Cristina: A tool for refactoring source codes of game prototypes into reusable codebases

Wolmir Nemitz[1]*

[1]Universidade Federal do Pampa, Software Engineering Department, Brazil

ABSTRACT

Game engines are expensive. One of its biggest hidden costs is in adapting its features to suit a particular game. Therefore, ease of modification is a key requirement in game engine selection. Even so, traditional game development studios prefer to develop most of their technology. In both cases, code reuse is an important factor. Prototyping, as a software process model, is used extensively in the game development industry. Game jams are rapid game development events. They are equivalent to the game prototyping process and, generally, have a source code submission archive on the Web. This means a good supply of non-reusable code. As per the objective of this work, we built an application that transforms the source codes of a set of game prototypes into a reusable codebase. We researched automated refactoring techniques to use in the application. We also looked for reusability metrics to validate the tool's effectiveness in producing reusable classes from the original code. The application was successful in transforming the source codes of a set of prototypes submitted to the game jam PyWeek. An experiment confirmed that the extracted codebase was more reusable than the original code. It also validated the proposed refactoring method. This work is important because it shows the recycling of development by-products to reduce costs.

Keywords: Game Engine. Game Jam. Iterative Process. Game Prototyping. Game Development.

1 INTRODUCTION

Game engine is an open and extendable software system on which a game can be built [1]. Game studios that acquire these systems do so at a great cost [8]. In his research, DeLoura found that a one year license of a high level engine may be priced above one million dollars. Of his interviewees, 86% agreed that the biggest hidden cost is in adapting the engine's code to suit specific games. That is why they consider ease of modification and availability of source code as the main criteria when choosing a game engine. Still, DeLoura's results shows that most traditional game developers use their own game technology.

Code reusability is an important factor in both cases. In the first scenario, developers need to adapt the engine source to their own technology. To avoid costs they need to modify as little code as possible. In the second scenario, they need to reuse past projects to avoid effort redundancy.

Prototyping is a software process model [19]. Pressman states that developers use it when there are uncertainties about the objectives and requirements of a system. It is an iterative model with the following steps: defining the overall objectives of the client; rapidly build a prototype; test it with the client in order to refine the objectives for the next iteration. However, Pressman points out two problems with this methodology. The first is when the client ignores the bad quality of the prototype in favor of development speed. The second is when developers decide tho keep the prototypes as legacy code, compromising the quality of the end product. This process should be used when clients and developers agree about the illustrative and disposable nature of the prototypes.

*e-mail: wolmir.nemitz@gmail.com

Game development is an area that conforms to the Pressman's criteria for Prototyping [14]. Manker says that game prototypes are communication tools that bid attention to a subset of a design space. This space is represented by all the possible decisions a designer can make along the course of a project. Manker, Callele, Neufeld and Schneider [6], and Fullerton [11] show that the requirements engineering in game development is difficult and sensitive to changes. Generally, the addition of rules to any complex system can produce effects difficult to anticipate [14] [11]. Game developers must reduce the risks of adding complexity to a game before the project goes into production, when the cost of making changes is higher [14]. Thus, short iterations are important, which explains the adoption of rapidly buildable prototypes in game development to test scope decisions.

A pragmatical application of this technique is often seen in *game jams* [16]. Musil et. al define game jams as events in which small teams must develop a game within a set of restrictions. The authors approach game jams as a solution to the problem of identifying requirements in the early phases of a game project. They use the fact that game jams constraints are the same as game prototyping constraints to establish an equivalency between a game jam and the prototyping process. The submission of games to most of these events is done on a web platform. A side effect of this is the online availability of these source codes.

Our motivation is the gap between a demand for very reusable code, as shown by DeLoura's research, and the supply of very unusable code on the web. Our objective is a software application to bridge this gap. We conjecture that the results of the application on a set of source files from game jam submissions is a more reusable code base. To support this claim, we use object oriented metrics to compare the output code with the original source code.

The rest of this paper is organised as follows. Section 2 presents the related literature. Section 3 presents the theory behind the algorithm we propose. We outline the application architecture in section 4 and the empirical experiment methodology in section 5. We finally present the results of our empirical validation in section 6.

2 RELATED WORK

We categorize the related literature in two areas: refactoring automation and reusability metrics. The first concerns the automated identification of refactoring opportunities as well as the automation of the refactoring techniques. The second is about quantitative measures for code reusability.

2.1 Refactoring Automation

Bavota, Lucia and Oliveto [4] propose a graph-based algorithm for automatically performing *Extract Class* [10] refactorings in *Java* source code. The method uses a combination of structural and semantic metrics as weights in a graph partition algorithm. The purpose is to extract two classes with a higher cohesion value than the original class. They confirmed the methods efficacy with an experiment. In this experiment they merged the classes of a system known by its design quality. The objective was to apply the algorithm on this modified class and show that the results were the two original classes. The authors concluded that a combination of semantic and structural metrics is more effective. However, their work assume a semantic relevance in the source code that is not present in game prototypes. Other problems with the method are: not considering class hierarchies; and not performing more than two extractions per class.

Fokaefs *et al.*[9] use a clustering algorithm to perform Extract Class refactorings. Their approach considers structural dependencies between the entities (methods and attributes) of a target class. Using this information, the algorithm builds an entity set for each attribute (containing the class methods which use that attribute) and for each method (containing the class properties this method uses, including the other methods it calls). Next, it computes the distance between pairs of entity sets in order to group entities by their cohesion values. These groups are the extracted classes. Their method, like the one we propose, use only structural metrics. However, it requires the user to manually define a threshold to discern between all the possible configurations of extracted classes. To find the right threshold without prior knowledge of the class being refactored is a difficult problem.

Simon *et al.* [20] implemented a visual metric-based tool to aid the software engineer in choosing refactoring candidates. Specifically, their tool allows the user to identify candidates for four types of refactoring: *Move Method, Move Field, Extract Class* and *Inline Class* [10]. Their algorithm also uses only structural metrics.

Bavota *et al.* [3] propose a new method for automatically performing *Extract Class* refactorings that addresses some problems of their old approach. The methods uses the same metrics to build a *method-by-method* matrix. The cells of the matrix are the probability that the line and column methods belong in the same class. This matrix represents a graph. The algorithm calculates a threshold and cuts the edges of this graph. The resulting subgraphs are called *method chains*. Trivial chains are chains with a number of nodes below a given boundary. Then, the algorithm merges trivial chains with the non-trivial chains with which they are most coupled. The resulting graphs are assembled into new classes. They validate the method with six open-source systems and a number of professionals of the area, who judged the usefulness of the refactorings. However, their work still ignores class hierarchies and dynamic programming languages, and assumes a semantically relevant code.

O'Keeffe and O'Cinneide [17] approach the automatisation of refactorings as a search problem. The authors test different optimization algorithms to improve the quality of two *Java* programs. They consider quality in three factors: understandability, flexibility and reusability. They measure quality with a linear combination of eleven metrics. Each factor represents a different combination of weight coefficients. They obtained positive results in improving understandability and flexibility. The negative results in reusability were attributed to the incompatibility of this factor with search methods. The authors conclude that their tool has potential for complex reengineering tasks. However, opposed to Bavota *et al.*, their algorithm only manipulates class hierarchies. The method also didn't focus on reusability, which might explain the negative results due to the trade-offs between this and the other factors. As well as the other works we considered concerning this area, O'Keffe and O'Cinneide worked with a strong-typed language.

2.2 Metrics

For Bansya and Davis [2] the literature, at the time, validated object-oriented design metrics with small and unrealistic data sets, which raises some doubts about the applicability of those metrics in an industrial setting. They also point to the inexistence of proven connections between the individual metrics and the quality attibutes they are supposed to measure. Furthermore, the authors state that the measurements apply only to implemented programs, which highlights the need for design-phase metrics. The authors propose a quality model composed of four levels and three mappings between these levels. The levels are: design quality attributes; object-oriented design properties; object-oriented design metrics; object-oriented design components. Bansya and Davis empirically identify the quality attributes: functionality, effectiveness, understandability, extendibility, reusability and flexibility. Design properties are tangible characteristics of design components. The authors identify eleven properties, such as cohesion and polymorphism. The authors derive eleven metrics from these properties and define components as classes, methods objects and relationships between classes. Finnally they map each quality attribute to a linear combination of the metrics. Bansya and Davis validate their model with several versions of two commercial systems. They test the model's ability to predict design quality and observed a positive correlation between the model's predictions and the evaluation of specialists. However, O'Keeffe and O'Cinneide [17] point out that the definitions of the metrics by Bansya and Davis lack formalism. They state that the definitions are done in natural language and are often ambiguous. Furthermore, the work of Bansya and Davis concerns the design phase of a project, while ours concerns implemented prototypes.

Goel and Bhatia [12] argument that reusability is a key attribute for reducing costs. They state that the Department of Defense of the United States would save U$$300 million annually if they improved their reuse by 1%. They also affirm that reusing software components improves productivity and reduce costs by up to 20%. However, quality improvement is only understood if measured objectively [12]. The authors use a combination of Chidamber and Kemerer [7] metrics to predict the reusability of software. They measure three systems with different inheritance strategies and compare the results with a reuse analysis. The results prove the efficacy of the metrics when it indicates that multilevel inheritance is the best reuse strategy. Goel and Bhatia do not provide a clear definition of Chidamber and Kemerer's metrics. For example, their interpretation of the *Lack of Cohesion Between Methods* is subjective. Our work, in the other hand, propose deterministic expressions for performing these measurements.

3 Theory

3.1 Game Prototypes

For Manker [14], the possible decisions of a game designer along the course of a project are represented in a design space. In this context, prototypes are communication tools with the purpose of addressing a region in this space. They also allow shorter iterations, because they can be quickly implemented. A low iteration period is important, because, according to Manker, Callele *et al.*[6] and Fullerton[11], requirements engineering in game development is difficult and sensitive to changes.

According to Fullerton[11], prototypes find design problems before the game goes into production, when the cost of making changes is higher. They are implemented with a specific design issue in mind. The objetive is to reduce the costs of adding complexity to the game. Fullerton and Manker explain that adding rules to a complex system produce effects that are hard to anticipate.

3.2 Game Jams

Game jams are events in which small teams must develop a game under a set of constraints. According to Musil *et al.*[16] the purpose of a game jam is to quickly prototype small games and, in doing so, inject new ideas into the industry. Their work approaches these events with a prototyping perspective. Their motivation is the difficulty of requirement elicitation in the early phases of a game development project [6]. They justify this approach with the fact that game jams were established over a decade ago, and that successful games can trace their origins to one of these events. The objective of the authors is to shape the elements that give a game jam its accelerating effect in new product development and to identify new research opportunities.

Musil *et al.* study game jams under three different perspectives: design paradigms, systems engineering processes and existing collaboration systems. For the authors, game jams deny the traditional paradigm of design as the product of a rational process, guided by rules and scientific laws. They are better framed in a pragmatic vision, according to which the design emerges from a self-organizing, reflective, know-how bricoleur system based on experience.

The authors classify game jams in no known development process, but they identify a basic systems engineering process. The team solves a problem in a recursive and iterative way, where requirements are balanced against the available development time and the team's profficiency with the tools. Iterative, because this process repeats for each feature. Recursive, because each step of an iteration may include the same steps. Tests are of the usability kind, scenario-based and focused on the final product's evaluation [16].

Musil *et al.* observe that each team implement their own vision of the proposed theme. Consequently, the final set of products is a thorough approximation of the scenario suggested by the theme. They conclude that, under a macroperspective, game jams are a pragmatical approach to game prototyping and identifying innovation opportunities.

Then, to analyze game jams at a lower level, they divide them in eight elements, shown in Figure 1. From the point of view of the selection of new products for development, game jams are practical because they offer executable prototypes, instead of abstract concepts. From a **participatory design** perspective, game jams are safe events for suggesting ideas, because of the low risk of hierarchical or role-based decisions. The team select ideas **focused on product-value** because of the time constraints, and will choose to discard them quickly, if their perceived cost is too high. They also favor a **lightweight construction** of the project, using available codebases, which reduces the product complexity and speeds up development [16].

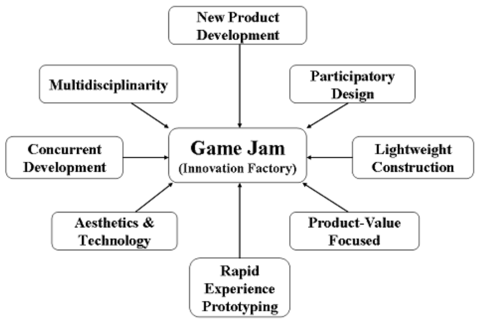


Figure 1: Elements of a Game Jam [16]

Rapid experience prototyping, because they favor subjective experiences and aesthetical integrity over technical qualities. These subjective experiences determine the balance between **aesthetics and technology**. Teams prioritize the product's utility rather than its reusability, for example. Musil *et al.* characterize game jams as set-based **concurrent development**. This means that the results of the event are a set of solutions that approximate a given problem domain. The team components come from different backgrounds, such as designers, programmers and artists. This introduces **multidisciplinarity** to the event.

3.3 Reusability Metrics

Goel and Bhatia[12] set a precedent when they use object-oriented metrics to perform reusability analysis. The six metrics, put forward by Chidamber and Kemerer[7], are:

1. **Depth of Inheritance Tree (DIT)**: measures the depth of a given class down its inheritance tree.
2. **Number of Children (NOC)**: measures the number of direct descendants of a class.
3. **Coupling Between Objects**: Number of distinct classes to which a class is coupled.
4. **Lack of Cohesion of Methods (LCOM)**: Number of disjoint sets of local methods. In this context, two sets of methods are disjoint when they share no instance variables. In a class with total cohesion, all methods share all instance variables.
5. **Weighted Methods per Class (WMC)**: Is the sum of the complexities of the methods. It can be any complexity metric, but it's usually McCabe's cyclomatic complexity [15].
6. **Response for Class (RFC)**: Number of methods of a class plus the number of calls to method and functions in the class body.

Goel and Bhatia organized these metrics in three sums. The authors state that these combinations have direct impact on the reusability of a class. They are:

1. **DIT + NOC**: Have positive impact on reusability.
2. **CBO + LCOM**: Have negative impact. Both metrics indicate a possible class subdivision.
3. **WMC + RFC**: Have negative impact. Complex methods are harder to modify for reuse.

3.4 Automation of Extract Class

Our method is a modified version of the one proposed by Bavota *et al* [3]. The method identifies classes with low cohesion and divide each class in two or more classes with higher cohesion. However, Bavota *et al.* show that this process can increase coupling between the classes. The authors explain this relation with the fact that both cohesion and coupling metrics are based on the similarity between methods. For this reason, the method uses these two metrics to produce a balanced solution. Figure 2 shows an overview of the method.

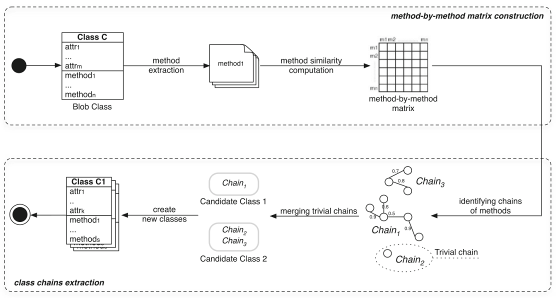


Figure 2: Class extraction process [16]

The first part of this approach is to build a $n \times n$ matrix, where n is the number of methods in the subject class. A cell $c_{i,j}$ of this matrix represents the probability that the methos i and j belong to the same class. This matrix is the representation of a graph whose edges are bonds between methods. A given threshold will filter these edges. The result is an initial set of method chains that will become the extracted classes. Method chais with a low number of methods are trivial chains. The method merges trivial chains with the non-trivial chains with which they are most coupled.

The probability that two methods belong in the same class is given by a combination of metrics. In Bavota *et al.* [3] and Bavota, Lucia and Oliveto [4], they are three: *Structural Similarity Between Methods* (SSM) [13], *Call Based Dependence Between Methods* (CDM) [4] and *Conceptual Similarity Between Methods* (CSM) [18]. These metrics are orthogonal [4]. That means they capture different dimensions of coupling between methods. However, the authors stated that the third metrics depends on the quality of the comments, and the names of the variables and methods. These two characteristics are often ignored in favor of development speed when building prototypes, as suggested by Pressman [19], Musil *et al.* [16] and Fullerton [11]. We'll ignore semantic metrics because of this fact.

Structural Similarity Between Methods is a measure of the overlapping use of instance variables between two methods of a class. Let I_i be the set of instance variables referenced by the method m_i. The SSM of methods m_i and m_j is given by the ratio between the number of variables they share and the number of variables they reference [3]:

$$SSM(m_i,m_j) = \begin{cases} \frac{|I_i \cap I_j|}{|I_i \cup I_j|}, & \textbf{if } |I_i \cup I_j| \neq 0; \\ 0, & \textbf{otherwise} \end{cases}$$

Bavota, Lucia and Oliveto proposed CDM, which measures similarity by the method calls made by methods. Let $calls(m_i,m_j)$ be the number of calls made by method m_i to method m_j and let $calls_{in}(m_j)$ be the total number of calls to m_j throughout the class. Then, $CDM_{i \to j}$ is defined as [3]:

$$CDM_{i \to j} = \begin{cases} \frac{calls(m_i,m_j)}{calls_{in}(m_j)}, & \textbf{if } calls_{in}(m_j) \neq 0; \\ 0, & \textbf{otherwise} \end{cases}$$

If $CDM_{i \to j} = 1$, then m_j is only called by m_i and must belong to the same class. If $CDM_{i \to j} = 0$, then m_i never calls m_j and, therefore, can be in different classes without increasing the coupling in the system. So there is no ambiguity in the method matrix, all metrics must be commutative. Then, Bavota *et al.* define CDM as:

$$CDM(m_i,m_j) = max(CDM_{i \to j}, CDM_{j \to i})$$

After assembling the matrix, we identify subgraphs that represent weak structural relationships between the methods. For that we define a lower boundary (*minCoupling*) and filter the edges:

$$\tilde{c}_{i,j} = \begin{cases} c_{i,j}, & \textbf{if } c_{i,j} > minCoupling; \\ 0, & \textbf{otherwise} \end{cases}$$

Bavota *et al.* studied two definitions for the *minCoupling* threshold. The first is to manually set it as a constant. It is easier to implement, but difficult to know beforehand which value is best for the codebase being refactored. The second is a variable threshold approach, which depends on the class bein refactored. The authors calculate it as the median of the values in the method matrix. The variable threshold produced better results in their experiments. The resulting subgraphs of this filtering are called *method chains*.

This set of method chains may include chains with a low number of methods, called *trivial chains*. We apply a new filter, using a *minLength* threshold, which identifies them. Next, we calculate the coupling between each trivial chain and each non-trivial chain. Then, we merge each trivial chain to the non-trivial chain with which it is most coupled. The formula for the coupling between two method chains C_i and C_j is [3]:

$$Coupling(C_i,C_j) = \frac{1}{|C_i| \times |C_j|} \sum_{m_i \in C_i, m_j \in C_j} c_{i,j}$$

,

where $|C_k|$ is the number of methods of the chain C_k.

The last step is to reassemble the resulting chains into classes. This involves distributing the fields of the original class to each extracted class according to their use by the methods. In this point out work differs from Bavota *et al.*. We relocate the extracted classes within the inheritance tree of the original class, ordered by their dependencies on the attibutes. The purpose is to increase the **DIT** measure which, according to Goel and Bhatia [12], increases reusability.

4 Application Architecture

We wrote the application in *Python*. The tool's design follows the Pipes and Filters pattern, proposed by Buschmann *et al* [5]. He defines the pattern as a chain of messages in which the output of a processing unit is the input of the next. We justify this decision with the nature of the method exposed in Section 3.4. *Filters* are components that manipulate data in a specific way. *Pipes* are components that connect filters. The source code of the prototypes are our data stream. Each step of the Extract Class algorithm is a Filter component.

The initial data stream are the source codes for the prototypes. The first filter transforms the plain text source code into abstract syntax trees that are easier to manipulate programmatically. The second filter identifies the class nodes. This step is necessary because Python is not a fundamentally object-oriented language, and one often finds different programming paradigms within the same codebase. The third filter wraps these raw nodes into a utilitary class. The purpose of this class is to abstract tree manipulations into an interface for accessing method nodes and attributes.

The fourth component builds the method by method matrix. Each cell of the matrix holds the probability that the two methods belong in the same class. The metrics SSM and CDM determine these values. This matrix represents a graph. The fifth filter applies the *minCoupling* threshold to the matrix, effectively isolating subgraphs by removing edges. The median of the values in the matrix determines the *minCoupling* threshold. The sixth component inspects the filtered matrix and outputs method chains. The seventh component identifies and merges the trivial chains with the non-trivial chains. The resulting chains are reassembled into class nodes and finally transformed into Python code by components eight and nine, respectively.

We created a subsystem called *pypeline* that implements a parallel version of the Pipes and Filters pattern. The purpose is to avoid processing bottlenecks when reading the source files or calculating the matrix. Figure 3 shows the UML class diagram of this subsystem.

The *Pipeline* class manages the connection and processing order of the filter components. The connection between filters is done by the *Pipe* class. The objects of this class feed the next filter with the data returned by the previous filter. They also buffer this data until the next filter is ready to process them.

The application is called **Cristina** and has eleven components. Each one is a *Filter* of the Pipes and Filters pattern. The project's class diagram can be seen in figures 4 through 8.

The class *CrisDataSource* reads a single Python source file or a directory and pushes the raw text source into its exit pipe. Then, *CrisCodeToAstTransformer* converts the text source into abstract syntax trees which figure 9 illustrates. *CrisClassNodeFinder* traverses these trees looking for class definition nodes (*ClassDef* nodes, seen in figure 9). The next filter, *CrisAstClassWrapper*

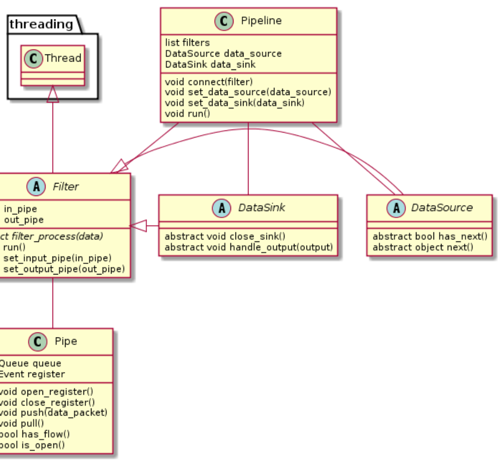


Figure 3: **pypeline** module diagram

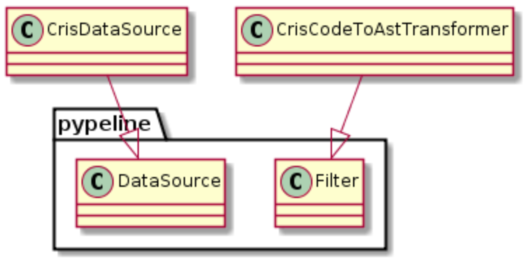


Figure 4: The first two stages of the app pipeline: the stream source and the ast parser.

wraps these nodes into utilitary classes designed to abstract raw node manipulations and provide an easy interface for accessing method nodes and symbol tables. *CrisMethodByMethodMatrix* uses this interface to build the method-by-method matrix described by Bavota *et al.*[3].

The component *CrisChainsOfMethodsFilter*, calculates *minCoupling* and filters the matrix. This filtered matrix is the input for *CrisMethodChainsAssembler*, which uses an algorithm for detecting isolated graphs within the matrix. These graphs are the method chains. The next step is to identify the trivial chains, which is done by the component *CrisTrivialChainMerger*, which also calculates the coupling between chains and merges the trivial with the non-trivial chains.

At this point, the method chains are represented as a list of function definition nodes, or *FunctionDef* in figure 9. These lists are the

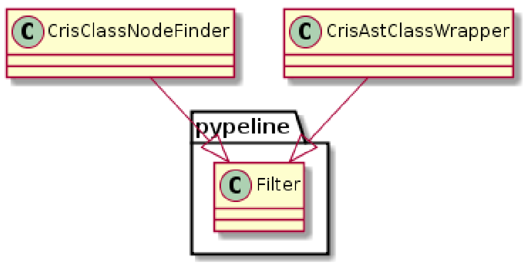


Figure 5: The third and fourth stages: the class node filter, and the ast utils decorator

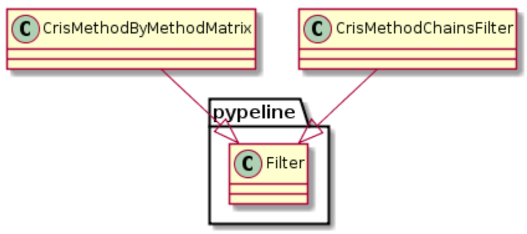


Figure 6: The fifth and sixth stages: the method matrix builder, and the method chains filter

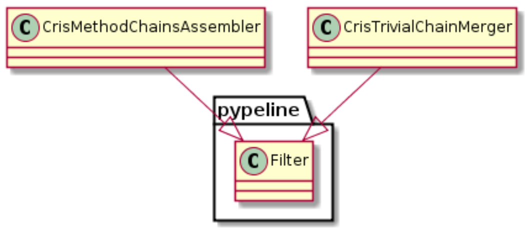


Figure 7: The seventh and eigth stages: assembling and merging the method chains.

input for *CrisClassAssembler*, that creates class definition nodes and appends the methods to them. This component also identifies the fields used by each method and, based on this information, inserts the extracted classes in the inheritance line of the original class. *CrisAstToCodeTransformer* transforms these nodes into raw source code and pushes them down the pipeline to *CrisDataSink*, that saves these codes into a file.

5 Experiment

We validate the application through an experiment designed to test the efficacy of the application in extracting classes more reusable that the original. We split the experiment in three steps. First, we measure the original source code using Goel and Bhatia's measurement technique [12]. This codebase is a sample of 30 Python source files, randomly selected from a poplulation of 322 files. These files come from submissions to the game jam PyWeek. The purpose is to establish a baseline against which we compare our results.

Next, we execute the application once for each combination of weights for the metrics SSM and CDM, and the *minCoupling* parameter. And, finally, we execute the application with a random cohesion metric, instead of SSM and CDM, to validate the usefulness of the algorithm. In short, we execute the following script:

1. For each *minCoupling* value:

   (a) Execute the application with SSM weight 0.0 and CDM weight 1.0;

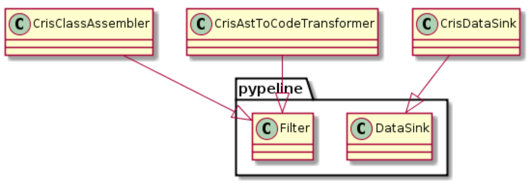


Figure 8: The last three stages: assembling the classes, transforming the asts into source code, and writing the codebase source file.

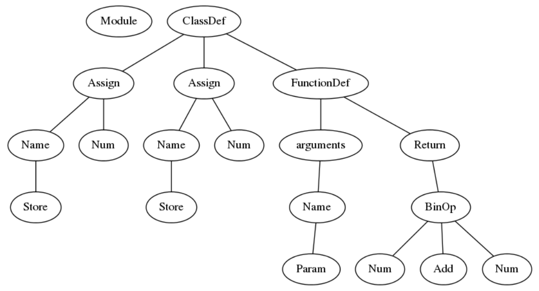


Figure 9: Example of an Abstract Syntax Tree

(b) Execute the application with SSM weight 0.2 and CDM weight 0.8;

(c) Execute the application with SSM weight 0.5 and CDM weight 0.5;

(d) Execute the application with SSM weight 0.8 and CDM weight 0.2;

(e) Execute the application with SSM weight 1.0 and CDM weight 0.0;

(f) Execute the application with a random cohesion metric;

2. Measure the resulting codebase using reusability metrics and compare these with the original measurements.

We obtain our control values when we execute the application with a metric that returns random cohesion values. The primary objective is to verify if the simple division of the methods of a class significantly enhances reusability. The secondary objective, although a consequence of the first, is to test the metrics efficiency in optimizing reusability.

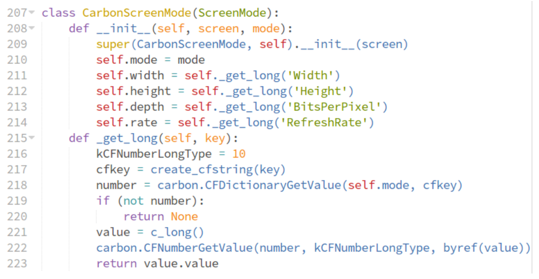
To obtain Goel and Bhatia's reusability metrics we must calculate Chidamber and Kemerer's OO metrics first. We calculate DIT in a recursive way, beginning in the leaf class and up the tree until the root class or until we find a reference error. This error can occur because the superclass may not be in codebase being refactored. Python supports multiple inheritance. Therefore, we consider the highest path until the root class as the DIT value. NOC is relatively easy to obtain, since we just count the number of direct subclasses of the target class.

Coupling Between Objects is a little difficult to calculate from a dynamic language standpoint. In most implementations, the coupling is determined by the types referenced in a class. In Python, and other weak-typed languages, we only know the type of an object in runtime. We learned that it doesn't make sense to talk of coupling by types, when dealing with dynamic languages. Instead, in Python, at least, we must consider that an object is coupled only with the foreign methods and attributes it expects in runtime. In other words, its coupling measure is the number of different interfaces it needs to run properly. We implement this by building a graph whose nodes are the external references a given class makes, and the edges are the number of local and instance variables that share this reference. Then, we identify the disconnected graphs within this matrix and the number of such graphs is equal to the coupling of the class.

We implement Cohesion of Methods (COM) by taking the number of pairs of methods that share instance variables and dividing it by the total number of pairs of methods. The Lack of Cohesion of Methods is simply $1.0 - COM$. Weighted Methods per Class is implemented as the sum of the complexity of the class methods. And Response For Class is the sum of the number of class methods with the number of calls the class makes to external methods. The last step of the experiment is to calculate the average of the measurements of each codebase and compare them.

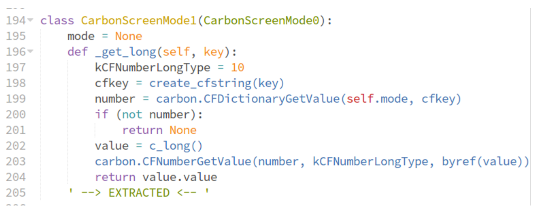
6 Results

The application was successful in extracting classes that are more reusable than the original. We illustrate the results of class extraction in figures 7 and 8. Figure 7 is a class taken directly from one of the prototypes being refactored. Figure 8 is one of the classes extracted from it. We observe that it is a subclass of *CarbonScreenMode0*, which was the first extracted class. This shows our mechanism of inheritance tree preservation.



```
class CarbonScreenMode(ScreenMode):
    def __init__(self, screen, mode):
        super(CarbonScreenMode, self).__init__(screen)
        self.mode = mode
        self.width = self._get_long('Width')
        self.height = self._get_long('Height')
        self.depth = self._get_long('BitsPerPixel')
        self.rate = self._get_long('RefreshRate')
    def _get_long(self, key):
        kCFNumberLongType = 10
        cfkey = create_cfstring(key)
        number = carbon.CFDictionaryGetValue(self.mode, cfkey)
        if (not number):
            return None
        value = c_long()
        carbon.CFNumberGetValue(number, kCFNumberLongType, byref(value))
        return value.value
```

Figure 10: Original class



```
class CarbonScreenMode1(CarbonScreenMode0):
    mode = None
    def _get_long(self, key):
        kCFNumberLongType = 10
        cfkey = create_cfstring(key)
        number = carbon.CFDictionaryGetValue(self.mode, cfkey)
        if (not number):
            return None
        value = c_long()
        carbon.CFNumberGetValue(number, kCFNumberLongType, byref(value))
        return value.value
    ' --> EXTRACTED <-- '
```

Figure 11: Extracted class

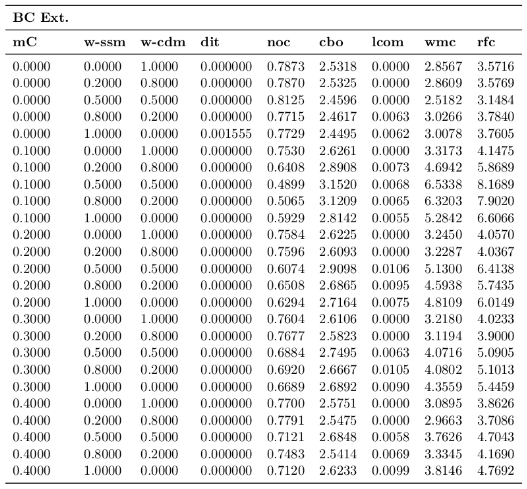
Our results are in the following three tables. Table 1 contains the measurements of the original codebase. Figures 9 and 10 show the measurements of the extracted codebases for each combination of parameters. mC is the $minCoupling$ parameter, and $w-ssm$ and $w-cdm$ are the weigths for SSM and CDM. Table 2 holds the results of the control group.

Original Codebase	
Metric	**Value**
DIT	0.1489
NOC	0.1915
CBO	2.9433
LCOM	0.0319
WMC	6.8582
RFC	8.5745

Table 1: Measurements of the original codebase.

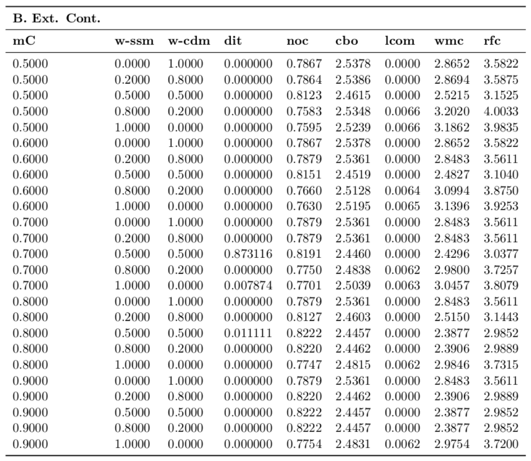
First, we must observe the increase in the NOC value relative to our baseline. The minimum value we obtained from this metric after the extraction was 0.4899, with mC set to 0.1, and both weights at 0.5. Even so, it represents a 155% increase. According to Goel and Bhatia, high NOC values are good for reusability.

Coupling Between Objects (CBO), on the other hand, has negative impact on reusability. We observe in figures 2 and 3 that this measure decreased with the extractions, but we can see a slight increase in some cases. The best case was 0.4457, with mC at 0.9 and weights at 0.5. It is a 20% improvement in class decoupling, from the baseline.



BC Ext.								
mC	**w-ssm**	**w-cdm**	**dit**	**noc**	**cbo**	**lcom**	**wmc**	**rfc**
0.0000	0.0000	1.0000	0.000000	0.7873	2.5318	0.0000	2.8567	3.5716
0.0000	0.2000	0.8000	0.000000	0.7870	2.5325	0.0000	2.8609	3.5769
0.0000	0.5000	0.5000	0.000000	0.8125	2.4596	0.0000	2.5182	3.1484
0.0000	0.8000	0.2000	0.000000	0.7715	2.4617	0.0063	3.0266	3.7840
0.0000	1.0000	0.0000	0.001555	0.7729	2.4495	0.0062	3.0078	3.7605
0.1000	0.0000	1.0000	0.000000	0.7530	2.6261	0.0000	3.3173	4.1475
0.1000	0.2000	0.8000	0.000000	0.6408	2.8908	0.0073	4.6942	5.8689
0.1000	0.5000	0.5000	0.000000	0.4899	3.1520	0.0068	6.5338	8.1689
0.1000	0.8000	0.2000	0.000000	0.5065	3.1209	0.0065	6.3203	7.9020
0.1000	1.0000	0.0000	0.000000	0.5929	2.8142	0.0055	5.2842	6.6066
0.2000	0.0000	1.0000	0.000000	0.7584	2.6225	0.0000	3.2450	4.0570
0.2000	0.2000	0.8000	0.000000	0.7596	2.6093	0.0000	3.2287	4.0367
0.2000	0.5000	0.5000	0.000000	0.6074	2.9098	0.0106	5.1300	6.4138
0.2000	0.8000	0.2000	0.000000	0.6508	2.6865	0.0095	4.5938	5.7435
0.2000	1.0000	0.0000	0.000000	0.6294	2.7164	0.0075	4.8109	6.0149
0.3000	0.0000	1.0000	0.000000	0.7604	2.6106	0.0000	3.2180	4.0233
0.3000	0.2000	0.8000	0.000000	0.7677	2.5823	0.0000	3.1194	3.9000
0.3000	0.5000	0.5000	0.000000	0.6884	2.7495	0.0063	4.0716	5.0905
0.3000	0.8000	0.2000	0.000000	0.6920	2.6667	0.0105	4.0802	5.1013
0.3000	1.0000	0.0000	0.000000	0.6689	2.6892	0.0090	4.3559	5.4459
0.4000	0.0000	1.0000	0.000000	0.7700	2.5751	0.0000	3.0895	3.8626
0.4000	0.2000	0.8000	0.000000	0.7791	2.5475	0.0000	2.9663	3.7086
0.4000	0.5000	0.5000	0.000000	0.7121	2.6848	0.0058	3.7626	4.7043
0.4000	0.8000	0.2000	0.000000	0.7483	2.5414	0.0069	3.3345	4.1690
0.4000	1.0000	0.0000	0.000000	0.7120	2.6233	0.0099	3.8146	4.7692

Figure 12: Measurements of the extracted codebase



B. Ext. Cont.								
mC	**w-ssm**	**w-cdm**	**dit**	**noc**	**cbo**	**lcom**	**wmc**	**rfc**
0.5000	0.0000	1.0000	0.000000	0.7867	2.5378	0.0000	2.8652	3.5822
0.5000	0.2000	0.8000	0.000000	0.7864	2.5386	0.0000	2.8694	3.5875
0.5000	0.5000	0.5000	0.000000	0.8123	2.4615	0.0000	2.5215	3.1525
0.5000	0.8000	0.2000	0.000000	0.7583	2.5348	0.0066	3.2020	4.0033
0.5000	1.0000	0.0000	0.000000	0.7595	2.5239	0.0066	3.1862	3.9835
0.6000	0.0000	1.0000	0.000000	0.7867	2.5378	0.0000	2.8652	3.5822
0.6000	0.2000	0.8000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.6000	0.5000	0.5000	0.000000	0.8151	2.4519	0.0000	2.4827	3.1040
0.6000	0.8000	0.2000	0.000000	0.7660	2.5128	0.0064	3.0994	3.8750
0.6000	1.0000	0.0000	0.000000	0.7630	2.5195	0.0065	3.1396	3.9253
0.7000	0.0000	1.0000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.7000	0.2000	0.8000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.7000	0.5000	0.5000	0.873116	0.8191	2.4460	0.0000	2.4296	3.0377
0.7000	0.8000	0.2000	0.000000	0.7750	2.4838	0.0062	2.9800	3.7257
0.7000	1.0000	0.0000	0.007874	0.7701	2.5039	0.0063	3.0457	3.8079
0.8000	0.0000	1.0000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.8000	0.2000	0.8000	0.000000	0.8127	2.4603	0.0000	2.5150	3.1443
0.8000	0.5000	0.5000	0.011111	0.8222	2.4457	0.0000	2.3877	2.9852
0.8000	0.8000	0.2000	0.000000	0.8220	2.4462	0.0000	2.3906	2.9889
0.8000	1.0000	0.0000	0.000000	0.7747	2.4815	0.0062	2.9846	3.7315
0.9000	0.0000	1.0000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.9000	0.2000	0.8000	0.000000	0.8220	2.4462	0.0000	2.3906	2.9889
0.9000	0.5000	0.5000	0.000000	0.8222	2.4457	0.0000	2.3877	2.9852
0.9000	0.8000	0.2000	0.000000	0.8222	2.4457	0.0000	2.3877	2.9852
0.9000	1.0000	0.0000	0.000000	0.7754	2.4831	0.0062	2.9754	3.7200

Figure 13: Continuation

Like CBO, lack of cohesion between methods indicate too much responsibility within a class, which is bad for reusability. Our results show that, in many cases, the extraction produced completely cohese classes. Our worst result was still only 67% of the LCOM measured in the baseline. It is in figure 9, with *mC* at 0.2 and both weights at 0.5.

The metrics Weighted Methods per Class (WMC) and Response for Class (RFC) are both directly proportional to the number of methods in a class. Our application is based on redistributing the methods of class between the extracted classes. So it makes sense that these measurements significantly decreased from the baseline, which is good, from a reusability perspective. However, observing the control results in table 4 we see that the random extraction of methods has little impact on reusability as a whole.

That is because, for a class with *n* methods there are many ways to combine them in different classes. For each configuration of extracted classes, there is a correspondent set of CK measurements that may not be better than those of the original class. This can happen, for example, when a minority of the extracted classes in a configuration hold most of the more coupled or less cohesive methods. This proves the optimizing factor of Bavota *et al.*'s choice of metrics, because they help find the most reusable configuration.

V. de Controle						
mC	**dit**	**noc**	**cbo**	**lcom**	**wmc**	**rfc**
0.00	0.000000	0.3260	3.9163	0.0264	8.5198	10.6520
0.10	0.000000	0.0253	5.0253	0.0506	12.2405	15.3038
0.20	0.000000	0.0723	4.8313	0.0361	11.6506	14.5663
0.30	0.000000	0.0778	4.8383	0.0539	11.5808	14.4790
0.40	0.000000	0.1250	4.6307	0.0341	10.9886	13.7386
0.50	0.000000	0.2165	4.3763	0.0361	9.9691	12.4639
0.60	0.000000	0.3108	3.9865	0.0315	8.7117	10.8919
0.70	0.003559	0.4733	3.5765	0.0320	6.8826	8.6050
0.80	0.002967	0.5608	3.2522	0.0267	5.7389	7.1751
0.90	0.006342	0.6913	2.9767	0.0085	4.0888	5.1121

Table 2: Control values

6.1 Limitations and Generalisability

A unit of code is one or more valid statements of source code. The first limitation of our approach is that we consider only classes as units of code. Reuse has two dimensions: effort and usefulness. Reuse effort is the amount of work a developer has to do to reuse a unit of code. Usefulness is how much that unit of code is applicable to what the developer is doing. In a reusability context, semantic metrics measure how much that code is semantically related to a developer's needs. So they can, to some extent, measure the usefullness of a unit of code. In a reusability context, structural metrics measure how much a unit of code is cohesive and coupled to, or dependent on, other units of code. They provide insight into the effort dimension of reuse, because a developer will have to manually resolve missing couplings or dependencies and cut off unecessary parts when reusing a unit of code. The second and third limitations of our approach is that it concerns only the effort dimension of reuse, and a specific reuse scenario, respectively. In this scenario, the developers already have a codebase of games that are not necessarily compilable/executable. Their primary requirement is that the extracted codebase is reusable, not functional. The fifth limitation is that we are assuming that the relationship between structural metrics and reuse effort is constant. This may be the general case, but there are specific areas in software development where architectural constraints would favor a unit of code with a lesser measurement, as long as that unit of code is better suited to those architectural constraints.

7 Conclusion

Game engines are important tools for the game development industry. However, their prohibitive price is the main cause for the preference of most developers to reuse their own code. Therefore, reusability is an important factor for this market segment. Our research of the related literature revealed that prototyping is more than a communication tool for interative concepts. It assumes a constructive role when used to explore a design space that, until then, would be abstract. We confirmed this exploratory role with academic works that establish an equivalency between the prototyping process and game jams. A consequence of this equivalency is the high online availability of source code for game prototypes. This availability, and the need for code reuse, was the motivation for our work. We built a tool that tranforms game prototypes in reusable code.

For that, we researched automated refactoring techniques that improve reusability. The works we studied, however, approach refactoring as a tool for improving quality as a whole. But we also observed that the quality factors under scrutiny are connected. For example, reducing the code complexity of a game project will reduce its defect rate, but will also increase its reusability. Based on this fact, we selected the work of Bavota *et al.*[3], that presents an automated refactoring technique based on class extraction by method cohesion. The authors motivation was to increase the overall quality of a system, but we conclude that the technique had a significant impact on reusability as well.

We observe this same effect in the utilization of object oriented metrics. They capture several quality dimensions at the same time, including reusability. We explore this effect to use the Chidamber and Kemerer [7] metrics suite to validate our application, a precedent set by the work of Goel and Bhatia [12]. Our results show the application's effectiveness in extracting more reusable classes from a prototype codebase. Therefore, our main conclusion is that this tool has economic potential, because it recycles development by-products to decrease costs. This fact is based on the work of Goel and Bhatia, and DeLoura's game engine survey [8]. This cost decrease is closely related with the decrease in reuse effort of the extracted classes. However, as discussed in the end of the section 6, it's not possible to know beforehand if the refactored codebase will be useful. Also, the application does not take into account architectural constraints that may affect reuse effort.

In future works we may explore the impact of the *minLength* paramaters on the results. We can also improve the refactoring method and organize the extracted classes into a game engine architecture. We can also study the impact of semantic metrics on reusability. And, finnally, validate the usefulness of the application in a case study with actual game developers.

References

[1] E. F. Anderson, L. McLoughlin, J. Watson, S. Holmes, P. Jones, H. Pallett, and B. Smith. Choosing the infrastructure for entertainment and serious computer games - a whiteroom benchmark for game engine selection. In *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2013 5th International Conference on*, pages 1–8, 2013.

[2] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, Jan. 2002.

[3] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 19(6):1617–1664, 2014.

[4] G. Bavota, A. D. Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, 1996.

[6] D. Callele, E. Neufeld, and K. Schneider. Requirements engineering and the creative process in the video game industry. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 240–250, Aug 2005.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[8] M. DeLoura. Game engine survey 2011. *Game Developer Magazine*, 18(5):7–14, 2011.

[9] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander. Decomposing object-oriented class modules using an agglomerative clustering technique. pages 93–101, 2009.

[10] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

[11] T. Fullerton. *Game Design Workshop, Second Edition: A Playcentric Approach to creating Innovative Games*. Morgan Kaufmann, 2008.

[12] B. M. Goel and P. K. Bhatia. Analysis of reusability of object-oriented systems using object-oriented metrics. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–5, 2013.

[13] G. Gui and P. D. Scott. New coupling and cohesion metrics for evaluation of software component reusability. In *ICYCS*, pages 1181–1186. IEEE Computer Society, 2008.

[14] J. Manker. Game design prototyping. *Games and Innovation Research Seminar 2011 Working Papers*, 1(1):41–48, 2011.

[15] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[16] J. Musil, A. Schweda, D. Winkler, and S. Biffl. Synthesized essence: what game jams teach about prototyping of new software products. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, (ICSE) 2010, Cape Town, South Africa, 1-8 May 2010*, pages 183–186. ACM, 2010.

[17] M. K. O'Keeffe and M. Ó. Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[18] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb. 2009.

[19] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, NY, eighth edition edition, 2014.

[20] F. Simon, F. Steinbrckner, and C. Lewerentz. Metrics based refactoring. Sept. 14 2001.