Clone Detection & Cataloging Method (CDCM)
Towards an automatic approach for bootstrapping reuse efforts in an organization

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Rachit Sood
Graduate Program in Computer Science and Engineering

The Ohio State University

2012

Master's Examination Committee:

Dr Jayashree Ramanathan, Advisor

Dr Rajiv Ramnath
Abstract

There are a considerable number of cases in the past where organizations have succeeded in reusing code developed for customized business applications. The approaches taken had one thing in common: they looked back at what they had reused earlier (in existing applications) and identified components that had a high potential for reuse in the future.

Today, organizations pursue a code reuse initiative on similar lines and often try to provide a library with potentially reusable components to its developers. This process, however, is not easy, as it requires the organization to revisit "most of" its existing assets and look for code components that may have a high reuse potential in future projects, transform them to populate the reusable asset library.

We propose a method that eases the task of "filling-up" the library with pertinent components, and present the **Clone Detection and Cataloging Method (CDCM)**, which is an innovative approach that automates the analysis of existing source code. CDCM identifies reusable assets by finding semantic (common domains) and structural (redundant code) similarities in existing applications.

We evaluate our approach in the framework of commercial enterprise applications. Initial results are mixed and indicate that CDCM is useful for annotating source code components with business specific knowledge and finding reuse opportunities within an application.
Dedication

This document is dedicated to my family.
Acknowledgments

I would like to thank Dr. Jay Ramanathan for her motivation, support, guidance and patience throughout the work in this thesis. I would also like to thank Dr. Rajiv Ramnath, CETI and the teams at Nationwide ADC, for their help and enthusiasm, particularly during initial phases of this work. Their continued support and assistance, thesis related and otherwise are highly appreciated.
Vita

2010................................................................B.E. Information Science & Engineering

2010 to present ..............................................Graduate Research Fellow, Department of

Computer Science & Engineering, The Ohio

State University

Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Chapter 1: Introduction ...................................................................................................... 1
  Motivation ....................................................................................................................... 1
  Code Duplication .......................................................................................................... 1
  Goals ............................................................................................................................. 2
  Innovation ..................................................................................................................... 3

Chapter 2: Research Contributions ..................................................................................... 4

Chapter 3: Problem Statement ............................................................................................ 5
  Solution Approach ......................................................................................................... 5

Chapter 4: Background and Related Work ......................................................................... 7
  Software Reuse ............................................................................................................. 7
  Industry Reuse Efforts ................................................................................................. 9
  Code Clones ................................................................................................................. 12
  Reusable Asset Libraries .............................................................................................. 13
  Domain Analysis ......................................................................................................... 15
  Summary of Related Work ........................................................................................... 17

Chapter 5: Data Selection .................................................................................................. 19
Domain Analysis - Topic Modeling ................................................................. 35
Observations ........................................................................................................... 38
Validation .................................................................................................................. 38
Analysis ...................................................................................................................... 38
Clone Detection ......................................................................................................... 39
Outputs ......................................................................................................................... 39
Observations ..................................................................................................................... 44
Validation .......................................................................................................................... 44
Analysis ............................................................................................................................. 45
Chapter 9: Conclusion .................................................................................................. 46
Chapter 10: Future Work .............................................................................................. 48
References ......................................................................................................................... 50
List of Tables

Table 1: File memberships to topics ................................................................................. 35
List of Figures

Figure 1: Reuse oriented software development model.................................................... 20
Figure 2: CDCM Approach .............................................................................................. 22
Figure 3: Source Code Pre-Processed for LDA................................................................. 24
Figure 4: Strategy Design Pattern (Source: Wikipedia) ................................................... 31
Figure 5: Topic Modeling - 100 Topics – Project Memberships....................................... 37
Figure 7: Code clones in all projects – shows similarity between code sections over the file system ......................................................................................................................... 40
Figure 8: Sample Folder Structure.................................................................................... 41
Figure 9: Relating Folder Structure to Scatter Plot Axis.................................................. 42
Figure 10: CCFinderX Output – clone location between files .......................................... 43
Chapter 1: Introduction

For an organization contemplating an investment in systematic reuse, one key question is how to use existing software in that effort. Intuitively, it would seem that organizations should make use of their existing software. After all, existing software embodies a great deal of information about an organization's problem domains. This information should be very valuable in creating reusable assets.

Motivation

Existing research suggests that 10-15% of the source code in large scale applications is duplicated. Code duplication caused by copying source code and then using it by slightly modifying it, is observed in large software systems. While such code cloning may be unstructured, it is commonplace and is unlikely to disappear.

The very commonness of duplicate code suggests programmers should have tools that allow them to use previously developed code.

Code Duplication

The act of copying code indicates the programmer’s intent to “reuse” the implementation of some abstraction of code. Duplication of code is an example of unstructured reuse. Duplication reduces the time to develop parts of the product but it must be noted that
original development is only a small fraction of a product's life cycle, maintenance time and costs are much higher and duplication causes issues in maintenance. Some of the specific problems include:

- **Code bulk affects comprehension:** Code duplication frequently creates long, repeated sections of code that differ in only a few lines or characters. The length of such routines can make it difficult to quickly understand them.

- **Purpose masking:** The repetition of largely identical code sections can conceal how they differ from one another, and therefore, what the specific purpose of each code section is.

- **Update anomalies:** Non-observance incurs update anomalies, which increase maintenance costs, in that any modification to a redundant piece of code must be made for each duplicate separately.

**Goals**

The focus is to identify and recover a small percentage (5-10%) of valuable assets (clones) from a much larger portfolio using automated and partial process towards the ultimate goal of improving reuse (by populating a reuse library) within an organization.

We focus on “unstructured reuse” that has already been done via code duplication to recover valuable assets. Additionally we identify metadata that is relevant to assets; we propose that annotating each asset with such metadata would improve the accuracy of searches performed.

We propose an approach that is:
• **Automated:** We need to process a lot of material to find a small amount that may be useful. We cannot use labor intensive techniques to deal with that volume of material. Instead, we must use approaches that are as automated as possible.

• **Cost Effective:** Ideally the organization should not incur a high cost to follow this approach.

• **Partial:** To be cost-effective, the method must be selective based on a measure of usefulness, rather than to recover every fragment of reusable software.

**Innovation**

We define a “reusable asset” as a semantic and structural similarity in the entire source code assets of an organization. CDCM is a hybrid approach, in the sense that it focuses on extracting both semantic and structural similarities in source code.

While previous approaches primarily focused on “architecture reuse” based on semantics (domains), we found that the use of semantic analysis does not provide results at the granularity necessary for a code reuse library (source code components). We identified the need for structural similarities to be used in combination with semantic analysis; this would allow us to identify reusable assets at the required granularity.

Additionally we investigate the use of duplicate code as reusable assets. While this approach has not been pursued in previous research, we found that code clones represent structural similarities in the systems and with sufficient re-engineering can be reused.
Chapter 2: Research Contributions

The Automated Reuse Method (CDCM) method makes the following contributions:

1. Detects common domain concepts across different applications developed by the organization.
2. Evaluates and investigates the use of duplicate code as a basis for enabling component reuse.
3. Provide methods for transforming the detected code clones into reusable assets.
4. Support the inception of a searchable library of reusable source code by providing relevant metadata. This could bootstrap component reuse efforts at the organization.
5. The approach would provide a high level overview of the current health of applications developed at the organization and identify areas for improvement.

The application of the CDCM approach on 3 projects (~5 million LOC) of the same type (3 tier web applications) yielded ~1 million structural similarities. We identified a potential of using roughly 10-15% of the code within an application and around 1-2% of the code between different applications.

We envision a library contains about 10% of the organization’s existing assets in reusable form. The inception of such a repository could boost code reuse in current development efforts by roughly 10%.
Chapter 3: Problem Statement

The core problem at hand is the detection, classification & the transformation of duplicate code present in existing software into potentially reusable assets. In this sense the problem is similar to classic information retrieval where the goal is to select a small set of documents in a large collection based on relevance to a particular query.

Solution Approach

In practice, applying existing software has not been a focus of systematic reuse of source code components. This is not surprising given the generally uneven quality of existing software and its overwhelming volume. [5] estimate that 50% of existing information systems software is of low quality.

Existing software might contribute to a software reuse effort by enabling a repository with potentially reusable assets. This repository could ideally bootstrap systematic reuse initiatives within the organization. At the very least it would provide an ideal solution from which the organization could learn important lessons about reuse and improve its efforts in the future.

Additionally organizations develop web applications to provide services to customers. Such services are constrained by business rules at the organization. It must be noted that both the rules and services are implemented in code; this indicates that the source code
that supports a service has a large amount of domain information about the organization and its business. It is also worth mentioning that both 1) business rules and 2) Services are fairly constant and are not subject to constant changes. This proposes that the source code that supports such operations mustn’t change as well.
Chapter 4: Background and Related Work

This section motivates our work, provides background on code reuse and discusses related work. The theory concerning software reuse, past trends in reuse, clone detectors and software artifact repository design will be provided to understand the requirements of an organization’s code reuse effort. This section can be divided into five sections.

1. Software Reuse
2. Industry Reuse Efforts
3. Code Clones
4. Reusable Asset Libraries
5. Domain Analysis

Software Reuse

Literature abounds with material emphasizing the benefits of software reuse, particularly with regard to component-based software engineering. [2] Approach the reuse problem from the perspective of complex services and service-oriented architectures. They propose that existing systems offer services and that the services can be reused while building newer systems. Essentially they concentrate on “Reuse of services” and hence enable reuse of code.
Reuse of even internally developed and consumed assets does not come free. Earlier efforts to extract parts from existing software led to a conclusion that parts must be designed for reuse [22]. [17] identify three costs associated with reuse:

- The cost of making something reusable (i.e., additional generalization, documentation, testing, and library support),
- The cost of reusing it,
- The cost of defining and implementing a reuse process.

These costs are incurred as part of what Schach calls “deliberate reuse”, or the utilization of software components constructed specifically for the purpose of future reuse [30]. [32] Make the claim that there are significant risks with achieving successes with reuse because of the poorly documented and maintained state of existing applications and the fact that many systems were initially developed for different paradigms than currently distributed systems.

There are three “cons” of software reuse that act as criteria for evaluating the reusability of a software component is:

- Concept (the abstraction the component represents),
- Content (the implementation of the component),
- Context (what is needed to complete the definition of a component under a certain environment) [19].

Another group of studies conducted in the 1985-1990 timeframe, explored ways to apply software metrics to software reuse. This work provides some general guidelines that may indicate attributes of existing software that are desirable for reusable software.
[4] Provide an insight into software ecosystems and their benefits. Ideally a software ecosystem would need to be focused on reuse of existing code, software models and knowledge.

Object-Oriented Design Patterns are inherently reusable because they present a solution to a problem that occurs often [1]; a plethora of techniques were proposed to detect design patterns in existing code.

**Industry Reuse Efforts**

[16] Mention that organizations are not aware of how reuse needs to be addressed. They go on to say that software reuse has not reached its full potential and propose a conceptual management tool for supporting reuse at an organizational level.

Research efforts have been made by industry to address the unique needs of this enterprise reuse problem. A large organization will need to introduce systematic reuse in phases. It is important to pick application domains that have the best probability of success for these initial efforts [33].

Rational Corporation has undertaken a reuse initiative known as Rational Development Accelerators that incorporates reuse standards, 13 reusable software frameworks (i.e., templates), and automation for reusable asset cataloguing and retrieval.

At the core of this effort is the Rational Corporation’s Reusable Asset Specification (RAS), “a collection of guidelines for the description, development, and application of different kinds of reusable software assets” [13].
Efforts such as the RAS demonstrate the complexity of software reuse at the enterprise level. The necessity of providing such a plethora of information in order to make a software asset reusable incurs a cost, which therefore incurs risk. An investment of developer time and payroll must be made in the present in order to detail software assets in such a way that they will be comprehensible and accessible by other developers at some possibly undefined point in the future.

The economics of reuse are such that the more an asset is successfully reused, the higher the return on this investment. [34] accurately say” Substantial experience and investment is a prerequisite for architected reuse”. The risk of a reuse strategy that incurs high overhead [20] in order to succeed is under the uncertainty of the reuse ever happening. If the asset is never actually needed, then the time and money spent describing and specifying the asset will be a loss.

Of course, this is not to say that reuse at the enterprise level is hopeless or economically infeasible. The risk involved necessitates a certain degree of planning for reusable software up-front, and calls for an estimation of how useful the software will be in the future. To achieve a balance between investment and reusable assets it is important to select 5-10% of existing software for analysis for reuse [35].

Many researchers argue that tools alone are insufficient to contribute to a significant software reuse practice and may not be necessary considering the crucial role played by other factors such as management involvement, developer education and training, and a software process that supports and encourages reuse. [36] in his PhD thesis mentions that reuse is hindered by IT governance as well. The work done revolves around how reuse is
affected by the relationships between IT strategy, IT governance, decision makers, business goals and software reuse successes. He goes on to propose a software reuse success strategy model which analyzes the factors involved in the success of software reuse.

Surveys conducted by [23] revealed a statistically insignificant increase in reuse due to the presence of a repository: 10% higher than an organization without a repository. It seems logical that the mere presence of a repository in a corporate setting would be a neither necessary nor sufficient condition for reuse, and this has been confirmed many times over in the literature.

Reuse efforts conducted by organizations like Toshiba, Hitachi, NEC, NTT (Nippon Telegraph and Telephone) and Philips. [3] [11] document the successes achieved by these organizations for their mainframe applications and medical applications. The successes are categorized into 3 focus areas:

- Domain Analysis
- Reuse of Existing Code
- Managerial Aspects of Organizational Reuse.

[3] provide case studies how organizations kick-started maintained and had success with their reuse efforts. It must be noted that all of them adopted CASE based development and recorded reuse ratios of 20% and above. The paper also explains how the organizations achieved 100% usage of their reuse repository in less than 5 years. This is paper is the primary motivator of our approach.
Code Clones

Code clones are traditionally seen as a maintenance nightmare. [24] “Cloning can be a good thing if you have the right tools in place. Let the programmers copy and adjust and then let the tools factor out the differences with appropriate mechanisms.” This also suggests that the detection and removal of code clones promises decreased software maintenance costs. They also identify the reasons for the occurrence of code clones:

- Code reuse by copying pre-existing idioms
- Coding styles
- Instantiations of definitional computations
- Failure to identify/use abstract data types
- Performance enhancement
- Accidents

[25] propose a method of partially redesigning existing Java software systems based on clone analysis to improve quality and improve maintenance costs, they also investigate the use of clones as a basis for reengineering actions that are useful to the maintenance of systems.

Clone Detectors

These tools are based on the idea that blocks of code show similarity to other code blocks. Ideally clone detectors define metrics to measure similarities; code blocks with the same similarity metrics indicate presence of clones.
CCFinderX

Originally proposed and developed by Toshihiro Kamiya et.al [37] from Future University Hakodate, Japan. CCFinderX has been a topic of constant discussion and research for the last 12 years. The team behind the idea has been awarded several prestigious awards for their work on the tool.

CCFinderX uses a token string matching approach to detect clones. It breaks the source code into tokens and looks for matches over the sequences of tokens. The token based matching approach has the advantage that it results in an approach that is independent of the actual layout of the source code; this allows the tool to detect structurally similar (reformatted) code in addition to duplicate code. The approach, however results in false positives i.e. code blocks being detected as clones even when there is no structural similarity between them.

Several commendable approaches have been used to detect clones like Pattern Matching [8], Abstract Syntax Trees [24], Token String Matching [37], Data flows, Function Based and dot plot style. Tools such as Clone Miner, Clone Doctor, Clone Digger, CCFinderX, Clone Tracker, Reclipse, etc. have been developed based on these approaches.

Reusable Asset Libraries

Version control systems appear at first glance to be a possible means of distribution for software assets in an organization. Version control systems store software projects throughout development, and easily allow previous versions of assets to be viewed. Since
they already provide code and document storage, it may seem redundant to store assets in a separate reuse repository. This is not the case.

Version control systems and reuse repositories serve different purposes in the software development cycle. Version control is meant to be used for work-in-progress, and therefore the features found in these tools are designed to serve that purpose. Specifically, they lack the ability to attach meaningful metadata to assets, and they are not readily searchable [26]. “Version control tools alone provide an ineffective means of tracking and distributing finished goods software inventory”. Software reuse patterns created to describe successful reuse practices recommend a “Divide and conquer” approach in which the functions of building, managing, and reusing software assets should be separated [27]. In light of these observations, version control systems are not a suitable tool for reusable assets.

Ideally we need a repository similar to Software Information Base [38] to successfully implement reuse of components and specifications.

Metadata

“The association of standardized descriptive metadata with networked objects has the potential for substantially improving resource discovery capabilities by enabling field-based (e.g., author, title) searches, permitting indexing of contextual objects, and allowing access to the surrogate content that is distinct from access to the content of the resource itself” [31].

This aptly describes the potential for metadata to increase the relevance of data returned to queries against a software asset repository. Metadata offers an opportunity for an
alternate form of repository and provides additional dimensions against which queries for
the data can be performed. It gives a semantic meaning to possibly isolated data and gives
context to the search process.

We established earlier that “Context” is the most important aspect of a reusable asset. We
must therefore record the metadata in a way that maximizes possibility of returning good
search results.

Some standards like Rational Reusable Asset Specification (RAS) [13], Sharable Content
Object Reference Model (SCORM) [29] and Dublin Core Metadata Initiative have tried
to create standard metadata elements to describe Reusable Assets. Standardization of
metadata is important when interoperability of desired components is necessary.

**Asset Availability**

In order for a software reuse support tool to be as useful as possible, the assets within
should be available to the developer anytime, anywhere, on demand. It seems logical that
the availability offered by the Internet and the World Wide Web should be extended to
reuse solutions. A web-based or web-enabled reuse support tool would allow developers
to search, access, and update reusable assets at any time.

**Domain Analysis**

A key idea in software reuse is domain engineering. The key insight is that most software
systems are not new, they are variants of systems that have already been built. Most
organizations build software systems within a few business lines called domains. This
key insight can be leveraged to improve the quality and productivity of software production process [20].

[15] propose that the reuse of software artifacts is obtained by organizing a reuse framework around a specific domain. They go on to propose a system for reuse where they organize similar software cases into domains and try to establish a systematic cross-domain reuse process.

Topic modeling has been used for the analysis of a corpus by generating a semantic overview. Such overviews have proven useful for indexing, classifying, and characterizing documents. These techniques combine topic modeling with clustering of the topics to identify the dominating commonalities (main domains) between software systems. [28]

Domain analysis is the “process by which information used in existing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems”. This is the first step in which we try to build and document a generic architecture for a particular domain based on an extensive set of data available for the domain. Thereafter application re-engineering processes can reuse this architecture to provide custom products.

[2] approach the problem from the perspective of enriching existing systems with business semantics and try to manage to bridge the gap between services and source code using domain analysis, thereby allowing reuse of services.

Topic modeling is an automatic technique that can be used to bootstrap commonality analysis. Techniques such as LDA (Latent Dirichlet Analysis) analyze the corpus and
produce a probabilistic model of topics that groups semantically related topics. Since the clustering is based on the co-occurrence of words in each document it gives us a high level picture of the domains in the corpus. Ideally topic modeling can be used to enable domain analysis to study the differences and commonalities between groups of applications at various levels of granularity.

Mallet

Mallet is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text. It has been developed by Andrew Kachites et.al from the University of Amherst and is released under the Common Public License [10]. Mallet provides topic models that are used to analyze large collections of unlabelled text. The toolkit consists of the Pachinko Allocation algorithm and Hierarchical LDA in addition to Latent Dirichlet Allocation to analyze the text. Mallet has been used successfully to analyze and categorize large amounts of source code [10]. Additionally Mallet has been known to create accurate topic models from textual data. Tools similar to Mallet are being used widely in industry to analyze large amounts of data.

Summary of Related Work

In sum a reuse asset effort must yield a tool that is accessible, automatic, customizable, searchable, navigable, flexible, secure and relational. Additionally the reuse effort based on existing software must focus on both semantic and structural similarities.
In light of the information provided by the theoretical framework we set out to provide an organization with a reuse effort that allows it to build a repository of transformed clones as reusable assets practically. We propose metadata that must be included to incorporate the ideal features of the repository. Additionally we propose methods to transform clones into reusable assets; this operation could improve the quality of existing software and reduce maintenance costs.
Chapter 5: Data Selection

We performed an exploratory empirical study to qualitatively evaluate our method on 3 software projects developed by the application development center of a large complex services enterprise. The projects vary in functionality and affect millions of customers and businesses every year. All of these projects are Java & J2EE web applications that range in size from 300,000 to 2,000,000 lines of Java source code.

Roughly 5 million lines of code were analyzed using clone detectors and topic modelers in the course of this study and the results were quite interesting.
Chapter 6: Approach

Figure 1: Reuse oriented software development model

Figure 1 illustrates a reuse oriented software development model [3]. The model proposes that reusable assets be created in addition to the original product. The model creates a “necessary” feedback loop to use (reuse) information from previously developed code. The model is a representation of how reuse must “ideally” take place in an organization. It proposes that reusable assets be created in parallel with current products; additionally it propagates the use of existing assets to develop new code. While this model is accurate it falls short on a few issues:
1. The definition of the reusable library and an authority that would maintain it is missing.
2. Representation of the components in the repository is not clear.
3. No clear indication of “how” existing components will be a part of the repository.

We propose the Clone Detection and Cataloguing Method (CDCM) to remedy the downfalls of the reuse oriented development model. CDCM would automate the process of finding potentially reusable code in existing software and would catalogue them semantically in a way that can be searched for future development.

**Clone Detection & Cataloging Method**

The **Clone Detection & Cataloging Method (CDCM)** proposes an effective strategy to recover and catalogue potentially reusable components from a large portfolio of existing assets. Additionally CDCM proposes methods for reengineering the detected “raw reusable assets” into a form that can be easily used by others.

Conceptually CDCM looks for projects that have similar domains and then finds structurally similar code in the domains. By looking for similar code in a domain, CDCM improves the probability of finding combining semantic and structural aspects inherently present in the code.
CDCM necessitates that reusable asset production from existing software needs to be done in 5 steps:

1. Combination of Source Code into a single codebase
2. Domain Analysis
3. Clone Detection
   a. Clone Cataloging
4. Clone Transformation
5. Reusable Asset Packaging

**Step 1: Combination of Source Code**

We combine all of code developed by the organization into a single codebase and visualize it as a single asset. We treat all the code assets as single entity. Previous research suggests that that development teams rarely collaborate with other teams when developing a product. This step is aimed at breaking the logical barriers that are introduced by the isolation of development teams.
Step 2: Domain Analysis

Generally projects within similar domains have higher probability for reuse. Our approach first pre-processes the source code of all the software systems in consideration to extract meaningful words. Each project’s extracted words are then analyzed by a separate LDA process,

Topic modeling gives clusters of dominating topics (domains) in the source code. We also obtain the memberships of each source code file to the identified clusters. If the source code files of 2 or more different applications are found to have similar membership in a topic cluster; we conclude that the applications belong to similar domains. This step focuses on finding semantic similarities between the applications.

Rationale

Generally, names for identifiers are chosen such that they are meaningful to the overall goal of the program. For example: if a program deals with customers and their accounts; the names chosen would probably be customerName, accountDetails, etc.

Writing code using the standard naming conventions improves the readability of the code. This naming convention also encapsulates information about what the code is meant for, in other words naming conventions carry enough information for us to guess the specifications for which the code was written.

Methodology

Prior to performing topic analysis on a project, each source code file must first be preprocessed. Traditionally, unwanted words and punctuation are removed to eliminate
frequently occurring words that contain no semantic meaning with respect to the code’s functionality.

**Source Code Pre-processing**

This removal comprises the following steps:

1. **Removal of language-specific words.** This step is the only programming language-specific step of our approach. We process the source code text to remove keywords specific to Java. Examples of the Java reserved words that were removed are `class`, `get`, `put`, `public`, `private` etc.

2. **Breakdown/splitting of identifier names.** We separate the (camel case) names for the identifiers used in the projects.

   For example, `myVariableName` would be split into `my`, `variable`, and `name`.

---

**Figure 3: Source Code Pre-Processed for LDA**
Figure 3 illustrates a source code file after pre-processing. It can be seen that the figure contains words that are relevant to the ultimate goal of the code.

**LDA Analysis**

In this step we apply LDA topic modeling technique to the pre-processed code. For domain analysis we apply LDA to the complete codebase. This step allows us to create clusters of common topics between different projects and hence allows us to locate semantically similar projects.

For LDA Analysis, we use the open source Mallet tool [10]. Mallet was configured to learn 150 topics from the entire codebase for domain analysis.

LDA topic analysis is also applied to each project’s pre-processed code independently. This step allows us to create training models to catalogue clones in the step 3a. Mallet was configured to create 60 topics.

Changing the number of topics would change the granularity at which analysis is done. 150 overall topics and 60 for each project were found to provide a fairly fine-grained overview of the code.

**Discussion**

Topic modeling was used:

- For domain analysis of the code: to identify commonalities and variability between different applications.
• To create several levels of code specific domains.
• To identify semantic similarities between projects.
• To facilitate code comprehension: to reengineer the detected commonalities
• To train a machine model from the existing corpus of source code. These trained models can be used to catalogue the detected clones (Step 3a).

**Step 3: Clone Detection**

Same or similar pieces of code in software are clones. A clone generally consists of 2 or more constituent code blocks. Presence of clones in different systems hints at the presence of design level similarities or of components that are perhaps used often. This step focuses on extracting structural similarities between the applications.

**Rationale**

Clone detection reveals the presence of copy-pasted (already reused) code in a system. We propose that clones should be included in the reuse library because they inherently indicate reuse.

The existence of code clones in semantically related projects (similar membership to a topic cluster) indicates the presence of code with a high reuse potential, because there is both “a strong semantic” and “a structural” relationship between the clones.

**Methodology**

To detect clones within and between different projects we used a clone detector called CCFinderX [37]. It uses a token string based matching algorithm to detect clones in the
code. It breaks the source code into language tokens for the language of interest, and look for matches over sequences of tokens.

The tool is free, extremely fast and is stable for large inputs. Additionally it provides various visualizations and metrics for the detected clones.

**Discussion**

The clones detected by the tool can be broadly classified into 4 types:

- **Type 1**: Copies of methods scattered in each individual project (in different files/classes)

- **Type 2**: Copies of code scattered throughout each file/class

- **Type 3**: Copies of classes with additional member methods.

- **Type 4**: Structurally similar code working on different data structures (usually found between projects)

Type 1 and 2 clones represent parts of the program that rely on or must affect many other parts of the system. Existence of such clones represents functionality that is needed by multiple methods within a class or different objects; essentially the code is a “cross-cutting concern” in the system. The duplication associated with such functionality results in loss of modularity.

Type 3 and 4 clones represent parts of the system that are similar in structure but have slightly different behavior. Although the ultimate goal of the code in these clones is different, the structural similarities present opportunities for reuse in future efforts.
Reusable assets produced from such clones will exhibit greatly varied behavior and could be easily extended.

**Step 3 a: Clone Cataloging**

A clone detected in step 3 of CDCM is cataloged using the trained model. Here we ask a trained model (created in step 2) to classify the code of the clone into previously identified topic clusters. Mallet provides a probabilistic model of the memberships of the code clone to different topic clusters. Since the model provides probabilistic memberships, it can be used as metadata to annotate the clone in the reuse library and lay the foundation for a library search tool.

Additionally cataloging is also necessary to understand its functionality and facilitate the clone’s refactoring into a reusable asset.

**Step 4: Clone transformation**

Though the detected clones represent frequently used code. They are usually not in a form that can be easily used (reused). Additional work must be done on the clone to transform it into a reusable asset.

The transformation process must

- Factorize commonalties found in clones.
- Parameterize the differences to preserve original behavior.

The asset created must possess the following characteristics:

- It must be decoupled from its environment to allow for expansion.
• It must be partially configurable to allow for its use.

• It must be reliable and testable.

Research suggests that clones can be transformed into reusable assets using approaches involving Aspect Oriented Programming (AOP) and Object Oriented Design Patterns (OODP). Details about clone transformation are discussed in the next section with examples.

**Step 5: Reusable Asset Packaging**

Each reusable asset that created from the transformation process must be packaged such that it is easily usable and searchable. The asset must consist of documentation about how its constituents were used in the original contexts. The asset must also carry the probabilistic values for the topics that it belongs to. Other information such as the team members, who checked in the code/transformed it and test scripts may be useful as well. Ideally we try to annotate the asset with as much knowledge as we can to ease its use. This would allow us to build a second tier of searchable information about the asset and hence improve the accuracy of the searches performed for the asset.
Chapter 7: Clone Transformation

We propose the use of Aspect-Oriented Programming (AOP) and Object-Oriented Design Patterns (OOPD) to reengineer the detected clones. Once a code clone has been identified as potentially reusable we reengineer it to make it reusable.

Clone Transformation for Type 1 and Type 2 Clones

For example: the code for logging the system is required by different components in the system. Therefore logging is said to “crosscut” all logged classes and methods in the system.

– Aspect Oriented Programming entails breaking down the program logic into distinct parts (concerns) providing abstractions that can be used for implementing the concerns. Essentially each concern resides in a single place from where it is invoked by different objects that need the functionality.

– We propose that the clones (Types 1 & 2) be transformed into aspects to make them reusable.

– Each class that needs the functionality (concern) will need to register itself in an aspect configuration file that will accompany each aspect.

Clone Transformation for Type 3 and Type 4 Clones:
Such clones need to be redesigned in 2 steps:

- **Step 1: Identifying Common behavior in the constituents**
  
  Common behavior must first be captured in the form of an interface or an abstract class. This allows us to implement common behavior in any classes that inherit from it.

- **Step 2: Identify the differences in the clone constituents**
  
  We need to use the Strategy Object Oriented Design Pattern which defines a family of algorithms, encapsulates each one and makes them interchangeable. The pattern lets the algorithm vary independently from the clients that use it.

![Strategy Design Pattern](source.wikipedia.org)

Each different solution offered by the clone constituents would have to be implemented as a strategy (a concrete class that implements a specific algorithm) by inheriting common behavior defined as an interface in step 1.

Additionally the asset would need a clone handler class that would contain logic for choosing a specific implementation that each constituent offers. To reuse the asset we would need to create an instance of the specific implementation; if a solution does not
exist but operates on the similar data the asset can be extended by writing a new strategy in the asset. This way, existing assets can be extended.

Example

Let us say that a clone has two constituents:

```java
void setStartAccount (int month, int balance, int year, String name){
    startMonth = month;
    startBalance = balance;
    startYear = year;
    accountName = name;
}
void setEndAccount (int month, int balance, int year, String name){
    endMonth = month;
    endBalance = balance;
    endYear = year;
    accountName = name;
}
```

**Common** behavior is the “setting of the name” variable.

This can be captured in an interface called cloneSimilarities:

```java
public interface cloneSimilarities {
    public void setName (String name);
}
```

**Differences:** variables that are being set (startMonth, endMonth, startYear, endYear, startBalance & endBalance). Such differences can be captured in another interface:

```java
public interface cloneDiffs {
    public void setMonth (int month);
    public void setYear (int year);
    public void setBalance (int balance);
}
```

We need to change the original methods and parameterize the differences. We do this by creating a different class for start and end accounts.

To implement each strategy we implement the cloneDiffs interface into each class.

```java
class startAccount implements CloneDiffs{

```
public void setMonth(int month){
    startMonth = month;
}

class endAccount implements CloneDiffs {
    public void setMonth(int month){
        endMonth = month;
    }

};

The newly created method in the clone handler will contain the strategies needed as parameters.

public class cloneHandler {
    void setStartAccount(int month, int balance, int year, String name, cloneDiffs diff, cloneSimilarities sim) {
        diff.setMonth(month);
        diff.setYear(year);
        diff.setBalance(balance);
        sim.setName(name);
    }

The handler is implemented as a new class and all method calls in the clone are made through it. The following code demonstrates how the calls must be made.

void setStartAccount(int month, int balance, int year, String name) {
    Handler.setStartAccount(month, balance, year, name, start_strategy_ID);
}

void setEndAccount(int month, int balance, int year, String name) {
    Handler.setStartAccount(month, balance, year, name, end_strategy_ID);
}

In the above step we call the same method (setStartAccount) in the handler and pass different strategy IDs as parameters to implement specific behavior.
We can see that though the clone transformation process is complicated it promises the creation of assets that can be easily reused. We propose that the transformed asset (its handler and associated interfaces) be included in the reuse library.
Chapter 8: Results

Domain Analysis - Topic Modeling

Mallet [10] outputs the topic proportions of each file in the form shown below:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Topic</th>
<th>Membership</th>
<th>Topic</th>
<th>Membership</th>
<th>Topic</th>
<th>Membership</th>
<th>Topic</th>
<th>Membership</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentDisplayDTO.java</td>
<td>41</td>
<td>0.787199</td>
<td>13</td>
<td>0.030864</td>
<td>48</td>
<td>0.023067</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>AgentDTO.java</td>
<td>48</td>
<td>0.290262</td>
<td>47</td>
<td>0.088015</td>
<td>41</td>
<td>0.043071</td>
<td>13</td>
<td>0.043071</td>
</tr>
</tbody>
</table>

Table 1: File memberships to topics

The table shows that the file “AgentDisplayDTO.java” has a membership of 78.71 % to topic 41 and a membership of 2.30% to topic 48.

Topic 41 contains the following words as major contributors.

**Topic 41:** agent, number, state, cross, sell, producer, search, agency, office

**Topic 48:** dto, display, agency, office, day, time, formatted, hrs, hours

The topic memberships of each project can be demonstrated in the form of a bar chart as shown in figure 5. This allows us to identify domains that are common to different
projects i.e. if 2 or more topics have similar memberships to a topic cluster, they represent similar domains.
Figure 5: Topic Modeling - 100 Topics – Project Memberships
Figure 5 illustrates the memberships of the each project to the learned topic clusters.

The x-axis indicates the topics (topic ids) that were learned using Mallet in step 2 of CDCM.

The y-axis shows membership of each project to the topic cluster.

For example: Topic 1 has a 58% membership for the project in blue, roughly 34 % for the project in red and about 8% for the project in blue.

Because the projects (red and blue) have similar memberships to topic 1, we can say they are related semantically and that there is higher probability of finding good quality code clones in the files that have high memberships to this cluster.

**Observations**

One of the projects in consideration was considerably larger than the others and seemed to overpower the other projects in terms of topic memberships. This result indicates that topic modeling using LDA is biased towards corpora with a larger size. However it is interesting to note the similar memberships of the projects in some domains (topics).

**Validation**

The topic memberships of 250 files were validated manually; additionally we conducted interviews with original developers to confirm the observations.

**Analysis**
Since topic modeling is based on the names of variables, methods and classes. It carries semantic information about the project. Similar memberships to topic clusters would indicate use of similar names for variables and methods in the two projects. Therefore it is highly likely that the operations performed in the constituent files are similar. We detect these similarities using topic modeling.

**Clone Detection**

The tool uses a token string matching approach to detect clones in the input. Additionally the tool provides useful metrics about the detected clones.

Clone detectors were configured to detect clones that exceeded 50 lines of code. This was done to ignore simple similarities between the projects.

- CCFinderX detected 1,132,463 pairs of clones (>50 LOC) in the source code of the 3 applications that was input to it.
- Assuming a false positive rate of 10% we get at least a million clone code pairs in the source code.
- This represents duplication in the code and shows code that was reused but not documented.

**Outputs**

CCFinderX offers a scatter plot view to display how code clones are distributed across the entire input of source code. It also offers a scatter plot of the source code of the clones and shows how these clones occur between corresponding source code files.
The scatter plot indicates the presence of clones in the folder structure for each project.

Each axis represents the folder structure as a linear system. It is known that developers try to place code in folders that are relevant.

Figure 6: Code clones in all projects – shows similarity between code sections over the file system

Figure 7 illustrates the scatter plot showing clones detected between projects A, B & C. The red, orange and blue squares represent the different projects under consideration. Gray squares in the brightly colored squares represent directories within each project.
The black dots in figure 7 that are located parallel to the main diagonal line are the clones that are detected in the input. The corresponding orthogonal projections to the horizontal and vertical axes are the locations of the similar code fragments.

In figure 7, the green box indicates the clones that are found between the three projects.

**Axes Interpretation**

The axes are a representation of the folder structure of the projects. For example if we have a folder structure as represented in figure 8

![Folder Structure](image)

Figure 7: Sample Folder Structure

The axis in the scatter plot would represent the root folder (memories) along its length. The subfolders (Cartoons and Nature) are represented as line segments that constitute the axis; the length of the line segments is proportional to the size of the subfolders. The folders within the Cartoons directory (Amul and Cricket) are further represented as line segments that constitute the line segment that represents the Cartoons line segment. This can be visualized using figure 9:
The average length of clones in

- Project C was found to be 238.055 LOC
- Project B was found to be 323.937 LOC
- Project A was found to be 276.526 LOC

CCFinderX also identifies the exact location of the clone within different files.
Figure 10 illustrates the co-occurrence of similar code in different source code files. Each vertical line represents a file; circular dots represent “similar code” and the lines/curves between the two vertical lines shows how the code is duplicated. The length of the vertical line is directly proportional to the lines of code in that file. In the figure above we observe how a smaller file is created by almost entirely copying a large file.
Observations

- The code detected as similar by the clone detector was found to differ in 3 key aspects:
  - Data Structures operated upon.
  - Sequence/ Number of method calls
  - Variable Names
- We observed a high concentration of clones within each project/application.
- In some directories 45-50% of the code was found to be either duplicated or similar.
- In Figure 4 we observe that the green square is not as densely populated with black dots as the other squares. This square represents similar code found between different applications.
- The constituents of the clones found between different applications were not as similar to each other as the ones found within the applications. Typically they belonged to the Type 4 category identified earlier.

Validation

We manually validated the constituents of 200 clones of Types 1, 2 and 3. The clones were found to be duplicates and were generally located in a single project. Additionally a few “mock” clones were created in each project to observe if they were detected. We also validated the output by planting completely dissimilar pieces of code in the directories and verified that they were not detected.
Analysis

- We found that the clones found within each project (in the same and different directories) consisted of exact duplicates. Interestingly the code was used in the same context and accomplished the same tasks. Ideally they should have been a part of a procedure/method and invoked from a single location.

- The clones within projects were found to be fairly large; in some cases clones larger than 1000 LOC were found.

- The clones whose constituents were from different applications were found to be structurally similar and not exact duplicates. In this sense the code looked similar but was found to operate on completely different data structures. Their context of use was different as well. At a high level we observed similarities in the ultimate goal of the code in the clones.
Chapter 9: Conclusion

The research provides key insights into the development practices at the team level in a large enterprise. The research augments studies conducted in the domain of reuse by providing a new dimension of code analysis i.e. reuse efforts need to be focused at a team level. We envision that a reuse framework that focuses on intra-team reuse would lay a solid foundation for inter-team reuse. In addition to validating the presence of reusable code in existing software, the research also proposes to lower maintenance costs associated with new applications.

The clone constituents in a project were found to have a strong semantic and structural relationship. This led to the finding of a high concentration of reusable components in each project. The real challenge however, was to detect similar code in between different applications. We noticed that though the clone detectors succeeded in finding code that was similar in between different applications, there seemed to be no semantic relationship between them. In fact the results for clone detection without domain analysis were similar to the results that were obtained after domain analysis.

Though the results suggest that CDCM succeeded to find reusable components at a project level but failed to find reusable components at an organizational level, it confirms the need for additional work in this area, possibly at a level between the very high level of domain analysis and the very low level code duplication.
Another important conclusion from CDCM is that continuous creation of reusable assets is necessary as the development on a project progresses. Teams must revisit already developed code, look for duplications and generate reusable assets. The assets produced must be used to re-factor the application being developed. This approach has the benefit of improving the maintainability of the application in addition to generating reusable assets.
Chapter 10: Future Work

CDCM has presented a new view that allows organizations to analyze their assets at a level that is necessary to build a searchable reuse library. The shortcomings of CDCM identified in the previous section indicate the need for more work to be done. There is potential for more work to extend the CDCM to make it an efficient method for reuse library population.

Our work focused on extracting reusable assets from the source code of three applications. CDCM needs to be applied to more projects and a detailed analysis of the results must be conducted to identify any emerging patterns.

We have identified two aspects of software development that could be used to extend CDCM and introduce additional levels of analysis between semantic and structural similarities:

1. Documentation & Service descriptions
2. Test Scripts.

Current trends in software development indicate that organizations are focusing on Acceptance Test Driven Development (ATDD) as a unit testing framework. The test scripts contain a large amount of semantic information about each project and this information could be potentially useful from the perspective of code reuse.
A framework is needed along with CDCM to unite documentation, test cases and source code together. Substantial progress can be made towards the goal of code reuse if the framework succeeds.
References


