Insights into System–Wide Code Duplication

Matthias Rieger, Stéphane Ducasse, and Michele Lanza
Software Composition Group
University of Bern, Switzerland
{rieger,ducasse,lanza}@iam.unibe.ch
WCRE'2004

Abstract

Duplication of code is a common phenomenon in the development and maintenance of large software systems. The detection and removal of duplicated code has become a standard activity during the refactoring phases of a software life-cycle. However, code duplication identification tends to produce large amounts of data making the understanding of the duplication situation as a whole difficult. Reengineers can easily lose sight of the forest for the trees. There is a need to support a qualitative analysis of the duplicated code. In this paper we propose a number of visualizations of duplicated source elements that support reengineers in answering questions, e.g., which parts of the system are connected by copied code or which parts of the system are copied the most.

Keywords: Code duplication detection, code visualization, polymetric views.

1. Introduction

Code duplication detection has received increased attention from the reverse engineering research community in the last decade. Many detection methods are being investigated, from lightweight to sophisticated ones [1], [7], [12], [15], [3] and compared against each other [4]. A main goal is to automate as much as possible not only the detection but also the removal of the duplication [2]. This is not always possible, either because the copied code has diverged too much or that applicable refactorings are not straight forward. In either case, manual investigation is required.

The problem is that duplication detection approaches report large amounts of data that must be treated with little tool support. For industrial systems a duplication rate of 5-10% is considered a low but common estimate. This means that in a system of 1 million lines of code, 50'000 to 100'000 lines of code are involved in duplication. If we assume an average length of about 25 lines for a copied fragment, we get a minimum of 1000 to 2000 clone pairs that have to be investigated[1].

The engineers charged with duplication investigation and removal are subject to the usual time and cost constraints of an industrial setting. They most likely do not have the resources to remove every last instance of duplication from the system but have to prioritize and decide which clones to remove. To do so, they have to

- assess the system regarding the occurrence of duplication, *i.e.*, get a "mental picture" of the redundancy situation.
- identify and select duplication that is "problematic" or "worthwhile to refactor". This includes, for example, large fragments that have been copied multiple times but eventually also duplication that is easy to refactor.

Moreover, the engineers need to process the duplication data in an organized way by prioritizing the investigations they must perform. For example one way is to start with the largest clones or the ones involving the most source files, or the ones where a refactoring would have the most impact. Since the ultimate decision on whether to refactor or not usually involves a manual investigation of the source code, the information presentation must be interactive and connected to the underlying code, to allow for short examination cycles.

In this paper we propose to apply polymetric views [14] to the context of duplicated code, *i.e.*, we visualize duplicated elements of different levels of abstraction and enrich the views with metrics that present qualitative information of these abstractions.

2. Visualizing Duplication Data

Our approach to support the understanding of duplicated code is based on data visualization. According to Ware [16],

1 Note that in this paper we are not interested in the method of clone detection and assume that the problem of false positives has been solved.

visualization is the preferred way of getting acquainted with and navigating large data pools. Duplication data is relational data: two source code entities are related by shared pieces of code. A natural way of expressing these relations is a graph: the nodes of the graph represent the source entities whereas the edges of the graph represent the duplication relations.

2.1. Entities and Relationships

Our hypothesis is that when investigating a system affected by duplication, our mental model basically consists of the following elements:

The *source entities* represent (fragments of) the source code. In the context of this paper we use files as source entities. Other entities such as subsystems, modules, classes, and methods can be used as well.

The *duplication relationships* connect the source entities.

The *duplicated fragments* are the source code that two (or more) source entities have in common.

Since the investigated systems are of huge proportions (millions of LOC), data growth reaches unwieldly amounts (thousands of clones). Would we visualize all individual clones, we would get views where overplotting of nodes and especially of edges are hindering interpretation. To achieve scalability we therefore aggregate related clones into higher level entities. We define three duplication entities which form a containment hierarchy. Each higher order entity aggregates lower level entities.

We identify the following duplication entities as illustrated by Figure 1:

1. **Clone Pairs:** The lowest level of detail on which to describe duplication is the *clone pair* `<a,b>`. The pair comprises two source code fragments `a` and `b` which are copies of each other.
2. **Clone Classes:** A clone class is the union of all clone pairs which have source fragments in common[2]. For example, if we have the clone pairs `<a,b>` and `<b,c>`, it is likely[3] that there is a clone pair `<a,c>`. The *clone class* then encompasses the fragments `a,b,c`. The *domain* of a clone class is the set of source entities from which its source fragments stem. The domain of the clone class in the middle of Figure 1, for example, are the files *F11*, *F12*, and *F13*.
3. **Clone Class Families:** We group all clone classes that have the same domain to form a *clone class family*.

2 In [15] clone classes are called *clone communities*.

3 Note that for some clone relation definitions transitivity does not hold in general.

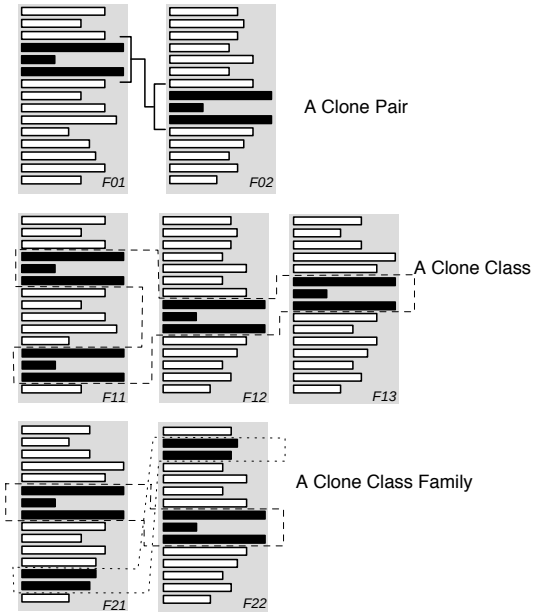

Figure 1. Containment hierarchy: Clone Pairs, Clone Classes, and Clone Class Families and the source files they are found in.

Note that the clone class family is not only a convenient way of reducing the entities and relations in the graph, it also holds direct interest for the reengineer: First, since clone class families contain only entire clone classes, they assemble all instances of a source fragment found in the system. Second, a clone class family reveals duplication activity that goes beyond the duplication of a single continuous source fragment. If two fragments, which were initially copies of each other, evolve differently over time, they may not be recognized as one clone pair any more (the "split duplicates" problem mentioned in [13]). The clone detector may identify smaller parts which are still similar individually. A clone class family will reunite those clone fragments. In summary, a clone class family aggregates all elements that are necessary to make informed decisions about refactoring measures for a particular fragment of copied code.

The visualizations we propose use the source files and the clone class families as entities. The decision not to display clone classes or clone pairs is due to scalability constraints. The lower level clone entities, *e.g.*, the clone classes and clone pairs which are the real targets of eventual refactoring operations, must however be reachable from their containers.

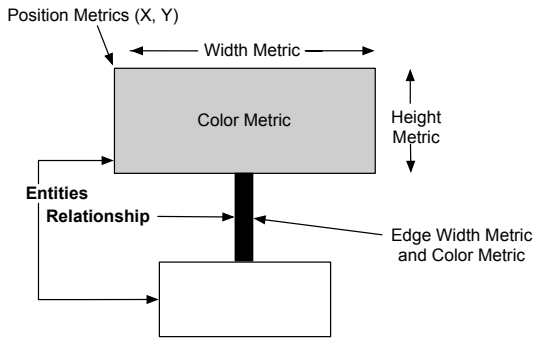

Figure 2. The principle of a polymetric view.

2.2. Polymetric Views

Polymetric views [14] are a visualization method for nodes–and–edges graphs enriched with semantic information such as metrics. Figure 2 illustrates how two-dimensional nodes representing entities, *e.g.*, software artifacts, and edges representing relationships can be enriched by software metrics: A node figure is able to render up to five metric values: in its width, height, x– and y–position, and in its color. An edge figure is able to visualize two metric values: width and color.

By applying metrics to the x– and y–position of the nodes, for example, similar entities are clustered close together in an easily identifiable region of the graph exhibiting some of their defining characteristics. Entities with differing characteristics are then placed in a distinct region of the graph. In this way, the shape of the visualized graph is able to communicate useful facts about the set of all visualized items.

2.3. Duplication Metrics

To discern between instances of code duplication we select a number of metrics that characterize the source files and clone class families (see Table 1). The choice of metrics is guided by the goal to create views that visually distinguish the entities in the view most effectively and intelligibly. The metrics are simple and can be computed from the results of any duplication detection tool without the aid of a parser.

The distinction between LIC and LEC is motivated by the possibly more complex situation that has to be understood when clone instances are located in different source entities. The smaller the amount of code that is involved in the duplication (the copied code *and* the surrounding context), the lighter is the cognitive load. Kapser and Godfrey [10] have proposed a clone taxonomy which is built on this distinction.

Source File Metrics	
Name	**Description**
LOC	**Lines of Code.** The size of a file is a common metric, despite its obvious drawbacks. It is immediately understood by every programmer and thus well suited to identify important files.
LCC	**Amount of copied code in the source file.** This is the central aspect we are interested in. This records every piece of code in the file that has been copied somewhere else in the system, including in the file itself.
LIC	**Lines of Code copied file–internally.** A subset of LCC, this metric records code that has been copied *within* a source file.
LEC	**Lines of Code copied file–externally.** Another subset of LCC. This metric records code that is shared with other files. Note that LIC and LEC are not necessarily disjunct.
Clone Class Family Metrics	
Name	**Description**
NSF	**Number of Source Files.** In how many source files are the copied fragments found? This is the set that defines the clone class family.
NCC	**Number of Clone Classes.** How many clone classes have been grouped together in a family? This says how many different source fragments are shared by all the files in the group.
LCC	**Lines of Copied Code.** How many lines of code does the clone class family encompass? For each clone instance that is part of the clone class family, the number of copied lines is summed up.

Table 1. Duplication Metrics for source files and clone class families.

2.4. Display Scalability

A well known problem in graph layouting is *overplotting, i.e.,* when too many nodes and edges are crammed on too little screen space, making a diagram unintelligible. Since we want to be able to display large datasets, we are forced to take precautions against overplotting. We employ the following techniques:

Reduction and filtering: By pooling related clones into clone classes and clone class families we reduce the size of the clone sets significantly. In the same manner source files can be combined into directories and subsystems.

Adaptive Graphical Representation: Since our views are intrinsically interactive, visual enhancements like highlighting can be triggered by roll–over mouse events. Multiple selections of nodes, *e.g.*, via their names or their connections are necessary as well to take advantage of the views.

3. Polymetric Views of Duplication

This section proposes a set of polymetric views that support the understanding of the duplication situation in a system and can guide refactoring measures. We order the de-

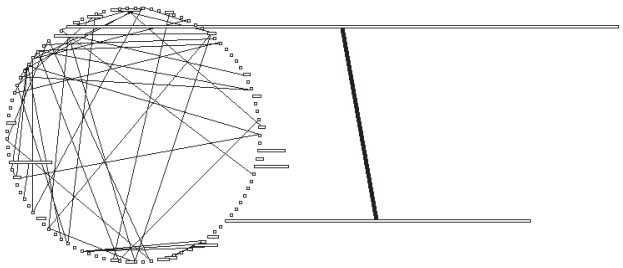

Figure 3. The Duplication Web of the MAILSORTING **system with** LIC **(Lines of file–internally Copied Code) as node width.**

scription of the views in a sequence that suggests a way for the engineer to walk through the task of understanding a system's duplication (a *reengineering roadmap*). After the discussion of each view we present a short overview of questions answered and potential further questions that are of interest at this stage. Each view is presented using the following schema:

Details. Gives a tabular technical description of the view, its entities, relations, and its layout.

Intention. Explains how the view can support the engineer in his tasks.

Symptoms. Details what kind of duplication problems the view reveals.

Examples. Shows sample views and explains their features.

Scaling. Investigates how the size of a system affects the view negatively and what can be done about it.

Overplotting. Investigates if the amount of data can cause overplotting problems and how they can be avoided.

3.1. The Duplication Web

The Duplication Web is the first view that an engineer can use as it introduces the user to the duplication situation. It shows all files in the system and all existing clone connections between them.

Nodes	Source Files
Edges	Clone Connections
Metrics	
Node Size	Height = – Width = LIC (Internally Copied Code)
Edge Width	LCC (Lines of Copied Code)
Layout	Nodes placed on a circle; Nodes with many connections are placed apart on the diameter.
Examples	Figure 3,Figure 4

Intention: This view gives an impression of the number of files in the system and the amount of duplication that connects them. It shows the entire system at once in a well defined shape that is independent of the physical organization.

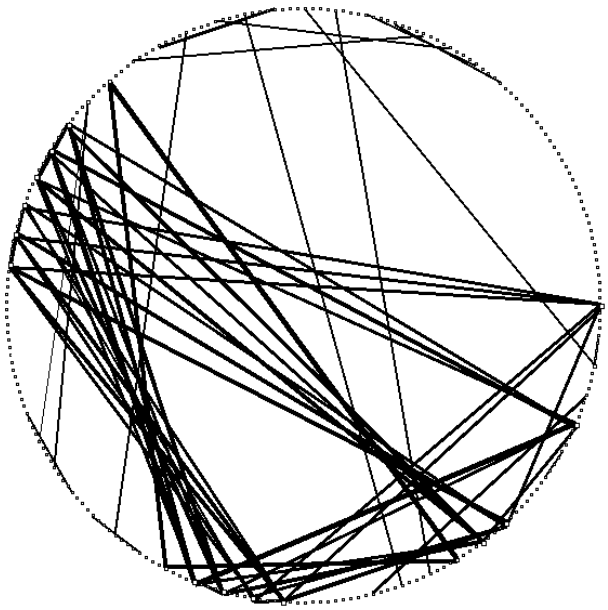

Figure 4. The Duplication Web view of MFC**. Setting heavily–connected nodes apart on the diameter emphasizes the overall amount of duplication connections.**

It improves on a textual report detailing all clones detected in a system.

Symptoms: The view reveals the following duplication problems in a software system:

- Wide nodes represent files that contain a lot of internal duplication.
- Thick edges connect files that share a lot of duplication.
- Nodes with many connected edges represent files sharing duplicated code fragments with many other files.

Example 1. Figure 3 of the MAILSORTING system shows 101 nodes, 57 of which share code with one or more other files. Most of the files are "copy–connected" to only one or two other files. The two files with the largest amounts of internal duplication also exchange the most external duplication.

Example 2. Figure 4 shows the application of the Duplication Web view to the Microsoft Foundation Classes (MFC). It is formed by 240 source files, 50 of which are connected by duplication links. In this variant, node size corresponds to number of connections. Following the edges one is able to divide the duplication activity of MFC into two larger groups of multiple interconnected files, and a few file pairs.

Scaling: The dimensions of the view can be controlled because of the fixed shape of the circle. For large numbers of

files the radius of the circle must be reduced, but its *gestalt* can still be recognized. If there are too many files, grouping them into directories, modules, or subsystems is helpful.

Overplotting: Since the intention of the view is to give an overview rather than to guide actual refactoring actions, the overplotting is not too problematic. Thanks to the fixed position for each node, overplotting can only become a problem if many nodes have a very high LIC value. If too many clone connections exist between the files, the edges in the center of the view will become impossible to distinguish. The view then only conveys the information that a lot of *copy&paste* programming has been going on.

Reengineering Roadmap Having gotten an impression of the duplication activity in general, we want to focus a bit more on the individual files. Which are the files that are heavily duplicated, which are the ones where only a small part has been copied?

3.2. The Clone Scatterplot

The Clone Scatterplot displays the same nodes and edges as the Duplication Web but the layout takes into account the size and duplication metrics for each file. It has still overview character but enables informed selections since more information is included in the presentation.

Nodes	Source Files
Edges	Clone Connections
Metrics	
Node Position	X–Pos = LOC (Lines of Code) Y–Pos = LCC (Lines of Copied Code)
Edge Width	LCC (Lines of Copied Code)
Layout	Scatterplot
Examples	Figure 5

Intention: The Clone Scatterplot confronts the size of the files with the amount of duplication they contain. Files of different duplication levels can be identified by the region they are positioned in. The edges tell us if code is shared between large and small files, or between files of similar size. Heavily copied files can be selected for closer inspection.

Symptoms:

- Nodes on the left represent small files, while the ones on the right represent large files.
- Nodes at the top of the view represents files having little or no duplication.
- Nodes that are not at the top of the view but are unconnected represent files having only internal duplication.
- Nodes close to the $45°$ diagonal represent files containing a lot of duplication with respect to their size.

Examples: The *gestalt* impression that this view gives can be best observed with the scatterplot of the AGREP system

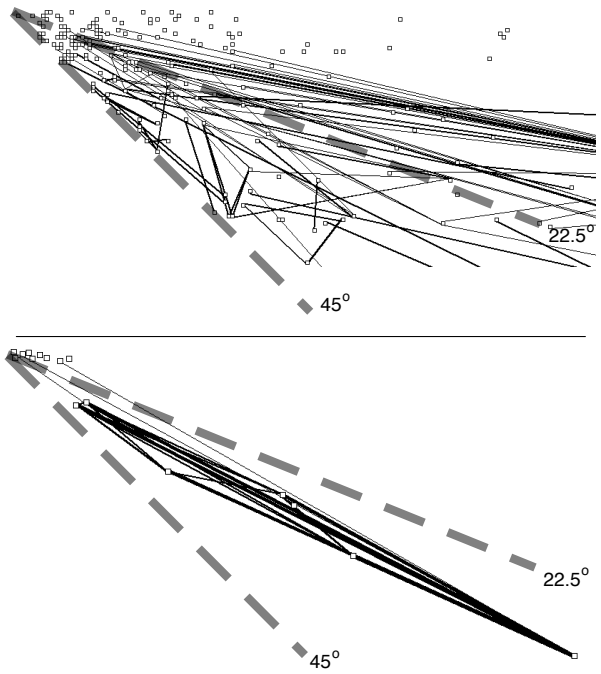

Figure 5. Two examples of the Clone Scatterplot: On top is an extract of the ACCOUNTING system. Below, the entire AGREP system. Both views are overlaid with lines indicating duplication rates of 100% ($45°$) and 50% ($22.5°$).

in the lower half of Figure 5. Here the system has very little variation around the main diagonal. This indicates that the level of duplication is equally high in all of the larger files. The largest file has common code with all the other files involved in external duplication.

Scaling: Since we use the LOC metric as X–Position, the view can grow very large when files contain a lot of lines. Logarithmic scales can then be applied to the X– and Y–metrics.

Overplotting: Thanks to the use of the LOC metric as X–Position, the source files are spread out over the view area, ameliorating the overplotting situation for the nodes. Smaller files without duplication are clustered in the upper left corner of the view, frequently overplotting each other. Since these files are not interesting for the user the problem does not have any impact. Clone edges tend to overplot quickly around the $45°$ diagonal, where the files with high duplication rates are located.

Reengineering Roadmap Until now the views only contained nodes representing individual files. Files are how-

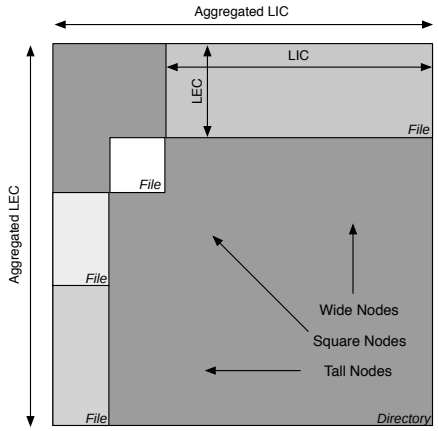

Figure 6. Node placement in the Treemap. Nodes are separated by their shapes and arranged so that the values of LIC and LEC are aggregated horizontally and vertically, respectively.

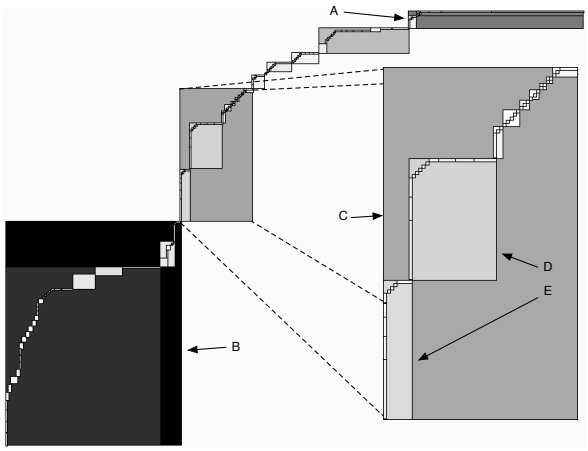

Figure 7. The tree map of the APACHE system. The rectangle on the right marked C is an enlargement of the second top–level node from left.

ever part of organizational system structures. We want to know how these larger entities are affected by duplication. This raises the abstraction level and we get the useful side effect that we can connect gained duplication knowledge more easily with the fewer elements of the coarse system structure.

3.3. The Duplication Aggregation Tree Map

This view aggregates the duplication that until now we have only seen attached to individual files. It shows the entire system top–down along the directory structure, annotating each directory node with the recursively aggregated amounts of internal and external duplication of its files and subdirectories. The view emphasizes system parts according to their involvement in duplication.

Nodes	Source Files, Directories
Edges	–
Metrics	
Node Size	Height = LEC (Externally Copied Code) Width = LIC (Internally Copied Code)
Node Color	LCC (Lines of Copied Code)
Layout	Modified Tree Map. Nodes are arranged according to the principle illustrated in Figure 6. The main difference to traditional tree maps is that empty space is allowed.
Examples	Figure 7

Intention: The tree map aims to give an overview of the ratio of internal to external duplication, aggregated from the individual source files up to the root directory of the system. The parts of the system which exhibit high amounts of duplication can be identified at a glance from the top level. Relative comparison of structures in the hierarchy is made possible. The view has a *gestalt* property, *i.e.*, it can give useful information immediately.

Symptoms:

- Nodes towards the lower left have increasing amounts of external duplication.
- Nodes towards the upper right have increasing amounts of internal duplication.
- Nodes in middle have no duplication or equal amounts of both kinds.
- Wide nodes have more internal than external duplication and vice versa.

Note that node height shows the sum of externally copied code only with regard to files. If two files within a directory D share some code, this amount will be aggregated as LEC for the node representing D, even though the code is not copied to files external of D.

Examples: From the shape of the overall diagram in Figure 7 we can determine that there is a bit more internal duplication than external duplication in APACHE. The rightmost node A representing the directory `lib` contains the most internal duplication, whereas leftmost node B representing the directory `modules` and its subdirectory `standard` contain most of the external duplication. The directory `os` (represented by node C) contains two subdirectories `win32` (node D) and `netware` (node E) which have a similar amount of external duplication (possibly shared between them).

Scaling: Thanks to the fractal property of treemaps we can display systems of any size on every screen. Zooming provides an adequate instrument to navigate even very large

systems. Aggregation of data will provide useful information even at the highest level where the smaller details are not discernible any more. Contrary to traditional treemaps this variant visualizes two values in every node, resulting in some waste of screen space. The advantage over the traditional treemaps is that the display is less crowded while still showing every element of the tree.
Overplotting: The layout precludes all overplotting problems.

Reengineering Roadmap Having gained an overview of the parts of the system involved in duplication, we want to know details about the copying. Is code shared within directories only, or also across directory borders, even subsystem borders? These informations are interesting since they uncover functional relationships between system parts that may not be documented. Such knowledge can also further the understanding of the design of the system.

3.4. The System Model View

This view shows the directory structure of the application, or alternatively the inheritance tree, using the familiar tree layout.

Nodes	Source Files, Directories
Edges	Clone Connections, Directory Containment
Metrics	
Node Size	Height = LEC (Externally Copied Code) Width = LIC (Internally Copied Code)
Edge Width	Clone connections = LCC (Copied Code) Directory Containment = –
Layout	Spaced Tree
Examples	Figure 8

Intention: The System Model view shows the duplication within the physical location of files, *i.e.,* their directory structures or the classes and their inheritance relationships. It helps identifying problematic subsystems and functional connections between subsystems.
Symptoms:

- Small squared boxes represent files without internal or external duplication.
- Flat wide boxes represent files that contain a lot of internally duplicated code.
- Tall narrow boxes represent files sharing a lot of duplicated code with other files.
- Thick edges among tall boxes represent the amount of duplicated code exchanged between them.

Examples. In the upper half of Figure 8 the directory structure of the JBOSS system is the basis for the arrangement of the source files in the view. Internal and external duplication are the metrics that are shown. Files A and B, as well as C and D share code as indicated by the duplication link between the files, as well as by the similar shapes of the nodes. What can additionally be seen in Figure 8 is that A and B are

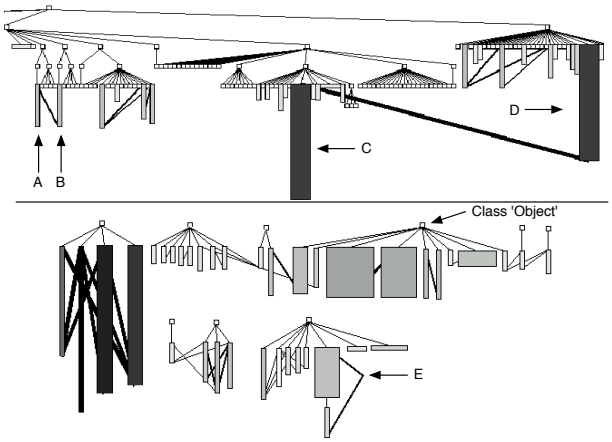

Figure 8. Two variant System Model views of JBOSS. The upper half shows part of the directory structure. The thicker edges represent clone relationships between files. The lower half shows extracts from the class hierarchy. Small squares represent superclasses defined outside of JBOSS.

located in sibling directories, whereas the duplication between C and D crosses 4 directories, *i.e.,* probably into another subsystem. This information is useful when deciding about refactoring measures. In the lower half of Figure 8 extracts from the class hierarchy of JBOSS are shown. On the left side, sibling classes copy heavily from each other. E marks a clone relation between a class and its superclass.
Scaling: The view becomes very large in a system with thousands of source files. Clone edges will likely go over the screen boundaries when connecting files in directories that are far apart, making good navigational features a necessity.
Overplotting: Trees are simple to layout without node overplotting. Displaying the clone edges can lead to serious overplotting problems, especially if the system model is a shallow tree.

Reengineering Roadmap Until now, our focus has been entirely on the files. We know their sizes, their locations and their connections. We now turn to an investigation of the connections, the code that is shared. How large is it? How many files has it been spread to? Are other common fragments copied along with it?

3.5. The Clone Class Family Enumeration

This view reduces the redundancy of the duplication connections that has been present in all the previous views. The

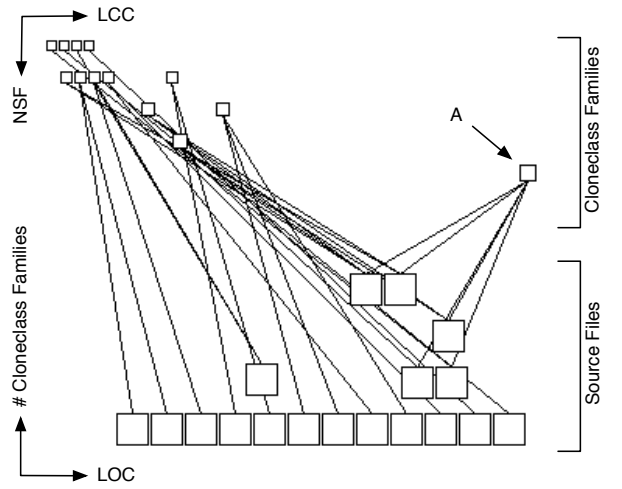

Figure 9. The clone class families of the MULTIMARKE **system.**

clones are shown in a concise nodes–and–edges view.

The layout uses the LCC and the LOC metrics to place clone class families and source files, respectively, on the horizontal axis. The intuition "the farther to the right the bigger" thus can be used to mentally classify both entity types presented in the view. The edges connect the upper half of the view - the cloneclass families - with the source files on the lower half.

Nodes	Cloneclass Families (CCF), Source Files
Edges	Participation in a Cloneclass Family
Metrics	
CCF Level	NSF (Number of Source Files)
CCF Position	X–Pos = LCC (Lines of Copied Code)
File Level	Number of clone class groups
File Position	X–Pos = LOC (Lines of Code)
Layout	
Upper half	Multiple levels of cloneclass families
Lower half	Multiple levels of source files
Examples	Figure 9

Intention: This view presents the clone class families to the user in a way that eases investigation of individual instances of duplication. It characterizes the families by the criterion of how many source files they comprise and how much code they contain. The user can start on a clone class family node and see which source files are participating. Or he can start with a source file node and see in how many clone class families the file participates. To make the view fully useful, lower level duplication entities, *i.e.*, clone classes and finally clones must be made available to the user via the nodes in this view.

Symptoms: Clone class families in the top rows are less important since they connect only a few source files. The families located on the rows towards the middle of the view have an increasing number of participating source files, which makes them interesting targets for investigation. Symmetrically, source files at the bottom of the view are only involved in a single clone class family, whereas files in the middle of the view are more interesting. Small files are to the left of the view and large files are to the right of the view.

Examples: Figure 9 presents 18 files and 13 clone class families which stand for 55 clone pairs (a 76% reduction of duplication entities). The largest clone class family A encompasses duplication in the 5 largest files, as can be seen from the figure. Clone class family A represents two clone classes—this means two different source fragments that are present in all 5 files—or 24 clone pairs.

Scaling: Clone class families or source files containing a lot of code are positioned at the far right, likely offscreen, which will require navigation.

Overplotting: The nodes must not overplot since the user has to be able to select from them. The layout mechanism thus arranges them side by side. Edge overplotting is of minor concern since the focus of the user lies on the nodes. Eventually, clone class families which represent only internal duplication in a single file could be removed from the view.

4. Discussion

The views achieve the goal of data reduction on different levels. We are able to display even very large systems on restricted screen space. Many of the views have a *gestalt* property, *i.e.*, they provide overview information *at a glance*.

The reduction of the cardinality of the clone sets, however, is sometimes not enough, resulting in cluttered displays which are hard to read. We must further support readability with interactive enhancements of the views, *e.g.*, with the highlighting of connected elements on mouse–over.

By using simple and heuristic layout mechanisms we provide a fixed arrangement of the nodes for all views except the System Model view. This is an advantage since there is no need for the user to rearrange the nodes to get a better view. This enables him to start *interpreting* the view immediately.

What is missing from the description in this paper is the necessary ingredient if the duplication is to be reengineered: making the source code of the clone instances reachable directly via the nodes and edges of the views by code browsers. This must be addressed by tool–builders.

That tool support is only one piece of the duplication refactoring puzzle in an industrial context is a fact which we have not included into our considerations. How business decisions and process questions affect the engineers in this matter will have to be addressed still.

5. Related Work

Visualizing Duplication with Graphs. Johnson [7] has used Hasse diagrams to visualize textual similarity between files. For each duplication–related cluster of files (a clone class family in our terminology) - the diagram shows the copied source text and the source files as nodes, and the inclusion relationships between the different code pieces as edges. The height of a node in the graph is determined by its size: large files or code fragments are towards the bottom, smaller pieces of code towards the top. His graph is similar to the clone class family enumeration proposed in Section 3.5.

In [8] Johnson proposes to navigate the web of files and clone classes via hyper-linked web pages. Although the entities and relationships that he defines are the same as we have used in this paper, his system lacks the overview and selection features that we think are necessary to find one's way in the mass of duplication data. His browsing system could however act as a backend for the views proposed in this paper.

Visualizing Duplication with Dotplots. A common method for visualizing duplication is the dotplot [6, 5, 9], where the lines of two source files are put on the two axes of a matrix and a dot is placed at each coordinate which represents two matching lines. Dotplots, however, have some drawbacks:

- Dotplots produce spacious images. The size of the image depends on the size of the input, not on the size of the duplication found.
- In a dotplot visualizing the comparison of multiple source entities there is no predetermined organization of the image. Some features may only be detected after rearrangement of the display.
- Dotplots contain a lot of redundancy. This can be overwhelming in the case of frequently repeated pieces of code.
- Dotplots give a detailed account of the duplication situation. As a consequence they convey overviews rather poorly.

Dotplots and polymetric views can be used as complementary duplication visualizations. The polymetric views are good as a starting point for the assessment phase. They give the user a *ToDo* list that has to be cleared point for point. Having selected a source file or a clone class, a targeted dotplot displaying only the clones belonging to the selected clone classes can be presented to the user for close inspection of the situation.

Visualizing Duplication in OO Class Hierarchies. Golomingi [11] has investigated how the information about the location of clones within an object–oriented class hierarchy can be utilized to decide upon refactoring measures. The focus of his work however was automation rather than visualization, *i.e.,* seeing the classes and their relationships was not the primary goal.

6. Conclusion

If a reengineer has to investigate and refactor duplication in a large system, he is in dire need of support for understanding and dealing with the potentially huge amount of copied code. To manage the overwhelming lists of detailed duplication information produced by duplication detection mechanisms, we reckon that he needs to (1) overview of the duplication situation and (2) navigate through the sea of information.

The approach discussed here is putting emphasis on the *'human in the loop'*, giving human expertise the helm, instead of pushing automatization. In this paper we have proposed first a way of grouping the duplication information into useful abstractions, *i.e.,* the clone class family which aggregates all duplication that is exchanged between a specific set of files. Second, we have proposed a number of polymetric views which structure the data and combine it with the knowledge about the system that the engineer already possesses.

We have only used a small and very simple set of metrics which can be computed without much investment in parsing infrastructure. Future work should include investigations of metrics or attributes oriented towards qualitative aspects of duplication. This will increase the selective capabilities of the views.

A pertinent problem is the overplotting of edges and nodes when systems and the amount of data get too large. We have proposed some aggregation abstractions to reduce the amount of data that must be presented on screen. Sophisticated filtering techniques should be the focus of tool development efforts if a visualization tool wants to be applicable to very large systems. Since the views greatly rely on their interactivity, this also means that they have a limited usefulness when committed to paper.

A. Example Systems

The sample systems we have used to produce the views in this paper are listed in the following table.

Name	Size		Language	Origin
	Files	LOC		
MAILSORTING	101	39'000	C++	Industry
MFC 4.2	245	107'000	C++	Industry
ACCOUNTING	336	22'000	COBOL	Industry
APACHE 1.3.20	141	65'000	C	Open Src.
JBOSS 2.3 BETA	403	35'000	JAVA	Open Src.
AGREP 2.04	22	12'000	C	Academia
MULTIMARKE	70	7'000	JAVA	Stud.Proj.

References

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

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings WCRE'00*, pages 98–107. IEEE, Oct. 2000.

[3] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM*. IEEE, 1998.

[4] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, Sept. 2002.

[5] S. Ducasse, M. Rieger, and G. Golomingi. Tool support for refactoring duplicated OO code. In *Proceedings of the ECOOP '99 Workshop on Experiences in Object-Oriented Re-Engineering*, June 1999. FZI-Report 2-6-6/99.

[6] J. Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. *TAPOS*, 2(1):31–41, 1995.

[7] J. H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of CASCON '94*, pages 9–18, 1994.

[8] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1996.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 28(6):654–670, 2002.

[10] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Proceedings of the First International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*. IEEE, Sept. 2003.

[11] G. G. Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, June 2001.

[12] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Journal of Automated Software Engineering*, 3:77–108, 1996.

[13] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings WCRE'01*. IEEE Computer Society, Oct. 2001.

[14] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE TSE*, 29(9):782–795, Sept. 2003.

[15] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of ICSM'96*, 1996.

[16] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.