handling clones

Now, after having discussed why clones should be avoided and where they result from, we focus on the question of *how* to avoid clones. The first distinction to be made in this context is between clones that are *explicitly* created by a developer and those that come into existence only *implicitly*.

It is desirable to inhibit the creation of clones before or at the time they come into existence by technical means. Unfortunately, this is only possible for explicitly created clones, and even for those this is not feasible in practice. One possible means were to monitor editor actions for copy-and-paste operations. One could simply inhibit copy-and-paste operations. Such an editor would not be accepted by developers as long as their reasons for using copy-and-paste programming in the first place remain. Another possibility were to allow copy-and-paste operations and monitor what happens to the pasted section afterwards. One advantage over an ex-post-analysis is that, no matter how much editing is done to such a clone instance, the information, that it has been cloned initially, will not get lost. But this poses another problem, as too many fragments will probably be marked as clones. In any case, copy-and-paste-monitoring is not sufficient since clones are created by other means as well.

Implicitly created clones cannot be addressed by such an inhibition mechanism at all. One could think of tool support not inhibiting clone creation but supporting an attentive developer by providing them with some means that avoid a course of actions leading into cloning in the future. This might be done by providing means for writing more general code. But, as pointed out by Fowler [Fow99], it is very difficult—and often turns out not to be feasible—to prematurely synthesize a useful generalization of a code fragment from only one or two instances.

For these reasons, we think such a solution to avoid clones is required, that relies on developer cooperation to a greater extent. Instead of inhibiting their creation, handling of clones soon after their creation is desirable. The aim of this approach is to *avoid independent evolution* of clone instances. In this context by independent evolution we understand such modifications to single clone instances that do not fundamentally remove the functional similarity of the clone instances, but destroy their structural similarity.

2.5 Summary

In this section we first discussed why the presence of clones in software systems is problematic. Then, we dicussed why and how clones come emerge within software. Afterwards, we explained why it is not sufficient to constrain to exact clones. We concluded the discussion by motivating that a systematic approach to clone management is necessary to overcome the problems introduced by clones. In the next section, we present our approach to modeling clones.

3 Integrated clone management

In this section, we will present one possibility to integrate clone management into the development process. First, three aspects of clone management are introduced, to which clone management activities can be attributed. Then,

scenarios for clone management which are closely related to the necessary tool support, will be discussed.

3.1 Aspects of clone management

Clone management summarizes all process activities which are targeted at detecting, avoiding or removing clones. All clone management activities can be associated with one or more of three groups, depending on their relation to the "life cycle" of the clones which they refer to. *Corrective* clone management deals with the removal of existing clones from the software system, *preventive* clone management deals with the avoidance of the introduction of clones into a software system, and *compensatory* clone management deals with avoiding negative consequences of existing clones in a software system. [1]

This classification was inspired by Mayrand et al. [MLH96] who proposed two concrete activities dealing with clones, which they call problem mining and preventive control. They may be grouped into compensatory and preventive clone management, respectively. They do not regard corrective clone management activities. In contrast, most work on clone detection tools exclusively consider corrective clone management as an application area.

3.2 Clone management scenarios

We identified five dimensions of clone management scenarios which are relevant to the support by software tools. These dimensions deal with the central or decentral character of the tools or measures, the locality of the data used, the question of what an individual clone management activity is triggered by, the scope of the activities with regard to the subject system and their relation to the development process as a whole. Each dimension will be explained in detail.

3.2.1 Centralization of clone management tools

The implementation of most software systems is created in teams whose members collaborate working on different, possibly overlapping parts of the system. The technical environment their work relies on can be characterized as a distributed system with local, decentralized programming environments or editors and a central repository which often provides configuration and version management functionalities.

In such an environment, clone management functionality may be implemented either at the central repository (*centralized* CM) or the local programming environments (*decentralized* CM). While a decentralized implementation may be done with neither regard to nor impact upon the central repository, due to the client/server architecture, a centralized implementation will affect both the repository and the local programming environments. With a decentralized scenario, every developer might use different tools, and some might not use CM at all. Thus, centralized CM requires greater effort and offers less flexibility than decentralized CM.

The concrete meaning of "introduction" of a clone into a software system is affected by this dimension. When using centralized CM, the introduction is performed when checking in a modified version of a file into the repository,

[1] name concrete activities?

whereas with decentralized CM, a clone may already be seen as introduced when it is typed.

3.2.2 Locality of clone management data

Staying with the scenario introduced in the previous section, the data building the foundation of the clone management may be used *locally* only or *globally*, i.e. it is shared between the developers. When clone management data is exclusively generated automatically[2], this dimensions only affects the efficiency of the implementation. However, when manually entered information is integrated into the clone management data, this dimension also affects whether communication between developers is supported by clone management tools.

Using a centralized implementation, the natural way—i.e. that which requires least effort—will be to share the data, while in a decentralized implementation a local-only use of the data will be easier to implement. However, both local and global solutions are possible for both forms of implementation.

3.2.3 Triggering of clone management activities

An individual instance of a clone management activity may be *scheduled* in advance as part of a larger plan of process activities, in which case a certain periodicity of clone management activities' instances may be assumed. In this case, the subject who triggers such an instance is not the developer herself (possibly in person, but not in role). This may be contrasted by an ad-hoc triggering of each clone management activity instance by the *developer*. A metaphor helpful in this context is that of the "bad smells" [BF00]: Upon notice of a bad smell presumably caused by code duplication, the developer initiates a clone management activity. Finally, an instance of a clone management activity may be triggered as a *side-effect* of another activity. This kind of trigger may be particularly useful for compensatory clone management activities. Of course, a combination of these approaches is conceivable.

3.2.4 Scope of clone management activities

An instance of a clone management activity may focus on a single specific clone and its instances, which we call a *clone-focused* activity, or deal with all clones in the subject system or one of its parts, which we call a *system-focused* activity.

3.2.5 Relation to development activities

Clone management activities may be considered part of regular code-centered development activities, and may then be called *development-accompanying*. When using agile development processes, this view suggests itself. On the other hand, clone management activities may be considered an element of the software process on the same level as code-centered development, which we will call *development-external*[3].

Development-external clone management is in some sense always a scheduled activity, as it is considered in the whole-scale process schedule. This variant

[2] introduce distinction between automatic generation and manual editing somewhere (as another dimension?)

[3] find a better term

will suggest to be system-focused, due to the greater overhead of each activity instance.

3.2.6 Discussion

The selection of a scenario for our work was driven by the question whether the scenario asks for a specialized, proprietary environment or it can be realized in a commonly used environment with low overhead. The scenario proposed and analyzed in [LPM+97] can be characterized as a centralized, global-data, side-effect-triggered, clone-focused, and development-accompanying scenario. Due to its centralized, client-server architecture, it requires special tool support both on the client and server sides, and use of a special configuration repository.

The scenario chosen in our approach can be characterized as decentralized, local-data, developer/side-effect-triggered and development-accompanying.

4 Generic modeling of code clones

In this section, we present the generic clone model we have developed. Firstly, in section 4.1 an introduction to clone detection techniques in general and their implicit models of code clones is given. We propose how candidates for clone instances are chosen in section 4.2, and how inexact clones are handled in section 4.3. We illustrate the concepts by a concrete example in section 4.4. Then, an overview of the elements of our clone model is presented in section 4.5.

4.1 Clone detection techniques at a glance

The overall goal of a clone detection technique is to determine a binary relation representing the clone property on some set of code artifacts A, i.e. $\simeq \subset A \times A$. Such a relation is called a *clone relation*. In the description of most clone detection techniques, the set A is only implicitly characterized. We will discuss this point in detail in section 4.2. To start with, we evaluate the characteristics of the output of clone detection methods, i.e. how the clone relation is presented to the user.

Bellon [Bel02] found that the majority of clone detection tools produce a list of pairs of clone instances, which can be seen as the natural representation of a binary relation. Exceptions to this rule are the algorithms described by Baxter et al. [BYM+98] and Mayrand et al. [MLM96], which produce a list of clone sets without and with distinguished elements, respectively. However, looking closely at these two exceptions reveals that they also share the notion of an underlying binary clone relation and only form the deviating representations during post-processing. It may be concluded, that the post-processing is not a direct implication of the clone detection method but of the intended mode of use of its result. In fact, this kind of post-processing is independent of the actual clone detection algorithm, but it depends on the characteristics of the clone relation. We consider clone relations to be reflexive and symmetrical. In case non-exact clones are considered, a clone relation cannot be generally transitive [Bel02].

4.2 Candidates for clone instances

In order to discuss the definition of the base set of the clone relation, we will first consider clones only as being partially exact at one specific level of granularity. This means, we ignore the details of differences below that level. In a second step, we look at how the model must be amended to fully support inexact clones in section 4.3. We propose to use as such the set of distinct artifacts in some structure derived statically from the source code of the subject system at a well-defined level of detail, which we call the *selection granularity*. The individual artifacts are called *selection units*. Each selection unit may or may not correspond directly with a syntactical element of the programming language concerned.

This definition leaves plentiful possibilities, so we consider several additional criteria in selecting a concrete granularity. However, we believe that different choices may be adequate, depending on the specific situation. The former three criteria refer to selection units as *system artifacts*, the latter two to selection units as *clone constituents*. Of course, the criteria are not mutually independent. Particularly, the latter two are closely related. The criteria are presented with a brief rationale:

coverage The set of all selection units should cover all relevant aspects of the software system, which most importantly include aspects determining its behavior.

significance of individual units A developer should be able to gain understanding of every individual selection unit with little effort. A proper naming scheme will help to achieve this goal.

distinction of multiple units A developer should be able to distinguish several selection units without having to explicitly consider their context. A proper naming scheme will similarly help to achieve this goal at a coarse-grained level.

significance of clone property The attribution of the clone property to a set of selection units should be reasonably significant. Particularly, this requires that the criterion used to decide on the property is easily comprehensible and selective. Selectiveness we must take care of the fact that the details of partially exact clones are ignored. It is not helpful if only such criteria can be established which attribute the clone property either to nearly all or to virtually no pairs of selection units.

probability of informal reuse The selection units should feature a high probability to be informally reused. Informal reuse in this context is to be understood as any means resulting in duplicated functionality other than rigorous reuse techniques. The most apparent among the possibilities of informal reuse is copying the relevant source code.

If artifacts are informally reused only on a level below the chosen selection granularity, it is difficult to find a selective clone criterion on this level: either few similar sub-artifacts suffice to make the selection units a clone, which delivers too many clones, or too many cloned sub-artifacts are overseen.

Considering software implemented in the Java programming language, several possibilities are apparent to choose the selection granularity from. We will

assess the fulfillment of the criteria for three possible choices of selection granularity:

classes/interfaces Every source code fragment besides certain comments, package declarations and import statements (which are considered not to contain information relevant to cloning) is contained in a class or interface declaration. Thus, the coverage is nearly complete in this case. The significance attributed to classes is generally well-defined, but may be too complex to comprehend instantly. It is possible to distinguish classes on a coarse-grained level by their names.

It is unlikely that whole classes are informally reused, as more rigorous concepts like inheritance and delegation are well-known for this purpose. Since the elements of a class are not ordered, comparison of classes is difficult to achieve, unless their members' names can be matched.

methods Not all behavioral aspects are contained in methods, e.g. static, instance or member initializers. However, we assume that such aspects may be neglected: it is unlikely that they alone contain relevant duplicated code, without correlating with duplications in method code. Methods are relatively constrained artifacts[4]. Developers usually attribute some meaning to methods that is well-defined at least in a subjective manner. Certainly, methods are less complex than classes, and are thus easier to compare. A coarse-grained distinction is easy by using the methods' qualified names.

Methods are likely to be informally reused, as there is no easy rigorous way to reuse methods from another context. On the other hand, it is likely that, when a functionality has already been considered a candidate for isolation in a method of its own once, this will be the case again in a different context.

statements Using a similar argument as before, statements do not feature perfect, but good enough coverage of a software system, though slightly worse than methods. A single statement is typically not very complex, but its meaning is incomprehensible without consideration of its context. Two identical statements may have completely different functions when used in different contexts.

The informal reuse of short statement sequences, and thus of single statements, is very likely, since they may be too small or too variable to be factored out into a method without much effort.

We conclude that, commonly, the choice of methods as selection units will be most adequate: it provides good-enough coverage, good significance, easy distinction and high probability of informal reuse. However, in case of a non-standard programming style—possibly resulting from an automated language conversion—, a different choice may be better. To make a proper decision for a given software system, a quantitative evaluation might be adequate. In this paper, we will constrain to giving an example in section 4.4.

[4]Here, issues arising with the use of anonymous or local classes are neglected.

4.3 Modeling inexact clones

A binary relation on a set of selection units suffices to describe exact clones. But, as reasoned in section 2.3, the majority of clones in general, and the majority of clones of greatest interest to developers in particular, are not exact. Thereupon, the question arises how the details of inexact clones may be described on the basis of the model just introduced. A simple possibility to extend the model to support the handling of inexact clones is to annotate each clone pair with some measure of the degree of difference or similarity. For two reasons, such an approach is not sufficient. Firstly, when trying to evaluate different clone detection algorithms, the results of two algorithms are difficult to compare: it is not transparent which structural elements below the selection granularity contributed to the decision to consider a pair selection units a clone. Secondly, if the result of the clone detection process is considered to be used for clone removal, such details of clone instance differences are required for selecting and applying removal strategies (see, e.g., [BMD+99b]).

Therefore, entities on a level of detail below the selection granularity are incorporated into the model, which we call *comparison units*. The following constraints are imposed upon comparison units:

1. Each comparison unit must be unambiguously attributable to an enclosing selection unit. Thus, the comparison granularity must be finer or as fine as the selection granularity.

2. The comparison units within a selection unit must be arranged as a sequence or as a tree-like hierarchy.

An *enumeration function* determines the linear sequence of comparison units contained in a selection unit. In the case of hierarchically organized comparison units, the pre-order linearization of the tree is used for this purpose.

An *induction function* relates cloned pairs of comparison units to a clone pair at selection granularity, and decides whether the selection units are sufficiently covered by cloned pairs of comparison units to be considered a clone. Basically, arbitrary cover, total cover and threshold cover may be considered.

Taking the Java language as an example again, when we fix the selection granularity at method level, the comparison granularity may be chosen such that the comparison units of one selection unit are the whole method body, the sequence of lines contained in the method body, or the hierarchy of subtrees of the AST of the method body. These choices will be illustrated by the example in section 4.4.

Given a specific choice of selection and comparison granularity, a language-specific classification of clone pairs based on the nature of their instances' differences can be sensible (cf. [BMD+99a]).

4.4 Example

We will now apply the model described before to a small example written in the Java programming language. Consider a class `MyDictionary` partially presented in figure 1. The class encapsulates a dictionary data structure, which maps keys to values. `MyDictionary` adds two alternative access methods `getValue` and

```
1  class MyDictionary {
2      Dictionary properties;
3      //...
4      String getValue(String searchKey)
5      {
6           Enumeration enum =
7               properties.keys();
8           while (enum.hasMoreElements())
9           {
10              String currentKey =
11                  (String)enum.nextElement();
12
13              if (currentKey.startsWith
14                      (searchKey))
15                  return (String)
16                      properties.get(currentKey);
17          }
18          return null;
19     }
20
21     String getValueIgnoreCase(String key)
22     {
23          Enumeration keyEnum =
24              properties.keys();
25          while (keyEnum.hasMoreElements())
26          {
27              String currentKey =
28                  (String)keyEnum.nextElement();
29
30              if (currentKey.equalsIgnoreCase
31                      (key))
32                  return (String)
33                      properties.get(currentKey);
34          }
35          return null;
36     }
37 }
```

Figure 1: Example: two similar methods

`getValueIgnoreCase` which search for keys in a certain manner[5]. In the two methods, differences are marked by bold typeface. Most differences consist only in the names of identifiers, the one exception is additionally marked by underlining.

We will now consider three alternatives for representing clone relationships between the methods within our model. Due to space limitations, only a fraction of each representation is shown.

Lines as selection units First, we will consider using code lines as selection units, without subordinate finer-grained comparison units. This is equivalent to choosing the comparison granularity at the same level as the selection granularity. The situation is visualized in figure 2.

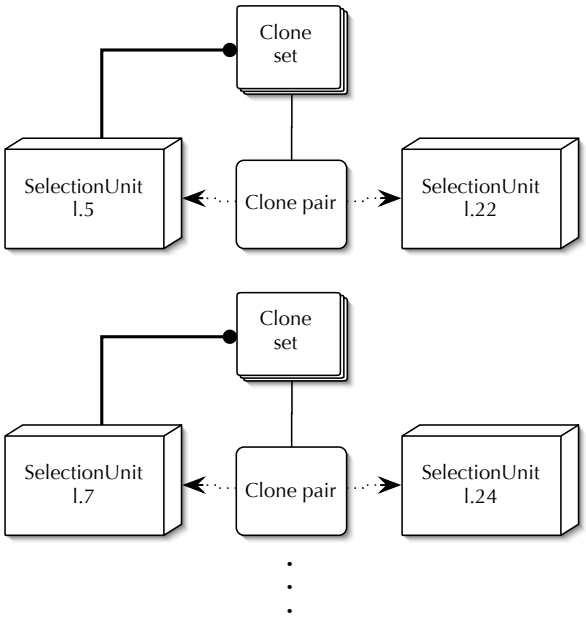

Figure 2: Example: lines as selection units

On the top level of the clone artifacts we find a *clone set*. A clone set consists of a set of *clone pairs* and a *reference element*. In our example, however, we will always find a one-to-one relationship between clone sets and pairs. There is a clone set and clone pair for each matching pair of code line. The first two of those are shown in the diagram. Lines 6 and 23 do not match in a line-level based comparison, since we considered only textually exact matches here.

Comparison units are not visualized in the diagram as there is necessarily a one-to-one relationship between selection units and comparison units when their granularity is the same. In a tool realization, the explicit recognition of comparison units may be sensible. It is then possible to uniformly perform comparisons only on comparison unit level. Consequently, the tool design is simplified.

[5]The two methods are not implemented very efficiently, but it is certainly conceivable that such implementations abound in real systems.

The presented result is only one possible outcome of a clone detection algorithm operating at the chosen comparison granularity. The concrete results depend on how the comparison of lines is performed, e.g. if and how a normalization of code lines is applied, which was not done in this example.

Reasoning about clones at this level is very difficult, since the information carried by one clone set is too constrained to be meaningful, and no relation between different selection units is made explicit. In principle, this problem could be solved by regarding line sequences as selection units, but this would pose the problem that selection units may overlap. An alternative solution, that is more natural in an object-oriented, imperative language, is to choose methods as selection units.

Methods as selection units, ...

... lines as comparison units Using methods as selection units, there are still several possibilities of how to choose the comparison granularity. We will first regard code lines as selection units, which allows direct comparison with the previous example. Again, only exact matches are considered. The resulting situation is visualized in figure 3.

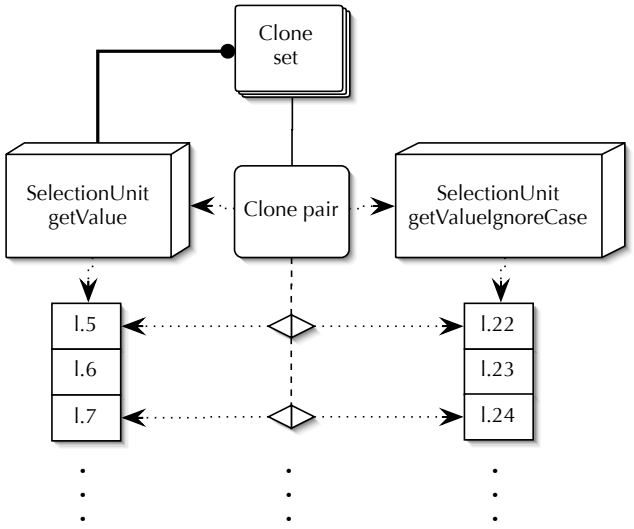

Figure 3: Example: lines as comparison units

In contrast to figure 2, the code lines, which were previously unordered, have been transformed into an ordered sequence of comparison units below the enclosing method as selection unit. In addition, the clone pairs of the previous example can be found as clone pairs at comparison granularity in this representation. These are depicted by a double triangle symbol which carries references to the corresponding comparison units. The clone pairs at comparison granularity enrich the information carried by the clone pair at selection granularity.

... AST subtrees as comparison units Finally, we consider the most structured of our example representations, using AST subtrees as comparison units.

The situation is depicted in figure 4.

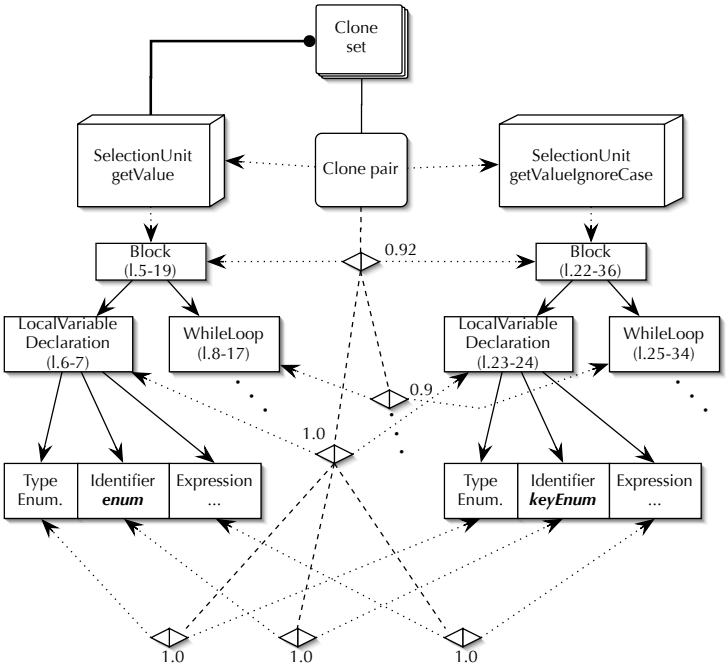

Figure 4: Example: AST subtrees as comparison units

The comparison units within each selection unit correspond to the nodes of the AST of the methods' bodies. In contrast to figure 3, they are hierarchically, not linearly structured. This attributes to the clone pairs at comparison granularity as well. In addition, the latter are annotated by a number denoting the degree of similarity of the participating units. Atomic similarity values are determined at the leafs, and are propagated towards the root using a weighting algorithm. In the previous examples no similarity values were shown. Since only exact matches were considered, all similarity values were equal to 1.0, rendering this information redundant.

While lines 6 and 23 were not considered clones in the previous examples, the variable declarations as a whole are considered a perfect match, since they differ only in the identifiers' names. However, in the loop which is not shown in detail, there is a significant difference (the method called on `currentKey` in lines 13 resp. 30), resulting in a similarity value below 1.0 on the loop and block level.

Various clone detection algorithms work directly on AST representations (e.g. [BYM+98]) or even more sophisticated representations based on the AST (e.g. [KH01, Kri01]). The AST is the representation which can be considered the most sophisticated common denominator of the representations that are used or conceivable. Therefore, this representation is likely to retain as much information as possible while minimizing the effort necessary to compare results.

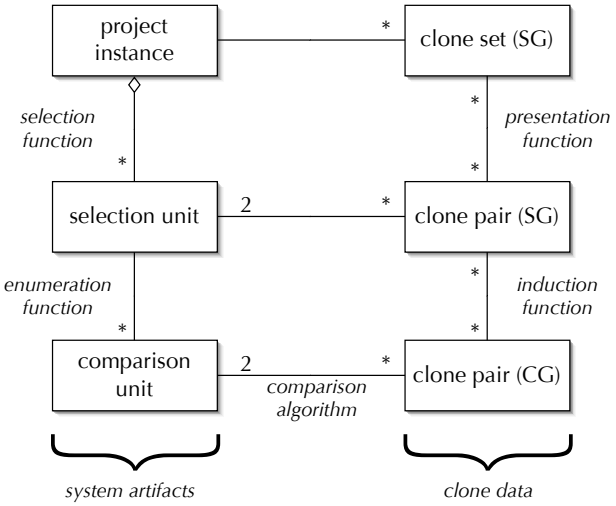

Figure 5: Overview of the generic clone model

4.5 Summary

An overview of the elements of the clone model is given in figure 5, which summarizes the previous discussion. Firstly, there are elements directly corresponding to system artifacts. Secondly, there are artifacts representing parts of the clone data which is generated by a clone detection algorithm on the basis of the system artifacts.

The top-level artifact representing the whole system is called *project instance* pi from some base set PI. An instance is structured into *selection units* and *comparison units*, through a *selection function* Sel and an *enumeration function* Comp, respectively. Given PI, we define SU(PI) and CU(PI) as the base sets of selection and comparison units, respectively. The clone data is aggregated on the top level in clone sets, which contains clone pairs. Clone pairs exist in two granularities, that of the selection units (SG) and of the comparison units (CG).

A *plain clone detection algorithm*, which neglects the level of detail of the comparison granularity, is a mapping $\mathrm{Alg}\colon \mathrm{PI} \to \mathrm{CR}$, where CR is the set of all binary, symmetric and reflexive (clone) relations over SU(PI). Alg must obey

$$\forall \mathrm{pi} \in \mathrm{PI}\colon \mathrm{Alg}(\mathrm{pi}) \subseteq \mathrm{Sel}(\mathrm{pi}) \times \mathrm{Sel}(\mathrm{pi})$$

and a partial mapping $g : \mathcal{P}(\mathrm{SU}) \to \mathrm{CR}$ must exist such that $\mathrm{Alg} = g \circ \mathrm{Sel}$.

Clone pairs are grouped into clone sets by a *presentation function*. The clone sets used here are sets with a distinguished reference element, i.e. the basic set of clone sets is determined by

$$\{(x, \{(x, y) \mid x \simeq y\}) \mid x \in \mathrm{SU} \wedge (\exists\, y\colon x \simeq y)\}$$

This set is further restricted to reduce redundancy. Given a clone set $C := (x, \bar{C})$, the elements of $\{y \mid \exists (x', y') \in \bar{C}\colon y = y'\}$ are called the *elements of* C.

Because of its definition, a clone set C makes no assumption about the clone relations among its elements, only about the clone relation between its

distinguished element and its other elements. Plain sets with a cardinality larger than two cannot be used to represent the clone relation, if the clone relation is not transitive [Bel02].

5 Uses of the model

Of course, the question arises what the representations within the model can be used for. Considering the underlying example code fragment (fig. 1) again, one might come to the conclusion that the duplication can be removed by factoring out the common parts and parameterizing it using the Strategy pattern [GHJV95]. In fact, a similar approach has been undertaken as described in [BMD+99b, BMD+00].

An advantage of having a generic clone model in this context is that it is possible to separate concerns regarding clone detection, clone description and clone management, including clone removal. Previous approaches to clone detection do not address clone description explicitly. As a consequence, approaches to clone removal have to address clone detection and clone description without clearly separating these issues from the primary goal of clone removal techniques.

Tool support

In order to effectively realize this separation of concerns, tool support for the three aspects is required. We have developed an implementation framework for such tools and a prototype tool on top of that.

Besides the clone model, we have also developed concepts for clone management as part of our research. These concepts are independent of the programming language considered and the tool platform used. Our approach to clone management can be characterized as decentralized, using local data. Clone management activities are triggered by the developer as tool user and as side-effects of other actions. Both system-focused and clone-focused scopes of actions are supported, with an emphasis on the latter. The primary mode of operation is development-accompanying. Details on the characteristics of our approach will have to be subject to another paper.

Since code clones can be found in systems of any kind and size, we see a need for tools supporting clone management within a widely-used IDE. Based on the generic clone model and the clone management concepts as a foundation, we have developed a framework on top of the open source tool platform Eclipse. The framework minimizes the effort necessary to implement new clone detection algorithms. To enable tight integration into the development process and particularly immediate display of detected clones, incremental operation is supported.

A prototype implementation offering this functionality is available under [Gie03]. A screen-shot of the prototype is shown in figure 6.

6 Related work

Beginning in the early 1990s several algorithms for automatically identifying duplicated source code fragments (code clones) have been developed [Bak92, BYM+98, DRD99, KH01, KDM+96, Kri01, MLM96]. Recent efforts are

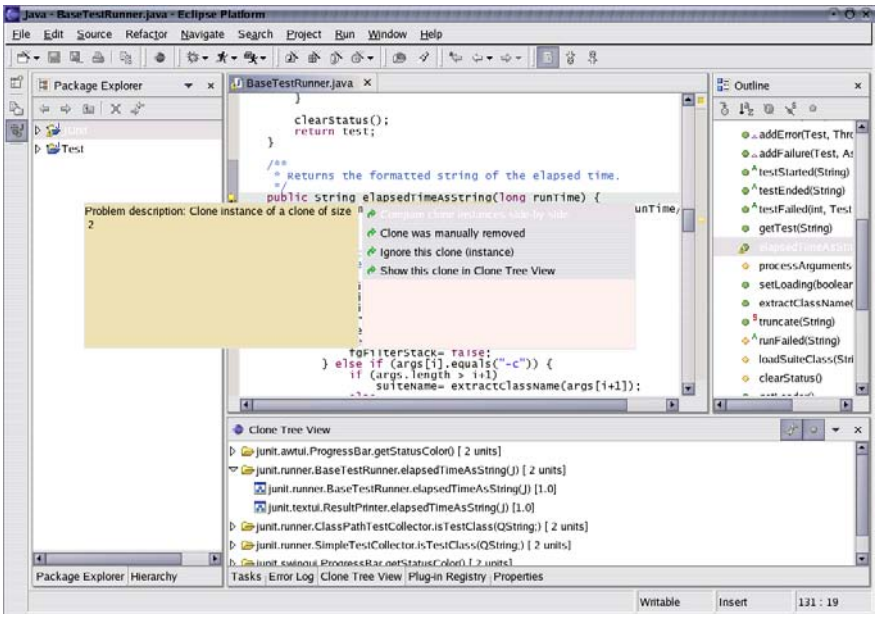

Figure 6: Tool screen-shot

targeted at evaluating the quality of these algorithms [Bel02] and defining common terminology for clones. The latter is a prerequisite to defining benchmarks [LLWY03].

The aspect of tool support for clone management has only slightly been approached. Many tools were created as part of the work on methods for clone detection (see above) and removal (part. [BMD+99b, BMD+00, BYM+98]). But this work was driven by the operational aspects of detection and removal, and was not done from a tool-centered perspective. Among the notable exceptions are Ariadne [KDM+96], integrated into a reverse engineering environment, and Gemini [HUK+02]. But, thoughts about separation of concerns were not made explicit, resulting in little systematic reuse. Much more, to our knowledge there has been no concept or tool allowing use of different clone detection algorithms within one conceptual or implementation framework. Some work has been done on clone visualization techniques [DRD99], which were not considered directly in our work, but may be realized on top of the tool framework.

Regarding the integration into the software development process, concrete clone management activities have been proposed earlier [MLH96], and their potential effect has been evaluated using historical data, i.e. several versions of a large software system. Technical foundation of this approach is a central repository which conducted the clone detection, together with special clients performing corresponding user interaction. This scenario requires a proprietary, specialized environment which is suitable for development of large software systems, but not for small to medium-sized projects where off-the-shelf IDEs are used. No tool based on this scenario has been actually realized according to [MLH96].

The concepts of selection granularity and comparison granularity were introduced by [KDM+96], but were not further developed therein.

7 Conclusion

In this paper, we have analyzed the notion of code clones, with respect to their effects on software qualities, the circumstances of their emergence in software systems, the particularities of inexact clones and general thoughts about clone avoidance. Then, we have defined the notions of selection granularity and comparison granularity. On this basis, we have defined a generic clone model, which is independent of programming languages, and is uniformly applicable independent of the choice of selection and comparison granularities. We have discussed issues to be considered in such a choice. In subsequence, we have illustrated the use of the model using a concrete example in the Java programming language. Finally, we have discussed possible uses of the model, and presented a framework and prototype implementation allowing implementation of tools adhering to the model.

Future work

Concerning the model itself it should be evaluated if the existing approaches to code clone detection can be adequately mapped to the proposed model. On the other hand, it should be studied in practice if the abstractions of the model are helpful to software engineers trying to develop and/or restructure a software system.

To allow technically effective separation of concerns which is independent of the tool implementation used, an open language for exchanging clone data between tools should be defined based on the generic clone model. This also enables shared distributed use of one clone data basis.

The basic tool currently available [Gie03] can be used as a starting point for making clone data described using the model available to software developers in a common environment. It is clear that, apart from the suitability of the model, the adequateness of the tool support is a dimension worth to be studied on its own. As already mentioned, the inclusion of clone management into the development process has not been studied in depth yet; neither has appropriate tool support for resulting process activities. Apart from that, an appropriate tool can be used for assessing the usefulness of the results produced by different clone detection algorithms in daily developers' work.

References

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bel02] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diploma thesis, Universität Stuttgart, September 2002.

[BF00] Kent Beck and Martin Fowler. *Extreme Programming Explained: Embrace Change*, chapter Bad Smells in Code. Addison-Wesley, 2000.

[BMD+99a] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. 6th Intl. Symp. on Software Metrics*, pages 292–303. IEEE, IEEE Comp. Soc. Pr., 1999.

[BMD+99b] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proc. 6th Working Conf. on Reverse Engineering (WCRE)*, pages 326–336, Atlanta, Georgia, October 1999. IEEE, IEEE Comp. Soc. Pr.

[BMD+00] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. 7th Working Conf. on Reverse Engineering (WCRE)*, pages 98–107, Brisbane, Australia, Nov 2000. IEEE, IEEE Comp. Soc. Pr.

[Bro87] Frederic L. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. 20th Intl. Conf. on Softw. Maintenance*, pages 368–377. IEEE, IEEE Comp. Soc. Pr., 1998.

[DDN02] Serge Demeyer, Stphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufman Publishers, 2002.

[DPML02] M. Dagenais, J. F. Patenaude, E. Merlo, and B. Lague. Clones occurrence in Java and Modula-3 software systems. In Hakan Erdogmus and Oryal Tanir, editors, *Advances in Software Engineering*, chapter 5, pages 95–110. Springer, 2002.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proc. Intl. Conf. on Softw. Maintenance*, pages 109–118. IEEE, 1999.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Longman, 1999.

[Ghe03] Carlo Ghezzi. *Fundamentals of Software Engineering*. Prentice Hall/Pearson, 2. edition, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[Gie03] Simon Giesecke. Eclipse duplication management framework project page, 2003. http://dupman.sf.net/.

[HUK+02] Yoshiki Higo, Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In Markku Oivo and Seija Komi-Sirviö, editors, *Proc. Product Focused Software Process Improvement (PROFES 2002)*, volume 2559 of *Lect. Notes in Comp. Sc.* Springer, 2002.

[KDM+96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1–2):77–108, 1996.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In P. Cousot, editor, *Proc. Static Analysis Symp. (SAS)*, volume 2126 of *Lect. Notes in Comp. Sc.*, pages 40–56. Springer, 2001.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. 8th Working Conf. on Reverse Engineering (WCRE)*, pages 301–309, 2001.

[LLWY03] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. In *Proc. 11th Intl. Workshop on Program Comprehension (IWPC)*, Portland, Oregon, May 2003. IEEE, IEEE Comp. Soc. Pr.

[LPM+97] Bruno Laguë, Daniel Proulx, Ettore M. Merlo, Jean Mayrand, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. Intl. Conf. on Softw. Maintenance*, pages 314–321. IEEE Computer Society Press, 1997.

[MLH96] Jean Mayrand, Bruno Laguë, and John Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *Proc. Workshop on Empirical Software Studies*, Monterey, California, USA, November 1996.

[MLM96] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. Intl. Conf. on Softw. Maintenance*, pages 244–253. IEEE, IEEE Comp. Soc. Pr., 1996.

[PC86] David L. Parnas and P. C. Clements. A rational design process: how and why to fake it. *IEEE Trans. Softw. Eng.*, 12(2):251–257, February 1986.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.