CloneTracker: Tool Support for Code Clone Management

Ekwa Duala-Ekoko and Martin P. Robillard
School of Computer Science
McGill University
Montréal, Québéc, Canada
{ekwa, martin}@cs.mcgill.ca

ABSTRACT

Code clones are generally considered to be an obstacle to software maintenance. Research has provided evidence that it may not always be practical, feasible, or cost-effective to eliminate certain clone groups through refactoring. This paper describes CloneTracker, an Eclipse plug-in that provides support for tracking code clones in evolving software. With CloneTracker, developers can specify clone groups they wish to track, and the tool will automatically generate a clone model that is robust to changes to the source code, and can be shared with other collaborators of the project. When future modifications intersect with tracked clones, CloneTracker will notify the developer, provide support to consistently apply changes to a corresponding clone region, and provide support for updating the clone model. CloneTracker complements existing techniques by providing support for reusing knowledge about the location of clones in source code, and support for keeping track of clones when refactoring is not desirable.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Design

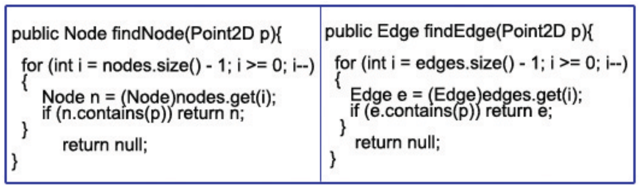
Keywords

Software Maintenance, Code Clone, Refactoring, Simultaneous Editing, Source Code Analysis

1. INTRODUCTION

Code clones — source code regions that match each other with varying degrees of exactness — have been reported to account for up to 30% of some large systems [1, 7]. Several reasons have been suggested for the prevalence of code clones in software systems, including the difficulty of factoring out functionality using programming language constructs, and the practice of writing code by example [9].

The presence of code clones in a system means that code that realizes identical or similar logic is not co-located. This duplication of implementation logic often leads to a necessity to modify multiple regions of code consistently. A recent study by Kim et al. found that up to 38% of code clones changed consistently with their counterpart at least once in their history [8]. Oversight to consistently apply changes to clones may introduce bugs into the system. For these reasons, much effort has been spent on the detection and removal (through refactoring) of code clones from software systems [4, 5, 7].



```
public Node findNode(Point2D p){
  for (int i = nodes.size() - 1; i >= 0; i--)
  {
     Node n = (Node)nodes.get(i);
     if (n.contains(p)) return n;
  }
       return null;
}
```

```
public Edge findEdge(Point2D p){
  for (int i = edges.size() - 1; i >= 0; i--)
  {
     Edge e = (Edge)edges.get(i);
     if (e.contains(p)) return e;
  }
     return null;
}
```

Figure 1: Example of Locally Unfactorable Clones

Unfortunately, refactoring code clones is not always feasible. Kim et al. further showed that, in one case, up to 64% of code clones that changed consistently could not be easily refactored. For instance, the clone relationship in Figure 1,[1] although identical in implementation logic, cannot be easily removed using standard refactoring techniques because the return types, and the objects referenced within the methods, are different. Even when feasible, immediate refactoring is not always beneficial since up to 72% of the code clones studied disappeared within an average of eight check-ins in a source code repository [8]. These observations indicate that, in certain situations, it might be beneficial to document and track clones as the system evolves.

Without tool support, developers must maintain a mental note of these clone dependencies, and keep track of them as the code base evolves. Consequently, existing clone relationships in need of consistent update might be overlooked. One possibility is to re-run the clone detection tool whenever the source code is modified to determine if the changes intersect with existing clone relationships. Clone detection is time consuming [3], and when completed, the developer must typically go through a non-negligible list of clone groups that possibly includes many uninteresting results. This problem

[1]Both methods from ...framework.Graph of the Violet UML editor source code.

is further compounded by the difficulty to reuse previously-generated data about clone locations since most clone detection tools describe clone regions in terms of line ranges, which are invalidated when modifications are made to code regions preceding the clone block.

To address these problems, we developed *CloneTracker*,[2] an Eclipse plug-in with support for robustly documenting and keeping track of code clones in an evolving code base. Our tool relies on clone documentation structures called *clone region descriptors* (CRD), which we proposed in a previous paper [3]. CRDs describes clone regions based on a combination of syntactic, structural, and lexical information. *CloneTracker* takes as input the output of a clone detection tool, and automatically produces CRDs to represent clone regions for different clone groups. With *CloneTracker* activated, clones are automatically tracked as the code evolves, the developer is notified when modifications intersect with the documented model, and support to simultaneous modify clone regions is provided. This way, software developers can specify clone groups they wish to track once and carry on with all their future modification tasks with the knowledge that modifications to clone regions will be detected and supported. *CloneTracker* also provides support for updating and sharing the clone model with project collaborators.

Although we introduced *CloneTracker* in a previous publication [3], this paper provides a detailed description of the most recent version of the tool, which is now released and which has been improved (e.g., integrating change notification with the Eclipse warning mechanism) and significantly expanded with new features such as the model updating support described in Section 2.3, and the support for sharing the model amongst collaborators described in Section 2.2.

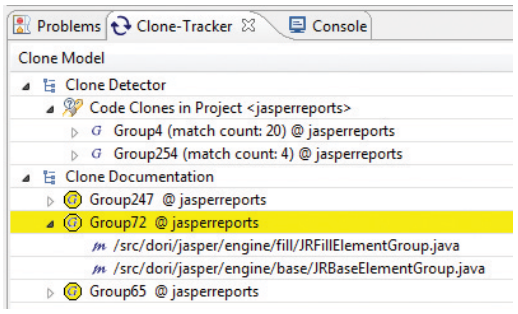
2. THE CLONETRACKER PLUG-IN

As an example, we describe a usage scenario in which a developer working on a modification to release 0.5.1 of JasperReport [3] (30kLOC) informally observes a number of code clones and decides to leverage the clone management support of *CloneTracker*.

2.1 Generating the Clone Model

To provide support for code clone management, *CloneTracker* needs as input the clone relationships to document and monitor in a given code base. It relies on a clone detection tool that scans the source code of a system and identifies code regions of varying similarity represented as clone groups. The current version of *CloneTracker* relies on SimScan,[4] but can easily be adapted to use other tools.

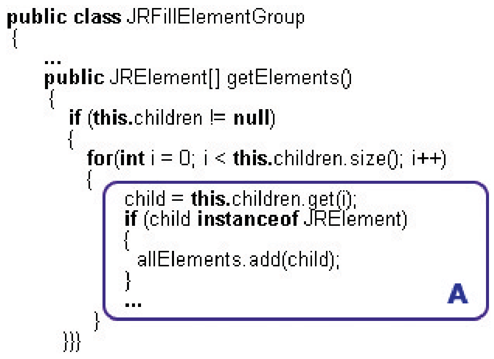
Using *CloneTracker*, the developer sets a number of search options for SimScan (Volume="medium", Similarity="fairly similar", Quality/speed="fast") and runs the tool. After approximately 21 minutes (WindowsVista, Intel Core2Duo-2GHz, 2GB), the detection completes and returns a list of 258 *clone groups* comprising between 2 and 35 *clone regions* (or individual clones). The output of Simscan is stored in a comma-separated-values file, and each clone region is represented in terms of a file name and a line range. To show the results of SimScan, and to allow the developer to indicate which groups to track, *CloneTracker* provides a view with two top-level nodes (Figure 2). The results of the clone detection tool are displayed as children of the *Clone Detector* node, and the documented clone groups as children of the *Clone Documentation* node.



```
Clone Model
◢ Clone Detector
  ◢ Code Clones in Project <jasperreports>
    ▷ Group4 (match count: 20) @ jasperreports
    ▷ Group254 (match count: 4) @ jasperreports
◢ Clone Documentation
  ▷ Group247  @ jasperreports
  ◢ Group72  @ jasperreports
      /src/dori/jasper/engine/fill/JRFillElementGroup.java
      /src/dori/jasper/engine/base/JRBaseElementGroup.java
  ▷ Group65  @ jasperreports
```

Figure 2: Clone Documentation View

Browsing the results, the developer notices a clone group of potential interest (`Group72`): in class `...engine.base.JRBaseElementGroup`, a large `for` block in method `getElements` is a clone of a similar region in method `getElements` of class `...engine.fill.JRFillElementGroup`. The developer observes that these clone regions, although identical in control logic and return type, cannot be easily refactored because they manipulate different types of objects. To generate the clone model, the developer transfers `Group72` from the *Clone Detector* node to the *Clone Documentation* node through a drag-and-drop operation. *CloneTracker* then automatically translates the location (i.e., file name and a line range) of the clone regions in the group into CRDs. A CRD describes a clone region based on the characteristics (syntactic, structural, and lexical) of each block in which it is enclosed. For instance, the CRD for code block A in Figure 3 is:



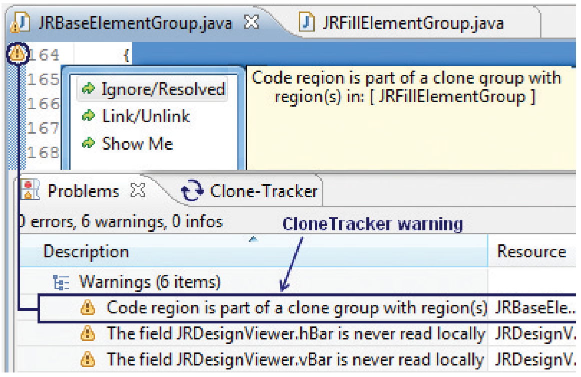
```
public class JRFillElementGroup
{
  ...
  public JRElement[] getElements()
  {
    if (this.children != null)
    {
      for(int i = 0; i < this.children.size(); i++)
      {
        child = this.children.get(i);
        if (child instanceof JRElement)
        {
          allElements.add(child);
        }
        ...
      }
}}}
```

Figure 3: Block Represented by a CRD

```
...engine.fill/JRFillElementGroup.java,JRFillElementGroup
getElements()
if,this.children!=null
for,this.children.size()
```

In other words, this CRD points to the block corresponding to the `for` statement with the "...children.size..." termination predicate, nested within the `if` statement with the "...children!=null..." predicate, within the scope of the `getElements` method, etc. The clone model now describes clones in a way that is resilient to changes in code blocks preceding the documented clone regions, or changes within the clone regions

[2]CloneTracker is available at: www.cs.mcgill.ca/∼swevo/clonetracker
[3]jasperforge.org
[4]blue-edge.bg/simscan

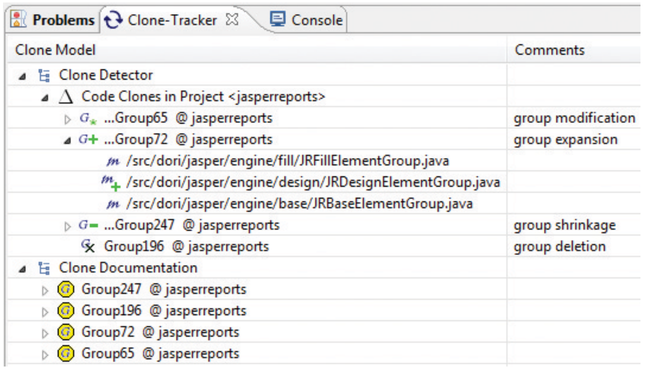


Code region is part of a clone group with region(s) in: [JRFillElementGroup]

0 errors, 6 warnings, 0 infos

Description	Resource
Warnings (6 items)	
Code region is part of a clone group with region(s)	JRBaseEle..
The field JRDesignViewer.hBar is never read locally	JRDesignV.
The field JRDesignViewer.vBar is never read locally	JRDesignV.

Figure 4: Change Notification in CloneTracker



Clone Model	Comments
Clone Detector	
Code Clones in Project <jasperreports>	
...Group65 @ jasperreports	group modification
...Group72 @ jasperreports	group expansion
/src/dori/jasper/engine/fill/JRFillElementGroup.java	
/src/dori/jasper/engine/design/JRDesignElementGroup.java	
/src/dori/jasper/engine/base/JRBaseElementGroup.java	
...Group247 @ jasperreports	group shrinkage
Group196 @ jasperreports	group deletion
Clone Documentation	
Group247 @ jasperreports	
Group196 @ jasperreports	
Group72 @ jasperreports	
Group65 @ jasperreports	

Figure 5: Updating the Clone Model

themselves. A detailed description of the CRD model, the related algorithms, and an evaluation of the approach, can be found in a previous paper [3]. To support team collaboration, *CloneTracker* persists the clone model within the project under investigation, which can then be shared with other developers. On start-up, the tool automatically detects and loads the model.

2.2 Change Notification

Some time in the future, a different developer (with access to the clone model) is assigned a modification task that involves the method `getElements` of class `...engine.base.JRBaseElementGroup`. Unbeknown to him, this code region is part of a clone group that requires consistent modification. Without dedicated tool support, the developer can either repeat the clone detection and investigation process that was performed by the other developer, or risk introducing bugs by modify only this clone region.

With *CloneTracker* activated, the developer is given immediate notification upon modifying the clone region. Change notification in *CloneTracker* is integrated with the Eclipse warning mechanism. Our plug-in adds a warning to the Eclipse Problems View (for easy access to the clone region), and attaches an Eclipse warning marker at the beginning of the clone region. The message of the warning describes the clone group to which the modified region belongs (Figure 4). In our example, the developer is informed that the modified code region has a cloning relationship with a region in the class `...engine.fill.JRFillElementGroup`, and is provided three QuickFix options.

The *Show Me* option points the developer to the group to which the clone region belongs by highlighting the background of the row in the *Clone Documentation* node to yellow (Figure 2). The number of documented clone groups may be non-trivial, and this functionality effectively eliminates the need to repeat the clone detection and investigation process. Once identified, the developer can reason about the cloning relationship of the group (e.g., to determine if consistent modification is necessary), and to consider refactoring.

The *Link/Unlink* option provides support for consistently modifying clone regions when necessary. When selected, *CloneTracker* opens a corresponding clone region, and modifications made within common sub-regions of both clone regions are echoed from the active clone region to the sibling clone region. Details of the simultaneous editing algorithm, its evaluation, and limitations are discussed in our previous paper [3].

The *Ignore/Resolved* option is used to inform *CloneTracker* to ignore the clone region. Once selected, the plug-in removes the marker from the clone region and from the Eclipse Problems View for the duration of the Eclipse session. Modifications to regions of this clone group during the current session are not communicated to the developer.

2.3 Updating the Clone Model

Future modifications such as the copy-and-paste reuse of a clone region, or its elimination, may invalidate the state of the documented clone relationships; hence, periodic updates of the clone model are necessary for it to remain accurate. To provide this functionality, *CloneTracker* maintains a set of all the source code files that were modified during a *development session*. We define a *development session* as the time between two updates to the clone model. The set of files modified during this session is called the *change-set*, and the list of files formed when the *change-set* is combined with the files tracked by the clone model is called the *delta*. To update the model, a developer invokes the clone detection tool with the *delta* as parameter. Once the clone detection is completed, the plug-in generates CRDs for all the clone regions identified in the *delta*, and performs a two-phase comparison against the CRDs in the model to determine the status of the documented clone relationships.

Determining Clone Group Status. In the first phase, the plug-in compares past and current clone group information, and assigns each group in the model a status based on the results of the comparison. The status *exists* is assigned to documented clone groups that were found in the *delta*, and the status *disappeared* to those that were not. Disappearance may be due to refactoring or divergence of clone regions. *CloneTracker* identifies disappeared clone groups with a *Gx* icon (Figure 5). To identify new clone groups not in the model, the developer would have to repeat the process explained in Section 2.1.

Group Evolution Pattern. In the second phase, the plug-in determines how clone groups with status *exists* have changed in the *delta*. This is accomplished by comparing clone groups in the model against their counterparts in the *delta* (Figure 6). The changes are described using a modified version of the evolution pattern of clone groups that was introduced by Kim et al. [8].

- *Group Unchanged*: all the clone regions of the group remained unchanged in the *delta*, and no new regions were introduced.

- *Group Expansion*: at least one new clone region was introduced in its counterpart in the *delta* (Figure 6). For example, in Figure 5 (`Group72`), the developer is informed that the clone region in our working example was cloned in class `JRDesignElementGroup` in the *delta*. *CloneTracker* identifies such groups with a *G+* icon, and the newly created clone regions with *m+*. Our tool does not only inform the developer which clone groups have changed, but also how each group changed.

- *Group Shrinkage*: at least one clone region does not exist in its counterpart in the *delta*. For example, a clone region was refactored or diverged from the rest of the group. *CloneTracker* identifies such groups with a *G-* icon, and the missing clone regions with *m-*.

- *Group Substitution*: an equal number of clone regions are found in the *delta*, but some regions are not the same as in the documented model. *CloneTracker* identifies such groups with a *G** icon, the substituted region with *m-*, and the replacement with *m+*.

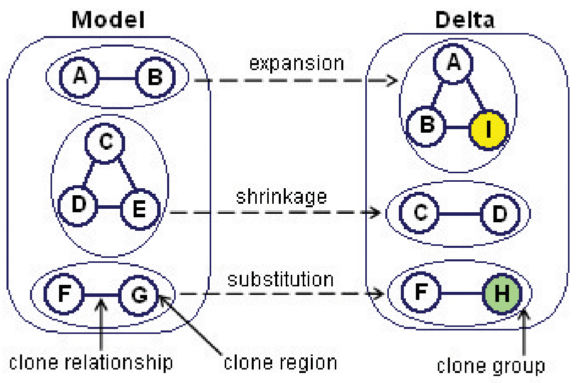


Figure 6: Evolutionary Patterns of Clone Groups

Once completed, the developer can update the model to reflect the desired status of the clones being tracked (e.g., the developer would update the clone relationship of `Group72` with the new relation in the *delta*).

3. RELATED WORK

Jablonski et al. proposed a tool for preventing errors that might be introduced when renaming identifiers in code regions formed through copy-and-paste operations [6]. Their tool captures copy-and-paste operations, groups common identifiers within the clone region from the AST of the code base, and provides support for consistently modifying a group of identifiers. Clonescape [2] is another clone management approach based on intercepting copy-and-paste operations, and generating clone relationships during a programming session. With this approach, the copied and pasted code forms, what the authors have called, a clone family, and are forever linked even when they eventually diverge. To compensate for diverging clone families, the tool provides a similarity metric that indicates how far the child has diverged from its parent. Clonescape also provides support for change notification. Both tools describe clone regions based on file name and line ranges, and are therefore unable to handle clones introduced outside an IDE extended by their tool. In addition, their tools' reliance on copy-and-paste actions to capture clone relationships implies they cannot be applied to existing source code. In contrast, *CloneTracker* describes clone regions using a combination of the structural, syntactic, and lexical information of their enclosing code block, and is therefore more resilient to changes than simple file name and line ranges descriptions. In addition, the use of external clone detection technology in *CloneTracker* implies it can be applied to both existing and future source code.

The interested reader can find a discussion of the work related to clone detection, *Clone Region Descriptors*, code clone analysis, and simultaneous editing in our previous paper [3].

4. CONCLUSIONS

The elimination of code clones through refactoring is not always feasible or cost-effective. We described *CloneTracker*, an Eclipse plug-in that provides support for code clone management in evolving software. Given the output of a clone detection tool, *CloneTracker* automatically generates a clone model based on *Clone Region Descriptors*, and provides support for change notification and simultaneous editing when future modifications intersect with tracked clones. *CloneTracker* is intended to complement refactoring and clone detection tools by providing support for reusing the knowledge acquired during clone detection and investigation activities, and the long-term management of clones when refactoring is not desirable.

Acknowledgments

The authors are grateful to Barthélémy Dagenais and the anonymous reviewers for their valuable comments on this paper. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

5. REFERENCES

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, pages 86–95, 1995.
[2] A. Chiu and D. Hirtle. Beyond clone detection. www.cs.uwaterloo.ca/∼dhirtle/publications-/beyond_clone_detection.pdf.
[3] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, 2007.
[4] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Addison-Wesley, 2000.
[5] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement*, pages 220–233, 2004.
[6] P. Jablonski and D. Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the Eclipse Technology Exchange at OOPSLA*, 2007.
[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
[8] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, 2005.
[9] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 69–73, 1989.