



UNIVERSITY OF STRATHCLYDE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

Analysing the use of casting in Java systems

Paul O'Hear

This dissertation was submitted in part fulfilment of requirements for the degree of
MSc Software Development

DEPT. OF COMPUTER AND INFORMATION SCIENCES UNIVERSITY OF
STRATHCLYDE

August 2019

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [☒] No [☐]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 16,389

I confirm that I wish this to be assessed as a Type 1 2 ☒ 3 4 5 Dissertation (please circle)

Signature:

Date : 18/08/2019

A handwritten signature in black ink, appearing to be 'G. J. H.', written over a horizontal line.

Abstract

This project aims to investigate the seriousness and potential problems that may occur following the implementation of casting. Casting is a Java function that is used to convert the data type of an object to access type specific functionality. However, if a system requires the continuous use of casting, there are normally resulting issues later on in the program. To prevent this, developers should consider revising the system design rather than having to repeatedly use the type conversion operator.

Having carried out an in depth investigation into the various opinions surrounding the use of casting, a software tool was developed to aid manual inspection of real life open source Java software systems. Various programs were analysed from the Qualitas Corpus, a collection of curated software systems that are used globally for research and development. The aim of this analysis was to provide a conclusion of each system to conclude whether or not they implement type conversions in an audacious manner and if code quality can be improved through the use of refactoring.

The findings of this project certainly illustrate how the use of casting can snowball throughout the program, resulting in an abundance of type checks later required. An abundance of unnecessary conversions were due to programs frequently passing in Object data types. These then required multiple type checks and explicit casting functions. Although there were various suggestions made, the project came to the conclusion that no matter how well designed a system, conversions will be required at some point in the program. Future recommendations were also included to increase the usefulness of the output produced from the analysis tool.

Acknowledgements

I would first like to thank my project supervisor Dr Murray Wood for his continuous support and patience throughout my dissertation. Particularly when I was first introduced to the visitor pattern and its complex design.

A special thanks to all my family, especially my parents. Without them, my five years of studies would not have been possible. Their continuous backing has been the foundation of my success. I simply cannot thank them enough.

Contents

.....	1
Abstract	3
Acknowledgements	4
Table of Figures	8
Table of Tables	9
1.0 – Introduction	10
2.0 Literature Review	13
2.1 Casting	13
2.1.1 Implicit Casting.....	13
2.1.2 Explicit Casting.....	14
2.1.3 Reference Variable Conversion	15
2.2 Code Smells and Refactoring	16
2.3 Static Analysis of Software	18
2.3.1 Parsing.....	18
2.3.2 The Visitor Pattern	21
2.4 Previous Work	22
2.4.1 Detection of Inheritance Hierarchy Smells	22
2.4.2 Java quality assurance by detecting code smells.....	24
2.4.3 JDeoderant.....	25
3.0 Methodology.....	27
3.1 Research Method	27
3.1.1 Qualitas Corpus	27
3.1.2 Cast Detection Tool.....	28
3.1.3 Manual Inspection	28
3.1.4 Result Format.....	28
3.2 Tool Development.....	29

3.2.1 Eclipse Java Development Tools (JDT)	29
3.2.2 ASTParser.....	30
3.2.3 ASTVisitor	32
3.2.4 Cast Expression	33
3.2.5 Other ASTNode Types	34
4.0 Analysis.....	36
4.1 Apache-Ant 1.8.4	36
4.1.1 Results	37
4.1.2 Most Cast Dense File – ‘ZipEncodingTest.java’	37
4.1.3 Most ‘Instanceof’ Dense File - ‘UnknownElements.java’	38
4.1.4 Summary.....	38
4.1.5 Conclusion	39
4.2 ArgoUML	39
4.2.1 Results	40
4.2.2 Most Cast AND ‘instanceof’ Dense File – ‘CoreHelperMDRImpl.java’	40
4.2.3 Summary.....	43
4.2.4 Conclusion	46
4.3 JHotDraw 7.5.1	47
4.3.1 Results	48
4.3.2 Most Cast Dense File – ‘Base64.java’	48
4.3.3 Most ‘Instanceof’ Dense File - ‘JavaPrimitivesDOMFactory.java’	49
4.3.4 Summary.....	50
4.3.5 Conclusion	52
4.4 Azureus (Vuze)	53
4.4.1 Results	53
4.4.2 Most Cast Dense File	54
4.4.3 Most ‘instanceof’ Dense File – ‘BEncoding.java’	55

4.4.4 Summary	55
4.4.5 Conclusion	57
4.5 Marauroa	58
4.5.1 Results	58
4.5.2 Most Cast Dense File – ‘ClientFramework.java’	58
4.5.3 Most ‘Instanceof’ Dense File – ‘RPObjct.java’	59
4.5.4 Summary	60
4.5.5 Conclusion	61
4.6 Discussion	62
4.6.1 Highest Cast Dense System	63
4.6.2 Lowest Cast Dense System	63
4.6.3 Passing Object Type Parameters.....	63
4.6.4The Use of ‘Instanceof’ Operator	64
4.7 Evaluation of Tool	64
5.0 Conclusions and Recommendations	66
5.1 Possible Future Work.....	66
5.2 Final Conclusion.....	67
6.0 References	69

Table of Figures

Figure 1 - Upcasting Example	13
Figure 2 - Widening Example	14
Figure 3 - Narrowing Cast	14
Figure 4 - Class Diagram 1	15
Figure 5 - Downcast Example	15
Figure 6 - Polymorphic Variable Example	16
Figure 7 - Syntactic Definition of 'ifThen' (Gosling, et al., 2019)	19
Figure 8 - Parse Tree Example (Spivak, 2015)	20
Figure 9 - AST example (Spivak, 2015)	21
Figure 10 - Message Chains Smell Example	24
Figure 11 - Smell Detection Graph	25
Figure 12 - Binding Example	30
Figure 13 - ASTParser.newParser() example	31
Figure 14 - Directory Selector using JFileChooser	31
Figure 15 - Recursive Method to extract '.java' files	32
Figure 16 - getChildren Method.....	33
Figure 17 - Unknown expression involved in casting	34
Figure 18 - Cast Inside for Loop.....	37
Figure 19 - Bit-Packing example	37
Figure 20 - 'UnknownElements.java' instanceof example	38
Figure 21 - example of casting between project classes	38
Figure 22 - ArgoUML type check example 2	40
Figure 23 - ArgoUML type check example 1	40
Figure 24 - ArgUML cast example 2.....	41
Figure 25 - ArgoUML cast example 1.....	41
Figure 26 - Poor code design in ArgoUML	41
Figure 27 - ArgoUML implementation on 'instanceof'	42
Figure 28 - ArgoUML cast without type check.....	42
Figure 29 - ArgoUML type checking from Object type.....	43
Figure 30 - Example of a type check method in ArgoUML	43
Figure 31 - ArgoUML casting example	44
Figure 32 - ArgoUML casting to type 'JPanel'	44

Figure 33 - GUI options available through Swings JPanel class (Boskovic, 2005)...	45
Figure 34 - ArgoUML declaring variables as type Object	45
Figure 35 - JHotDraw converting char types to byte types	48
Figure 36 - JHotDraw 'else if' chain	49
Figure 37 - JHotDraw using both instanceof and cast operator	50
Figure 38 - Project class casts in JHotDraw	51
Figure 39 - JHotDraw casting from type Component.....	52
Figure 40 - JHotDraw casting within a set method	52
Figure 41 - Azureus casting values in a 256byte construct	54
Figure 42 - Encryption keys used in DESParameters.java.....	54
Figure 43 - Azureus type check before encoding	55
Figure 44 – Azureus casting from Object data types.....	56
Figure 45 – Azureus using casting during array instantiation	56
Figure 46 - Azureus using casting inside for loop.....	57
Figure 47 - Marauroa using casting inside SWITCH statement.....	59
Figure 48 - Marauroa Type Check 1.....	59
Figure 49 - Marauroa Type check 2	59
Figure 50 - Marauroa declaring 'netMan'	60
Figure 51 - Marauroa instantiating 'netMan'	60
Figure 52 - Marauroa casting without type check.....	60
Figure 53 - Marauroa casting inside FOR loop.....	61
Figure 54 - Marauroa Cast that causes tool error.....	61

Table of Tables

Table 1 - Code Smell Examples and possible Refactoring's	17
Table 2 - Analysed systems from Qualitas Corpus	28
Table 3 - Apache-Ant Analysis Results	37
Table 4 - ArgoUML Analysis Results.....	40
Table 5 - JHotDraw analysis Results.....	48
Table 6 - Azureus analysis Results	53
Table 7 - Marauroa Analysis Results.....	58
Table 8 - Total files analysed compared to average casts per file.....	62

1.0 – Introduction

Throughout this report, references will often be made to many technical phrases and terminology which must be discussed prior to the main body if one is to understand the topics covered fully.

First off, what is casting and why is it considered to be the by-product of poor software design? It all comes down to a term called *technical debt* which is accumulated through *code smells*. The term code smell was first coined by Kent Beck (Fowler & Beck, 2000) and has since been widely discussed in the software development community. The term subtly describes what it actually is; a scent, trace, pointer to a deeper problem within the systems design that has forced the developer to implement such smells. Fowler also explains that “A *code smell* is a surface indication that usually corresponds to a deeper problem in the system.” The types of code smell greatly vary, there are five distinct categories of code smells:

- Bloaters – Large sections of code such as methods and classes that accumulate over time to become unmanageable. Other data clumps such as large parameter lists are also considered bloaters (Anon., n.d.)
- Object-Orientation Abusers – These smells occur when the principles of object-orientated programming are incorrectly applied. Examples include ‘switch’ statements and sequences of ‘if’ statements. (Anon., n.d.)
- Change Preventers – System that is designed in such a way that a change to the code in one place results in other changes required in multiple other locations.
- Dispensable – Duplicate or dead code for example that could be eliminated from the system and result in more efficient and understandable code.
- Couplers – Excessive coupling between classes such as one class accessing methods and other data fields of another class (Anon., n.d.).

All of the above code smells have the potential to add to the systems technical debt. Coined by Ward Cunningham (Letouzey & Whelan, n.d.), technical debt can be compared to financial debt. When money is borrowed, one must repay the total sum in instalments. If these instalments are not met, interest is added over time and the total sum increases, further worsening the initial problem.

Technical debt is the same, if developers continuously elect a quick fix rather than dedicating more time to find the root of the problem, the technical debt will accumulate. If this debt is given no attention and no 'repayments' are made, it will continue to grow until it is near impossible to implement any changes to the software. In the worst case scenario, the project will have to be discarded or started again, this is called technical bankruptcy (Girish, et al., 2015). If teams of developers are not careful and do not communicate effectively, technical bankruptcy can be inevitable when new code is pulled together.

Fowler and Kent also go on to identify and classify many code smells and the steps that should be followed to overcome their necessity by refactoring the system design. However, there is one distinct code smell that is not covered, casting. In fact, compared to all other code smells and refactoring recommendations, typecasting and other conversion types are rarely mentioned in literature. Casting is the implicit or explicit conversion of a variable type, this can range from primitive types such as Integers and Doubles to Object data types such as converting from one class to another.

- Primitive Type Conversions – Specifically explicit conversions will be the main focus of this report as an assumption is made that any automatic conversions (widening) will pose no threat to the system (Sierra & Bates, 2015). However, the conversion of a '*double*' to an '*int*' for example will be scrutinised.
- Object Type Casting – Unlike primitives, reference variables simply refer to an object and do not contain the object itself (baeldung, 2019). Although upcasting is frequently implicitly performed by the compiler, the report will still discuss the impacts it may have along with an in-depth discussion of downcasting.

In the following chapters, the use of casting in the Java programming language will be investigated. Extensive research will be carried out to conclude if casting should be considered a code smell at all or if it can be used impetuously throughout Java systems. As most casting errors only arise at runtime, a tool is to be developed to identify all instances of casting in a project and identify the Object or Primitive type before and after the cast. If successful, the tool will also be able to identify some metadata about the cast and the potential effects it may have in the future.

- The following chapter highlights previous literature that discuss code smells, specifically the use of casting and the impacts it can have on the overall performance and integrity of Java systems.
- A review of previous projects will also take place to determine any recurring focal points when developing code analysis tools. Additionally, the various software packages available to efficiently build such a tool within the time constraints set.
- With consideration of all methodologies, a static analysis tool will be developed to aid developers in identifying cast instances and their severity.
- Lengthy testing will follow and results will be recorded to determine if analysis tools can compete against human intuition when identifying the use of casting and the information it provides about a systems design.
- To conclude this report, further discussion will take place on the ways in which the development of analysis tools can be improved and the proposition of future work.

The report will conclude with a reflection of all findings and personal opinions on the topic as a whole.

2.0 Literature Review

This chapter will include in-depth reviews of numerous literature that not only discuss the use of casting in Java systems but also cover; code smells and when they were first discussed as well as the root problems in which they originate from. The various types of casting will be covered and the errors that can arise if used incorrectly such as 'CastClassException' at runtime and similar issues that can go unnoticed by developers. Relationships between data types in a system are key to the implementation of casting which is why the concept of *Hierarchies* will also be deliberated and their significant importance in object-oriented programming.

An appreciable amount of knowledge is required to analyse software systems. Therefore, the key areas such as parsing, abstract syntax trees and visitor pattern will be covered in this chapter. Additionally, a considerable amount of time will be allocated to research the various software packages that have previously been used to develop analysis tools, specifically those that cover the recognition of casting, if any.

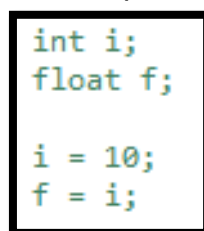
2.1 Casting

Casting or otherwise known as type casting and type conversion, is the process of implicit or explicitly changing a data type from one to another for a specific reason.

However, the function does not have any impunity that can be used haphazardly. There are unique rules when it comes to casting that if one does not abide by, can result in prolonged amounts of time trying to refactor code with hidden runtime errors.

2.1.1 Implicit Casting

This is most commonly used when assigning variables to data types. However, especially in Java, the rigorous type checking ensures that the location and destination of the cast are compatible. Implicit casting or Coercion (Schildt, 2007) occurs when this compatibility is valid and variables can automatically change type. Figure 1 demonstrates a very simple coercion example and the way it can be used in Java.



```
int i;  
float f;  
  
i = 10;  
f = i;
```

Figure 1 - Upcasting Example

The validity of this code stems from the fact that variable 'i' is changing from primitive type 'int' to the larger primitive type 'float', this is called widening. As float can store values with greater precision than integer types, the Java type check allows the conversion. This is also the case when converting any data type to a destination that is larger than the source type. Figure 2 shows the automatic conversions that are authorised due to widening.

byte → short → int → long → float → double

Figure 2 - Widening Example

However, when changing from a float to an integer, for example, an explicit cast is required.

2.1.2 Explicit Casting

Explicit casting is required when the source and location data types are compatible but cannot undergo automatic conversion. Following on from the previous example, explicit casting is necessary when converting a variable data type from a float to an int. As int is a smaller, less precise data type, programmers must use the cast function to explicitly carry out the narrowing conversion. This is done by surrounding the target type in parenthesis before the object that is changing type. Figure 3 demonstrates the explicit use of casting.

```
double x, y;  
  
x = 4555258624.78;  
y = 1;  
  
int result = (int)(x/y);
```

Figure 3 - Narrowing Cast

Figure 3 is an example of when a whole number is required even though the two values in the sum may be doubles with decimal points. Although this conversion is perfectly valid it must be used very carefully as the system is losing precision due to the decimal point being dropped when converting to an integer. If this form of casting is repeated multiple times, the integrity of the data being produced on the console, for example, is highly questionable. Additionally, if the result of the sum is greater than the maximum value that an integer can hold, information will be lost and that maximum will be displayed instead.

2.1.3 Reference Variable Conversion

Explicit casting is not limited to primitive data types. It can be used in various manners such as converting a String to an int using 'Integer.parseInt(String)'. However, the most common application is converting an objects data type from a superclass into a more specific subclass to enable a greater range of functions. This is called downcasting. Upcasting (converting a subtype to a super type) is not necessary in Java as one can call superclass methods automatically. However, downcasting is a more delicate process that requires care and attention otherwise the infamous ClassCastException may be a recurring theme. An example that is regularly used to explain this is the use of an Animal superclass with Cat and Dog subclasses.

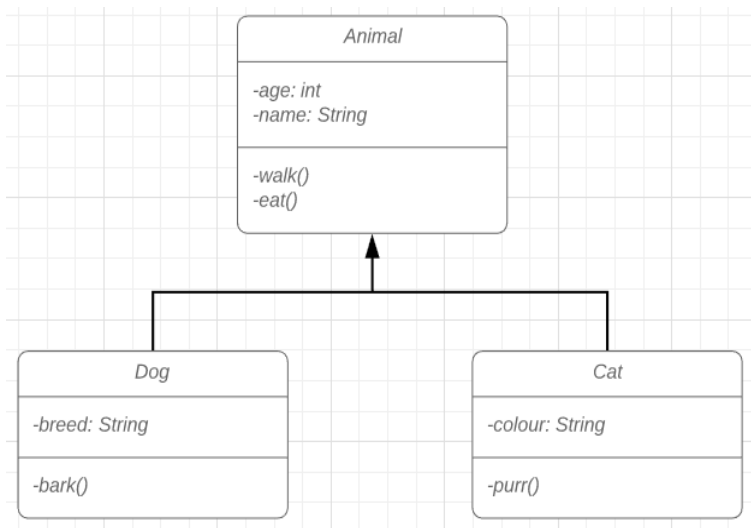


Figure 4 - Class Diagram 1

Figure 4 represents the three classes along with their attributes and methods. Note that a dog can walk, eat and bark. However, an Animal can only walk and eat. If there is a collection of animals, the code in Figure 5 can be used to differentiate between Cat and Dog and call upon the appropriate method.

```
for (Animal animal : allAnimal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        dog.bark();
    }
    else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        cat.purrr();
    }
    else {
        animal.walk();
    }
}
```

Figure 5 - Downcast Example

If the variable `animal` was in fact instantiated as an `Animal` and then one attempts to cast it as a `Dog`, `ClassCastException` will occur at runtime.

Furthermore, if a `Dog` is upcast to an `Animal`, it will lose its bark capabilities until it is cast back down to a `Dog`, just as a `Cat` would with `purr`. Both `Cat` and `Dog` object can be considered polymorphic as they can “refer to a variety of class types during a single execution” (Budd, 1991). If for example, we want to ‘walk()’ all `Animals` without caring about the type of animal. Figure 6 shows how this can be done.

```
List<Animal> animals = new ArrayList<>();
animals.add(new Dog());
animals.add(new Cat());
for (Animal animal : animals) {
    animal.walk();
}
```

Figure 6 - Polymorphic Variable Example

The new `Cat` and `Dog` are automatically upcast to an `Animal` implicitly, added to the list of `Animals` and then walked. The program can then later implement the ‘instanceof’ function to identify which of the `Animals` are of type `Cat` and `Dog`. However, to access their functionality, the object would have to be cast back to its original type. In real life systems, this form of inheritance can have multiple depths with a large number of classes and methods which is why it is important to know where and when to use the cast function.

2.2 Code Smells and Refactoring

As previously mentioned, the term ‘*code smell*’ was introduced by Kent Beck in the book ‘*Refactoring: Improving the design of future code*’ (Fowler & Beck, 2000). As this book was published in the year 2000, some of the ideas have already been explored. However, to ensure a solid foundation has been set for the remainder of the report, an in-depth analysis will take place of the full text. Beck explained that developers have learned to look for specific patterns and indicators that tell them a possible refactoring is in order to improve the code. Which is why he allocated these indicators the name “Smells” because developers can follow the scent to the root of the problem and decide whether or not the underlying problem will have a detrimental effect on the program. Beck along with Martin Fowler, co-wrote the chapter ‘Bad smells in code’ and detail many different instances of code smells that could be erased with the use of refactoring.

Smell	Explanation	Refactoring Suggestion
Temporary Field	When an instance variable is instantiated and used only in certain circumstances. One would expect a class to use all of its variables and therefore can be confusing to anyone reading the code.	<i>Extract Class</i> is considered a solution with all code that implements the variable in question transferred over.
Inappropriate Intimacy	The fundamentals of Object-Oriented programming say that classes should know as little about each other as possible. They should not be intruding into private fields and using them elsewhere.	Use <i>the Move Method</i> to relocate the method to the class in which it is used most often.
Middle Man	If a class has multiple methods that simply delegate requests to other classes.	Remove the methods that delegate and force the request to deal with the required class directly by introducing an additional getter for example

Table 1 - Code Smell Examples and possible Refactoring's

Table 1 provides some examples of code smells, why they are considered bad practice in Java and possible refactoring techniques to eliminate each smell.

Fowler lists many common code smells and provides a very helpful guide on not only how to refactor but where it is most likely required. However, he explicitly states that his literature is only the beginning of an in-depth topic with the potential of great expansion. The final chapters are co-written by Don Roberts, John Brant and Kent Beck and cover the future of refactoring. Roberts and Brant go into great detail about the lack of tool support for refactoring code, in order to cut cost and time spent doing so. They firmly believe (with the support of Beck) that with the aid of automated tool support, developers can drastically cut the time spent refactoring. Furthermore, they insist that developers will be less likely to turn a blind eye to time-consuming refactoring's resulting in higher quality code. It is important to note that although this book is a great introduction to the topic, it either does not consider casting to be a severe enough code smell to feature in the book or the numerous authors do not yet consider casting to be a smell at all.

2.3 Static Analysis of Software

Static analysis of software involves taking the syntax of a system and analysing it for possible defects without any dynamic execution. This form of analysis can identify problems in the system at an early stage and offline if necessary. Static analysis can be performed by both humans and machines.

Humans can manually scan through the code and ensure the programmer has followed Object-Oriented conventions for example. Whereas, machines can analyse the code for lexical, syntactic and possibly semantic errors (Ghahrai, 2018). Static analysis is frequently carried out after coding but before extensive testing is performed, it allows developers to eliminate both silly and more complex flaws in the code before integration. However, the process of static analysis can be time-consuming if done manually and it requires trained personnel that can be in short supply if it is a small company. Analysis tools can greatly improve the efficiency of the process but can reduce accuracy and consistency by producing both false positives and negatives (Acellere, 2017).

2.3.1 Parsing

In the Oxford English Dictionary, the phrase parsing means to “Resolve a sentence into its components and describe their syntactic roles according to a given grammar” (Anon., 2019). However, this is not limited to an English sentence, it can be a computer program, a piece of music, a sequence of geological strata and even a knitting pattern. The grammar for parsing computer programs (syntax analysis) is a set of rules that defines how the syntax is broken down and how each construct can be composed. To parse a Java system, two grammars are required.

- Lexical Grammar – A lexical grammar defines the structure of each token in a program. Tokens are a type of lexeme which are the smallest elements of a program that are purposeful to the compiler. These include operators such as ‘+’, ‘-’, ‘%’ and ‘*’ as well as keywords like ‘int’, ‘class’ and ‘return’ and all other Java operations and statements. Elements like white spaces and comments are automatically discarded (Gosling, et al., 2019).

- Syntactic Grammar – A syntactic grammar uses the tokens defined by the lexical grammar and describes how each token produces syntactically correct systems. Each programming language has a unique syntactic grammar specific to its syntax.

Lexical and Syntax analysers follow the rules of each grammar to produce a parse tree. As such the parse tree is constructed following one of two methods. For Java, the context-free grammars can be accessed and inspected to learn about the language specifications (Gosling, et al., 2019). Figure 7 illustrates the syntactic definition of an 'if-then' statement. An 'If' token will only be recognised if it is accompanied by a left parenthesis, an expression, a right parenthesis and a resulting statement. This is the highest level representation of the definition as both 'Expression' and 'Statement' will have large definitions of their own.

```
if ( Expression ) Statement
```

Figure 7 - Syntactic Definition of 'ifThen' (Gosling, et al., 2019)

- Top-Down Parsing – The construction of the parse tree starts at the root and works through the syntax deciding where the next token belongs in the tree then and there. After each token has been analysed, the tree is complete. 'Lookahead' can be used which allows the parser to inspect the next token before placing the current one. However, anything other than simple expressions can cause major problems during the production of the tree as anything more than a lookahead of one is very expensive. Top-down parsing also finds the left most deviation when constructing the parse tree, this means it applies the rules of the grammar on the leftmost derivation (reads left to right).
- Bottom-up Parsing – Similar to top-down but more general, just as efficient and more common to use in practice. Bottom-up parsing or an LR parser reads tokens from left to right and traces a rightmost derivation in reverse. In doing so, the parse tree is constructed starting from the leaves and working towards the root.

If the construction of the parse tree is successful, each node in the tree denotes a syntactic construct in the Java source code. Parse trees are profoundly complex and can grow quickly grow from a small section of code. Figure 8 is the corresponding parse tree for the simple statement '7 + ((2+3))'. Notice that all parts of the syntax are represented with a node, even parenthesis.

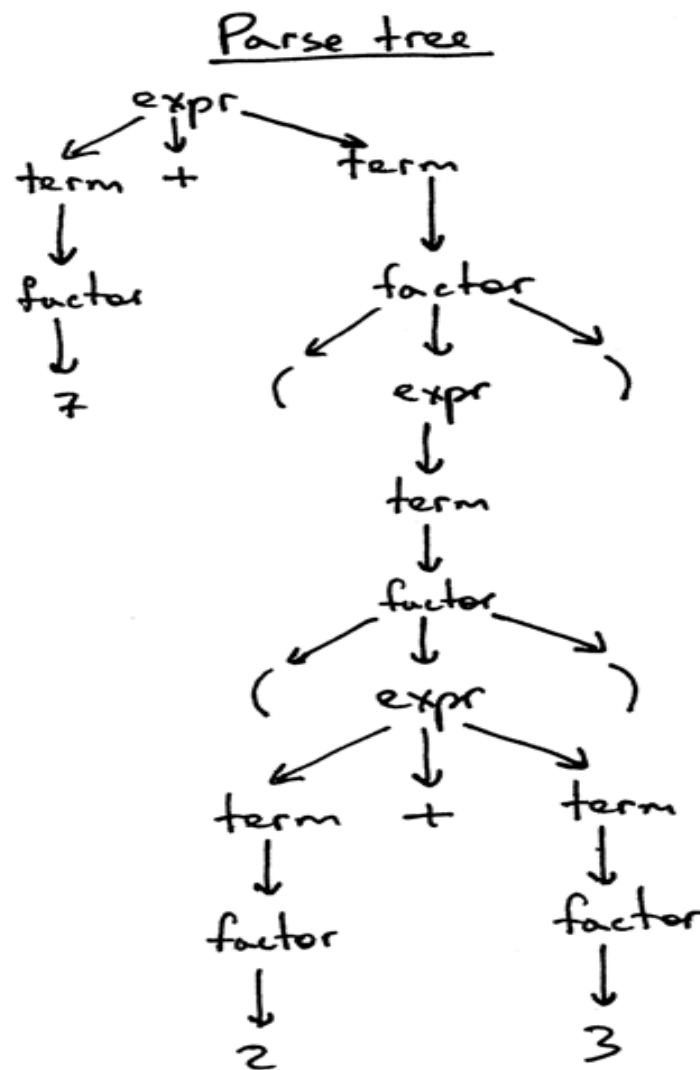


Figure 8 - Parse Tree Example (Spivak, 2015)

For this very reason, more often or not, Abstract Syntax Trees (AST) are used instead of the sometimes unnecessary parse trees. The key difference is that it is abstract, meaning they do not display needless or redundant data within the tree. Parentheses and grammar rules, for example, are removed. Moreover, nodes of the tree are made up of operations/operators and operands are used as their children. As a result, an AST is considerably smaller, more compact and easier to work with both visually and programmatically. However, key data is still included so there is no loss of information and the input can clearly be identified. Figure 9 is an AST of the same phrase used to produce the parse tree in Figure 8 - Parse Tree Example . Although less than half the size, the same crucial information is characterised.

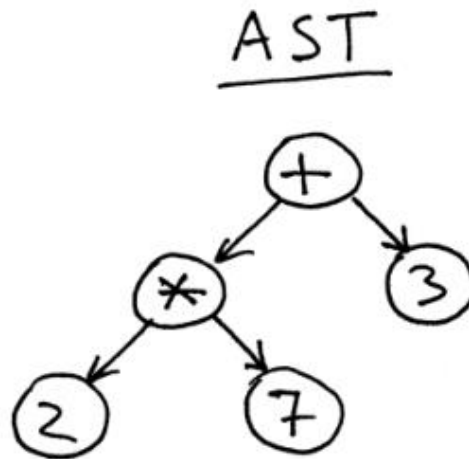


Figure 9 - AST example (Spivak, 2015)

Despite the simplistic design of an AST, they too still have the potential to become very convoluted when representing larger real-life systems, classes can also become crowded with unrelated operations. Therefore, it is essential to be able to efficiently traverse through the tree otherwise the effort to create it would be in vain.

2.3.2 The Visitor Pattern

The visitor pattern is a design pattern that allows programmers to define new operations without changing the type or classes of the element that it is currently operating on (Sciore, 2019).

When implemented on an AST, the visitor must first be 'accepted' by each node. The visitor will then determine the type of the current node and execute the defined operation for that type. Essentially, the node lends itself to the visitor as a parameter to let the visitor access its state (Anon., 2011). Programmers can further expand the functionality of visitors by creating new NodeVisitor subclasses and developing actions to be taken for every type of node in the system. A few of many advantages of using a visitor include (Anon., 2019);

- Able to add functions to class libraries in which you are not able to change the source.
- Able to attain data from unrelated classes to obtain overall results and identify data patterns.
- Develop all related operations in a single class rather than trying to adapt current classes to add the same operations.

The main aim is to encapsulate methods used for obtaining data from many classes that have different interfaces. The key to developing a successful visitor is to define a 'visit()' method for each concrete derived class within the AST (Anon., 2019). Also, only a single argument can be passed to each visit() method which directs the visitor to a specific node type. Each base class or in this case AST must also employ an 'accept()' method that again, receives a single argument. This argument is a reference to the particular visitor within the Visitor hierarchy that is to traverse through the tree visiting specified nodes.

Having set an adequate background to the topic of casting and static analysis, the remainder of the chapter will detail previous research, projects and work that has been carried out. Projects with similar aims to this will be inspected in order to discover if there are any trends and/or recommendations about developing a static analysis tool. Their methods of development along with other findings will be discussed and recorded so to consider all options and compare with the final results obtained from this project as a whole.

2.4 Previous Work

2.4.1 Detection of Inheritance Hierarchy Smells

In 2017, a former student of the University of Strathclyde, Ioannis Ziamos, carried out an investigation into the detection of inheritance hierarchy smells (ZIAMOS, 2017). Having read the book "Refactoring for Software Design Smells: Managing Technical Debt" (Suryanarayana, et al., 2015), Ziamos aimed to investigate the smell patterns identified by the literature and develop a tool that can extract such smells from system source code for review.

The Eclipse JDT framework and the ASTParser library were used to create a list of names of each class in the system, any class it may extend and any interface that it may implement. Additionally, the names of each type and variable were recorded. Having gathered all this information, Ziamos was able to link each class and illustrate each hierarchy within the system. Multipath hierarchy smells were detected by comparing any implemented interfaces of each class and each superclass. This is only one of the many hierarchy smells detected.

The use of the tool was based on a command-line interface. By providing the path to a directory, users were able to list each inheritance hierarchy smell in accordance with their needs. They could also view individual hierarchy extracts from within the system.

Ziamos carried out a very thorough investigation into inheritance hierarchy smells and successfully developed an efficient tool for developers to use in practice. He finds that there are instances of each studied smell in all systems analysed. He also goes on to explain the usefulness of the tool and that there is “much utility to performing static analysis of this nature”. The report recommends that the tool should not be used as the sole method for detecting code smells. Manual inspection is still required by developers for conformation of each instance detected by the tool. However, the tool still provides an efficient pointer towards each code smell and allows future developers to quickly learn the overall design of the system in question.

2.4.2 Java quality assurance by detecting code smells

E. van Emden and L. Moonen both based in Amsterdam, Netherlands try to develop a tool that aids automatic code inspection by attempting to identify the presence of many code smells simultaneously (Emden & Moonen, 2002). They believe that they can assess the overall quality of code by pinpointing things like “code duplication”, “long methods” and “message chains”. Message chains are when a client requests an object and that object then requests another object. When this pattern continues, the ‘Law of Demeter’ is violated (Anon., 2019). Figure 10 gives and examples of what a message chain may look like.

A code snippet showing a message chain: `return getAddress().getCountry().isInEurope();`. The code is highlighted with a black border.

Figure 10 - Message Chains Smell Example

These are only a few of the smells that the paper suggests the tool will identify. However, the report states that many of the smells they wish to detect are subject to change depending on the user. They continue that code smells are based entirely on personal opinions and experiences and the tool should be able to accommodate these differences.

The idea of this report is that each code smell has a number of ‘smell aspects’, the tool will analyse the system and flag up smells only when all of its aspects are found. To do so, Emden and Moonen specify that a static analysis of the system is all that is necessary for the successful completion of the task at hand. They use the ‘Asf+sdf Meta Environment’ that aids the production of parser generators. A custom parser was created and used to parse each system, producing a parse tree. A custom analyser was also developed to traverse the parse tree and store data on each element of the system such as classes, methods and constructors as well as relations between each of these entities. Interestingly, the method used for detailing the results is not that of a table that may have included the total for each smell but instead, a visualisation tool was used to illustrate the results in the form of a graph model.

After providing their prototype, jCOSMO, for real-life testing, a user interface was later integrated as the original instruction was too complex to efficiently apply to systems. However, following this, a great success of the tool was reported with Emden and Moonen now considering their tool to be a key point in the development cycle of automated code inspection and quality assessment.

Figure 11 is an example of the output from the prototype. The graph model not only details the code smells present in the system but also gives an insight into the overall structure of the system.

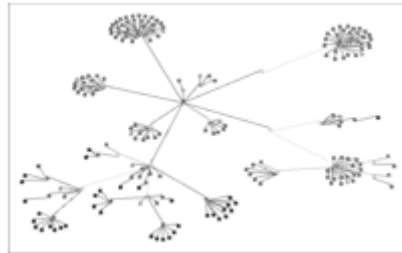


Figure 11 - Smell Detection Graph

2.4.3 JDeoderant

JDeoderant is a plug-in that was developed for the Eclipse environment and is used to identify feature envy code smells in Java systems. A feature envy smell occurs when a method uses more methods of another unrelated class than that of its own class. Data and functions should traditionally be in the same place to avoid having to access other classes unnecessarily. JDeoderant not only extracts all instances of feature envy smells but also improves the overall quality of the code by applying an automatic refactoring.

Developed by Marios Fokaefs, Nikolaos Tsantalos and Alexander Chatzigeorgiou, the main aim of JDeoderant was to give the user the “ability to pre-evaluate the impact of all possible move refactoring’s on design quality and apply the most effective one” (Fokaefs, et al., 2007). Fokaefs et al identified the feature envy smell by measuring the distance between a method and the class that it has accessed. The greater the distance, the greater the dissimilarities between the method and the corresponding attributes of the class it is accessing. A specific smell is flagged if this measurement is less than the measurement between the same method and the class it belongs to.

Like Ziamos and the detection of hierarchy smells, JDeoderant utilises the ASTParser in Eclipse JDT. Additionally, the ASTRewrite operator was employed so that the tool could apply move refactoring’s on system source code. By implementing a move method refactoring, users are able to improve the quality of code by increasing encapsulation and decreasing over intimacy of classes.

The tool was tested on real-life systems and was able to successfully identify six out of six and seven out of eight feature envy bad smells that were manually selected beforehand.

Having closely analysed the previous work discussed in this chapter, it is clear to see that there is a recurring aim to further develop automatic code inspection and quality checks. Although many code smells come down to personal opinions, the use of inspection tools are still highly beneficial as they can provide developers with the knowledge about the system and areas that could possibly be improved, without having to manually inspect each line of code. Having covered various case studies and examined the underlying knowledge of static analysis, the desire for improved code inspection tools for more and more code smells is growing which leads into the practical development section of this report. With consideration of previous work and other research carried out on tool development, a design process will be drafted and finalised to establish the best software and approaches that are best suited for this project. The aim is to contribute towards the automated code inspection lifecycle in the form of analysing systems for casting code smells and other type-changing operators.

3.0 Methodology

3.1 Research Method

A custom tool has been developed with a view to aid developers during static analysis of Java systems to determine if there is an overabundance of cast operations used within the program. Furthermore, the tool aims to provide analysis results that can identify all instances of casting and the metadata of each instance. This metadata will aim to include features such as the object type that is being cast and what it is being converted to. The accuracy of the tool will also be tested to ascertain whether or not the tool is able to, first of all, locate all instances of casting and if so, is it maintaining data integrity when displaying final results. Irrespective of the success of the tool, each system under investigation will undergo manual inspection to try and determine why a particular volume of casting has been used and why the syntax regularly requires other type-check functionality so the report as a whole can provide useful findings on the subject.

Following the development, it is essential to assess all aspects of the tool on real-life systems that vary in size and complexity. This allows both benefits of the tool to be recorded and areas of systems that the tool may struggle with. Additionally, it provides a fair analysis and allows comparison with other tools of similar functionality such as the one developed by Ziamos to identify hierarchy smells. With this in mind, a request was sent to Dr Ewan Tempero to gain access to the Qualitas Corpus so to select a number of systems that will provide a diverse range of results for analysis (Tempero, n.d.).

3.1.1 Qualitas Corpus

The Qualitas Corpus is a collection of open-source systems intended to be used for research purposes to enable 'reproducible' studies of software (Corpus, 2013). The collection was first constructed as many software investigations were not detailing the systems they studied. Therefore, the validity and accuracy of the findings were unknown. The most recent release of the collection 20130901r included multiple open-sourced Java software systems. A number of these systems will be evaluated using the developed tool to fully test its efficiency, functionality and overall usefulness to developers during static analysis.

The selection of such systems was an iterative process due to the fact that some systems did not include Java source code or have any instances of the cast operator. Although all results were recorded, this report will only go into the details of the systems that provided a means of analysis by providing results that can be compared to other systems. The final set of systems are displayed in Table 2.

Name	Version	Domain	Number of '.java' Files
Apache Ant	1.8.4	Parsers/generators	1196
ArgoUML	0.34	Diagram Generator	1922
JHotDraw	7.5.1	3d/Graphics/GUI	613
Azureus (Vuze)	1.8.1.2	Databases	3319

Table 2 - Analysed systems from Qualitas Corpus

3.1.2 Cast Detection Tool

A custom tool has been developed specifically for this project to analyse the use of casting in Java systems, the development, implementation and results of the tool are detailed in the following chapters. The tool can detect all instances of casting in each system and other data that surrounds the use of casting and code smells such as the use of the 'instanceof' operator that was included later in development to provide a means to further analysis. The tool is applied to all the Java systems selected from the Qualitas Corpus with all results being recorded.

3.1.3 Manual Inspection

Since developers can cast in a huge variety of ways, it is extremely difficult to enable the tool to provide extra information about all cast instances within the time constraints set. Therefore, any instance that has not been accounted for within the tools code will be listed. Manual inspection will be used to identify any recurring patterns that can provide extra information surrounding the system, its design and the general approach of each system to using the cast function.

3.1.4 Result Format

For each system, a 'summary' method will be called to print the numerical results of each measured variable. Corresponding files will also be presented to aid manual inspection of the densest uses of casting and type-check functions. Users also get the option to print out all cases that the tool was unable to gather information for. This can be for various reasons like null pointers or message chains.

The findings from both the tool and manual inspection will be discussed and possible improvements of the tool will be suggested to increase the accuracy, efficiency and automation of the analysis process. It must be noted that only the final results are included in this report. However, this was an iterative process with continuous changes being made after each simulation in order to optimise both the tool and the usefulness of its output.

3.2 Tool Development

As previously discussed in chapter 2, there are a number of possible approaches to statically analysing source code. However, having to create and develop custom grammar rules, a custom parser for parsing java source code that produces an AST and a custom visitor for traversing through the AST identifying all nodes that are of cast expression type can be a very time consuming and complex task. Fortunately enough, this can all be done in a single environment with all functions and libraries installed as standard.

3.2.1 Eclipse Java Development Tools (JDT)

This is a framework of the Integrated Development Environment (IDE) Eclipse and comes with all the necessary plug-ins that support the development of any Java application (EclipseFoundation, 2019). It is commonly used for the design of plugin projects as it is especially useful for manipulating Java source code. However, the framework also allows the development of standalone projects separate from the IDE. To use the tool for static analysis of source code, there is one library that is part of the JDT framework as standard and must first be imported to the workspace to enable the functions required. This library is:

`'org.eclipse.jdt.core.dom'`

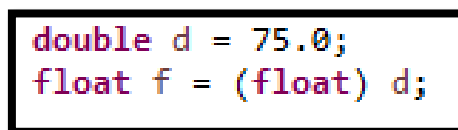
There was a considerable amount of learning that was required before the development of the tool could even begin. Initially, a JDT Plug-in project was to be produced so that it could be integrated with the IDE and possibly other development environments. However, it was established that a standalone project would be preferred to allow the tool to analysis systems without them having to be present in the current workspace.

3.2.2 ASTParser

The ASTParser class is a Java language parser that has pre-defined grammar rules and produces an AST from a continuous string input. Using the 'ASTParser.setKind(ASTParser.K_Compilation_Unit)' method, the parser produces a single CompilationUnit object for each Java source file. A CompilationUnit is the highest-level syntactic structure recognised by Java. The resulting CompilationUnit is then used to create the corresponding AST for each file. The ASTParser can also be configured depending on the type of project and its requirements.

The configurations important to this project are the 'ASTParser.setKind ()' that has been mentioned above as well as 'ASTParser.setResolveBindings()' and ASTParser.setEnvironment()'.

ASTParser.setResolveBindings () – This must be set to 'True' to instruct the compiler to provide extra binding information for each ASTNode in the AST. The importance of resolving bindings in this project allows the tool to inspect the expression that is being cast and the class it was originally bound to.



```
double d = 75.0;
float f = (float) d;
```

Figure 12 - Binding Example

Figure 12 shows variable 'd' being instantiated as the primitive type 'double'. Subsequently, by resolving the binding of 'd' in the second line, 'double' will be returned. However, it is being narrowed to a float which is why the explicit cast operator '(float)' is used.

ASTParser.setEnvironment () – This sets the environment of the parser as, by default, aims to parse source code in the current workspace if no Java project is provided. As the aim is to develop a standalone application, it is important to allow the tool to import Java systems without having to be imported into the same workspace environment. Therefore, the tool must be configured in a way that allows external Java source code to be analysed.

When instantiating a new ASTParser the method `ASTParser.newParser()` is used. This method takes a single parameter which is important to set correctly. The parameter is the version of Java Development Kit that is currently installed. Figure 13 shows how method must be used within the tool, 'JLS9' is used as it is the most up to date Java language specification that supports JDK 1.8 (the version of Java that the tool is based upon).

```
ASTParser parser = ASTParser.newParser(AST.JLS9);
```

Figure 13 - `ASTParser.newParser()` example

Although the utilisation of ASTParser is relatively simple, the input must first be converted into a String before parsing can begin. Thus a number of methods had to be developed to do the following.

1. Direct the tool to the relevant system that the user wished to analyse. Initially, this was done by manually inputting the full path of the directory. However, with continues testing and frequently changing the target system, the `JFileChooser` class was implemented. By doing so, a file browser window shown in Figure 14 is displayed at the beginning of each simulation allowing users to select any locally stored directory.

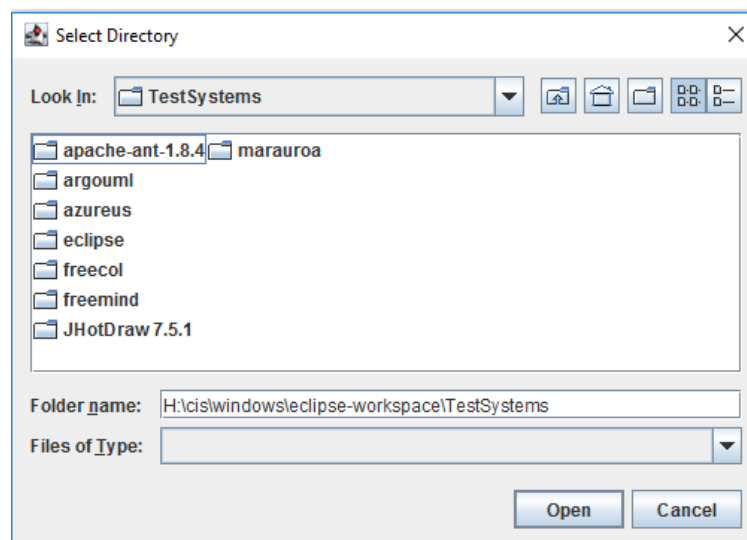


Figure 14 - Directory Selector using `JFileChooser`

2. All files of the selected directory were then extracted but only the '.java' files were temporarily stored for analysis. This was done by developing a recursive method that calls upon itself to open each internal directory of the chosen system. Figure 15 displays the recursive method explained. It takes a directory path as a String and an empty list of files as parameters.

```
public static void getJavaFiles(String directoryPath, List<File> javaFiles){
    File directory = new File(directoryPath);
    File[] allFiles = directory.listFiles();

    if(allFiles != null) {
        for(File file : allFiles) {
            if(file.isFile() && file.getName().endsWith(".java")) {
                javaFiles.add(file);
            }
            else if (file.isDirectory()) {
                getJavaFiles(file.getAbsolutePath(), javaFiles);
            }
        }
    }
}
```

Figure 15 - Recursive Method to extract '.java' files

If the method '.isDirectory()' returns true, the method is executed again for the newfound directory. Otherwise, it extracts the files that end in '.java', adds them to the directories list of files and returns the list once all internal directories have been opened and checked.

3. Each Java file was then converted into a string of characters using the StringBuilder class. Only then can the ASTParser class be used to parse each Java file using the corresponding String as a single parameter. Producing an AST and compilation unit which is required to accept and implement the visitor pattern.

3.2.3 ASTVisitor

The visitor pattern is considered one of the most complex design patterns to implement and although this report has provided a brief explanation into the background of the pattern, developing a custom visitor is a very intricate process. Which is why Eclipse JDT is so beneficial by providing a pre-constructed ASTVisitor class. Developers are able to create a custom concrete visitor class of their own that extends all functionality of ASTVisitor as well as their own added specifications. This was the process followed in this project to create a 'CustomVisitor' class.

For any visitor to obtain access to an AST, the linked compilation unit produced from parsing must 'accept' a given visitor object. Only then can the visitor traverse through the AST and collect information on the node types specified in each 'visit' method. In this case, the crucial node type is 'CastExpression'.

3.2.4 Cast Expression

To provide users with a thorough analysis of each system, the tool must do more than simply provide the total number of cast expressions present. Clearly, this was lacking adequate information to support users in coming to any conclusion about the general design of the system and its overall approach to the use of casting.

Further development gave the tool the following functionality:

- Visit the node in which the cast operator was type-changing through the use of a 'getChildren' method. The development of this method was very beneficial as it could return the children of any ASTNode, not just cast expression types. It was also used to determine the presence of message chains within expressions. Figure 16 shows this method. By implementing the StructuralProperty class, the tool was able to check the properties of a specific ASTNode and return the elements that were of type ASTNode themselves.

```
public static List<ASTNode> getChildren(ASTNode node) {  
    List<ASTNode> children = new ArrayList<ASTNode>();  
    if (node != null) {  
        List propertyList = node.structuralPropertiesForType();  
        for (int i = 0; i < propertyList.size(); i++) {  
            Object child = node.getStructuralProperty((StructuralPropertyDescriptor) propertyList.get(i));  
            if (child instanceof ASTNode) {  
                children.add((ASTNode) child);  
            }  
        }  
    }  
    return children;  
}
```

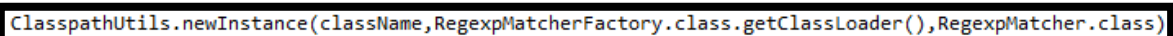
Figure 16 - getChildren Method

- Resolve the original binding of the object that is changing type. This is so that the tool can compare the initial data-type of the object to the data-type it is being converted to.
- Discover the relationship between the two data-types involved in each cast.
- Lend the user with all unknown bindings for manual inspection to allow for a greater understanding of the system that the tool cannot provide.

3.2.5 Other ASTNode Types

To determine whether or not the system was casting objects from one instantiated class to another, all 'TypeDeclaration' nodes were identified using the 'preVisit' method of ASTVisitor. This method visits all ASTNodes of the AST before type-specific nodes are found. Each type declaration node was temporarily recorded so that a list of system classes could be obtained and compared to expression bindings during analysis. The capability to recognise 'instanceof' operators was also included so to compare the total number of cases with the total number of casts to establish any underlying relationships or trends between the two. For example, if the same file has the most uses of casting **and** 'instanceof' operators or if a super class has the most uses of casting and a subclass has most uses of 'instanceof'. Again, to assist in drafting a more thorough conclusion of the system.

During early simulations, it was clear to see that the cast operator was being used in a huge variety of ways that were not anticipated. There was a recurring error being thrown when attempting to resolve bindings of expressions that were not simply a single variable. In particular, when the cast function was followed by a method or string of method invocations known as message chains, discussed earlier. For example, Figure 17 illustrates an expression that is involved in casting that the tool was unable to resolve the binding for.



```
ClasspathUtils.newInstance(className,RegexMatcherFactory.class.getClassLoader(),RegexMatcher.class)
```

Figure 17 - Unknown expression involved in casting

Therefore, it was necessary to implement code that recognised each method invocation and attempt to gain knowledge of its return type which would be the type that the object is being cast from. Again, due to time constraints, rules for all outcomes could not be enforced. However, with the aid of output to the console, manual inspection was able to identify the majority of the unknown cases. Although considered a completely different smell type, the use of message chains was recorded to find out what percentage of unknown expression types were a result of their use. This is done by using the previously developed 'getChildren' method again and if there are more than two returned (the expression is made up of 3 or more method calls), the expression is considered a message chain.

To conclude each analysis, the following results will be displayed to the console:

- Total number of Java files analysed
- The average number of casts per Java file
- Total number of cast operators identified in the system
- Total number of 'instanceof' operators
- The file that has the most cast instances
- The file that has the most 'instanceof' instances
- Total casts that involve internal system classes
- Total casts that were from Object class to a more specific type
- Total casts that only involve primitives
- Other return types in the system and option to print them all
- Total Unknown cast expression types
- Number of unknown values caused by message chains
- Option to print all unknown expression types for manual inspection

This chapter has discussed the software used and important aspects of the code that one may need if the project is to be reproduced or further developed in the future. The benefits of using Eclipse JDT have been highlighted and the depths of research that can be reached with the package. In this project, it cannot be stressed enough how important it was in the development of the cast analyser tool. After a number of considerations, the tool was finalised with its current functionality. This was to allow for adequate analysis of as many real-life systems from the Qualitas Corpus as possible.

The following chapter will contain the results obtained from the analysis of each system. The official output of the tool will be included, evaluated and discussed. To ensure the integrity of each output, each system will undergo manual inspection even if all cast instances are accounted for. All findings will also be deliberated to give a fair, unbiased review of the analysis tool.

4.0 Analysis

This chapter details all results obtained from the analysis tool and other observations as a result of manual inspection. A brief description of each system and its general use in real-world environments will be included to provide a background of the program. This will then be followed by the analysis part. Each section will first present the output values as a result of the system being passed through the tool. Secondly, an in-depth discussion of the values will be included followed by the findings discovered through manual inspection. The aim of the discussions will be to try and identify why the cast operator has been used and the general reason for its necessity. Particularly, the analysis will investigate the most cast dense files and if their use is a result of poor overall design. Additionally, the same process will be followed if there is an abundance of the type-check operator 'instanceof'. Values that the tool provide to solely aid manual inspection will not be included and will only be discussed along with overall conclusions of each analysis. For example, methods, method chains and variables in which the binding could not be resolved due to them returning null.

As previously mentioned, the implementation of the final tool was an iterative process to optimise the usefulness of the overall report. Therefore, so too are the results. For example, there were systems analysed that did not, in fact, have any files ending in '.java' and were therefore left out of this chapter. The reason being that manual inspection alone of the whole system would be far too time consuming and difficult to compare with other analysis outcomes from the tool. Similarly, systems that had very little instances of casting were left out as they added very little value to the investigation.

4.1 Apache-Ant 1.8.4

Apache-Ant is a Java library that is used to efficiently build Java systems as targets and extension points. It is commonly used to build Java applications as there are many advantages in doing so. For example, it can automatically remove '.ignore' files and other local temporary directories. The open-source software is also very powerful in compiling '.java' files but is not limited to only compilation tasks but also testing and other aspects of development. It is an extremely popular tool used by Java developers. Although used to enhance the capabilities of the Java language, it is also written in Java which is why it is suitable for analysis by the tool.

4.1.1 Results

	Value
Total .java files analysed	1196
Total number of cast instances	2590
Average casts per file	2.166
Total 'instanceof' instances	365
Total casts that involve a system Class	140
Total casts that involve the Object Class	1093
Total casts from that involve primitives	348
Total errors that occur	365
Total errors due to message chains	51

Table 3 - Apache-Ant Analysis Results

4.1.2 Most Cast Dense File – 'ZipEncodingTest.java'

'ZipEncodingTest.java' has the most instances of casting out of the 1196 '.java' files that were analysed in the system. Although the tool returned a total of 124 instances of casting, upon manual inspection, it is clear there are considerably more. The cast operator is used within a conventional for loop as shown in Figure 18.

```
for (int i = 0; i < 256; ++i) {  
    testBytes[i] = (byte) i;  
}
```

Figure 18 - Cast Inside for Loop

As a result, there are an additional 256 instances of casting not recognised by the tool. This has been noted for future work and similar implementations will be checked during the manual inspection. Initially, this seems rather excessive. However, the file is actually using the cast operator to convert data types to '(byte)' in order to carry out operations needed for bit-packing. A small section of code is shown in Figure 19 to demonstrate this approach.

```
(byte) 0x80, (byte) 0x82, (byte) 0x83, (byte) 0x84  
(byte) 0x85, (byte) 0x86, (byte) 0x87, (byte) 0x88  
(byte) 0x89, (byte) 0x8A, (byte) 0x8B, (byte) 0x8C  
(byte) 0x8E, (byte) 0x91, (byte) 0x92, (byte) 0x93  
(byte) 0x94, (byte) 0x95, (byte) 0x96, (byte) 0x97
```

Figure 19 - Bit-Packing example

Bit-packing is the process of inserting non-byte size data into primitive data types. There is no serious cause for concern as bit-packing is used to increase the efficiency of output during testing. Unlike the normal use of casting which is to access the functionality of a class.

4.1.3 Most 'Instanceof' Dense File - 'UnknownElements.java'

'UnknownElements.java' has the most cases of the 'instanceof' type-check operator. It has a total of 14 instances out of the total 365 for the system. It is clear from the name of the file that it has been set up to solely check the data-types of various return values. All but two cases of the operator check for the 'Task' datatype and then follow up with a cast if true is returned.

```
if (o instanceof Task) {  
    Task task = (Task) o;  
    task.setTaskType(ue.getTaskType());  
    task.setTaskName(ue.getTaskName());  
    task.init();  
}
```

Figure 20 - 'UnknownElements.java' instanceof example

An example is illustrated in Figure 20 in which the initial binding of 'o' is of type Object. The file comments describe it as a wrapper class that creates Tasks and data types that are otherwise unknown during runtime. The file appears to be unrelated to 'ZipEncodingTest.java' which had the most instances of casting. The sole purpose of this class is to take in an object and return the data type of that object that is otherwise unknown.

4.1.4 Summary

The overall system has a total of 2590 instances of casting giving an average of 2.166 casts per file. There is a total of 1093 casts originating from Object data-types to more specific project classes. 348 of the casts are changing from primitive types but it is now clear that this can be considerably more due to the flaw in the tool. Most of the cases observed are narrowing values from 'int' to 'byte' during the bit-packaging process discussed. However, casting from the data type 'DirectoryScanner' to 'ArchiveScanner' was also a regular occurrence with a total of 47 instances. 'ArchiveScanner' is deemed a necessary subclass of 'DirectoryScanner' to add archive specific functionality. This is exactly why the cast function has been used as displayed in Figure 21.

```
ArchiveScanner as = (ArchiveScanner) getDirectoryScanner(getProject());  
return as.getResourceFiles(getProject());
```

Figure 21 - example of casting between project classes

Furthermore, the use of casting to convert from a 'URLConnection' type is also predominant in the system. The cast operator is often used to convert the object to

type 'URLConnection', a subclass of 'URLConnection' that has extra functionality for systems that are only dealing with HTTP or HTTPS. Web protocols that are used for secure communications over a computer network. Other than this, there aren't many project class conversion that are substantial enough to suggest the need for any form of refactoring. Out with these three parameters, casting is also regularly used to convert 'Object[]' arrays. There were 365 errors that the tool could not distinguish bindings for. Although 51 of these were due to message chains, the majority of them were caused by the same expression 'getCheckedRef().touch(modTime)'. The 'touch()' method is used to update a modified file at a specific modification time. However, the tool is seeing it as an additional method call and trying to resolve the binding for that instead of the return type of 'getCheckedRef()'.

4.1.5 Conclusion

Apache-ant is a relatively large system that presents multiple arguments for the necessity of the cast operator. It will be interesting to see if other systems have similar results. For example, casting from Object data-types to specific project classes greatly outweighing all other conversion types. This could be a design that is very common in large Java systems which allows general objects to be passed around until a definite type is decided later on in the program. As all the systems have test classes, maybe this bit-packing will be a regular occurrence to increase the efficiency of output. With an average of 2.166 casts per file, Apache-Ant does not appear to require major refactoring due to design flaws.

4.2 ArgoUML

ArgoUML is a diagramming application that is written in java and is therefore accessible by any platform running on Java. Developers can simply download the zip file from '<http://argouml.tigris.org/>' and add the '.jar' to the classpath of their projects. ArgoUML provides developers with an efficient interface that allows them to create and modify numerous UML diagrams such as class and sequence diagrams. Additionally, there is an advanced code generation feature available with ArgoUML that can automatically generate classes, source code and interactions based on the diagram you have created. This feature is available for many languages including C++ and Java and is based on strict Java standards.

Not only does ArgoUML provide extra guidance for developing well designed UMLs, it also evaluates and suggests possible improvements that can be made to the overall design of the diagram using its 'design critics' feature.

4.2.1 Results

	<u>Value</u>
Total .java files analysed	1922
Total number of cast instances	8367
Average casts per file	4.353
Total 'instanceof' instances	3458
Total casts that involve project Class	1103
Total casts that involve Object Class	4794
Total casts that involve primitives	668
Total errors that occur	564
Total errors due to message chains	141

Table 4 - ArgoUML Analysis Results

4.2.2 Most Cast AND 'instanceof' Dense File – 'CoreHelperMDRImpl.java'

Unlike the system Apache-Ant, the greatest number of casting operators in ArgoUML is in the same file as that of the 'instanceof' operator. There are a whopping 447 cases of both casting and 'instanceof' in the file 'CoreHelperMDRImpl.java'. Upon manual inspection, almost every method employs at least one of the operators. However, there is a great variety of type checks that occur. For example, the program checks if an object is of type Class or Subclass and then later checks if an object is of type 'Namespace', as shown in Figure 22 and Figure 23 - ArgoUML type check example .

```
if (!(type instanceof Class) || !(subType instanceof Class)) {
    throw new IllegalArgumentException("Metatypes are expected");
}
```

Figure 23 - ArgoUML type check example 1

```
if (parent instanceof Namespace && isVisibleOwned(element, (Namespace) parent)) {
    return true;
}
```

Figure 22 - ArgoUML type check example 2

Similarly, with casting, the system casts to many different types such as List and 'Dependency' type as shown in Figure 24 and Figure 25.

```
if (handle instanceof Dependency && element instanceof ModelElement) {
    ((Dependency) handle).getSupplier().add((ModelElement) element);
    return;
}
```

Figure 25 - ArgoUML cast example 1

```
if (features instanceof List) {
    featuresList = (List) features;
} else {
    featuresList = new ArrayList(features);
}
```

Figure 24 - ArgoUML cast example 2

Out of all the different type checks, the one that occurs the most is 'instanceof Classifier' with a total of 93 occurrences. Classifiers are a type of element that is characterised in Unified Modelling Languages (UML) which have similarities whether it be attributes, structural features or behavioural features such as methods. Therefore, the result of the type check solely depends on the characteristics of the Classifier Class, set by the system itself.

Not only does 'CoreHelperMDRImpl.java' carry out a large number of type checks, the way that it has implemented it is not only hard to understand but hard to physically inspect due to its repetitive use of else if statements. Figure 26 shows a small section of code from the file.

```
} else if (handle instanceof State
    && container instanceof StateMachine) {
    ((State) handle).setStateMachine((StateMachine) container);
} else if (handle instanceof Transition
    && container instanceof StateMachine) {
    ((Transition) handle).setStateMachine((StateMachine) container);
} else if (handle instanceof Action
    && container instanceof Transition) {
    ((Transition) container).setEffect((Action) handle);
} else if (handle instanceof Guard && container instanceof Transition) {
    ((Guard) handle).setTransition((Transition) container);
} else if (handle instanceof ModelElement
    && container instanceof Namespace) {
    ((ModelElement) handle).setNamespace((Namespace) container);
} else {
    throw new IllegalArgumentException("handle: " + handle
        + " or container: " + container);
}
```

Figure 26 - Poor code design in ArgoUML

ArgoUML should consider refactoring in some way to improve the quality and maintainability of the code. Possible refactoring examples could be to use the Enum data type or switch statements that do not alter the logic of code but can simply improve the readability. Figure 27 shows another example of how 'instanceof' is implemented in the system.

```

else {
    if (!(modelElement instanceof UmlPackage
        || modelElement instanceof Classifier
        || modelElement instanceof UmlAssociation
        || modelElement instanceof Generalization
        || modelElement instanceof Dependency
        /* The next 2 needed for issue 2148: */
        || modelElement instanceof Extend
        || modelElement instanceof Include
        || modelElement instanceof Constraint
        || modelElement instanceof Collaboration
        || modelElement instanceof StateMachine
        || modelElement instanceof Stereotype)) {
        return false;
    }
}

```

Figure 27 - ArgoUML implementation on 'instanceof'

From the 447 instances of casting, the most use cases of the operator are to convert objects to the type 'ModelElement' with a total of 63 instances. The majority of use cases first check the type to confirm the object is indeed of type 'ModelElement'. However, there are a handful of circumstances that do not use any type check methods but still explicitly cast to object, illustrated in Figure 28.

```

public Collection<Permission> getPackageImports(Object client) {
    if (!(client instanceof Namespace)) {
        throw new IllegalArgumentException("invalid argument");
    }
    List<Permission> result = new ArrayList<Permission>();
    try {
        for (Dependency dependency : ((ModelElement) client).getClientDependency()) {

```

Figure 28 - ArgoUML cast without type check

4.2.3 Summary

The overall system has a total of 8367 instances of casting which would suggest the developers for ArgoUML view casting in a different light than that of Apache-Ant. Is it possible that they do not see casting as a code smell at all which is why they have used the operator so frequently? The system has an average of 4.353 casts per file, more than double than that of Apache-Ant. Again, the greatest number of objects being cast are of Object data type with 4794 occurrences. Most of which were carrying out type checks beforehand.

```
public boolean isAbstract(Object handle) {
    try {
        if (handle instanceof Operation) {
            return ((Operation) handle).isAbstract();
        }
        if (handle instanceof GeneralizableElement) {
            return ((GeneralizableElement) handle).isAbstract();
        }
    }
}
```

Figure 29 - ArgoUML type checking from Object type

Like the example shown in Figure 29, ArgoUML constantly passed in parameters of Object data type, carried out a type check and then cast to that specific type. Similar examples show that ArgoUML had the approach of keeping as many objects of type Object as they could. The system then uses custom files that have the responsibility to check general Object type variables and then cast them to specific classes. 'FacadeMDRImpl.java' is an example of such files, Figure 30 shows one of its methods. However, there are many more that are near identical that check for a huge variety of data types.

```
public Object getReceiver(Object handle) {
    try {
        if (handle instanceof Stimulus) {
            return ((Stimulus) handle).getReceiver();
        }
        if (handle instanceof Message) {
            return ((Message) handle).getReceiver();
        }
    } catch (InvalidObjectException e) {
        throw new InvalidElementException(e);
    }
    return illegalArgumentObject(handle);
}
```

Figure 30 - Example of a type check method in ArgoUML

There were a total of 668 instances of casting that involved primitive data types. Upon further inspection, the system interestingly uses the explicit cast function to store integer values as their corresponding ASCII (American Standard Code for Information Interchange) character. A method that employs the casting operator in this way is shown in Figure 31.

```
private boolean isLastTag(int ch) {
    if (ch == '<') {
        inTag = true;
        currentTag.setLength(0);
    } else if (ch == '>') {
        inTag = false;
        String tag = currentTag.toString();
        if (tag.equals(endTagName)
            // TODO: The below is not strictly correct, but should
            // cover the case we deal with. Using a real XML parser
            // would be better.
            // Look for XML document has just a single root element
            || (currentTag.charAt(currentTag.length() - 1) == '/'
                && tag.startsWith(tagName)
                && tag.indexOf(' ') == tagName.indexOf(' '))) {
            return true;
        }
    } else if (inTag) {
        currentTag.append((char) ch);
    }
    return false;
}
```

Figure 31 - ArgoUML casting example

As ArgoUML is such a vast system, other files that were manually inspected were those that had the most instances of casting after 'CoreHelperMDRImpl.java'. This was to identify how the system used the casting operator to convert custom data types, which occurred 1103 times. Other than casting to 'ModelElement' previously discussed, the system often casts to 'JPanel'. In Java, the JPanel class is used to store components and can be customised to provide various layouts depending on the organisation required. JPanel is part of the Swing package that is used to enable the development of graphical user interfaces (GUI). Figure 32 shows how the casting operator is used in the system and Figure 33 shows an example of the GUI's possible by using JPanel.

```
Object o = it.next();
if (o instanceof TabToDoTarget) {
    tab = (JPanel) o;
    break;
}
```

Figure 32 - ArgoUML casting to type 'JPanel'

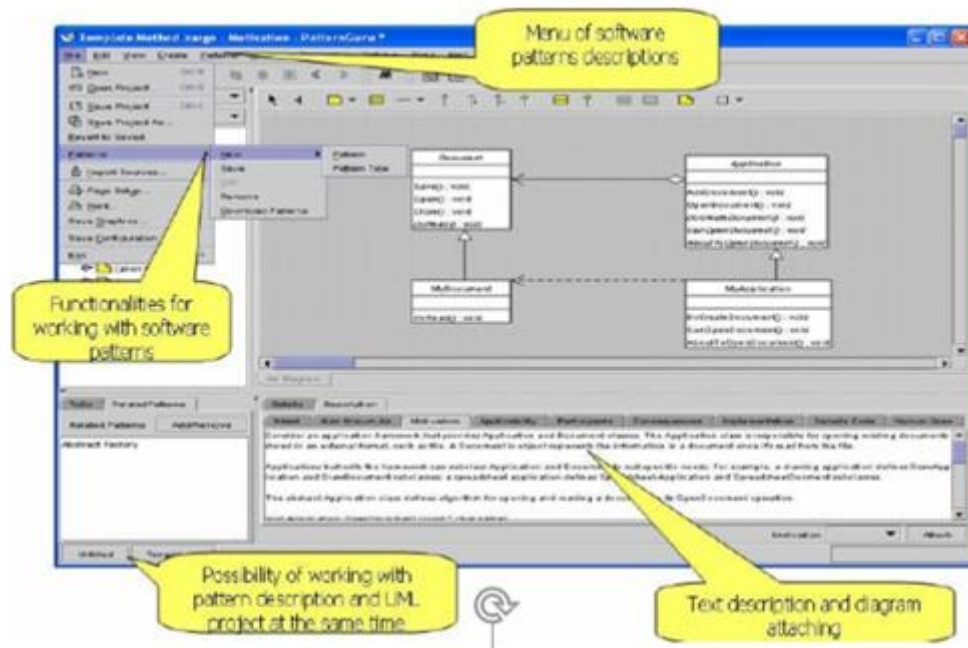


Figure 33 - GUI options available through Swings JPanel class (Boskovic, 2005)

There were 570 expressions that could not be fully analysed, this is not as bad as what was first expected as it is only 198 more than Apache-Ant despite having nearly double the number of analysed Java files.

143 of these errors were caused by message chains that the tool has difficulty dealing with. Although not particularly high compared to the size of the system, it is still a substantial amount that could be inspected for possible refactoring.

With the aid of the analysis tool providing a list of all unknown bindings for manual inspection, an observation was made across the whole system. Often, the return value of the method 'getPanelTarget()' was involved in many cast instances. The returned object is of type 'Fig' but is often declared as type Object, supporting the point made earlier. Figure 34 shows an example of this.

```
Object target = getPanelTarget();
// TODO: Why is this code even getting called for a FigGeneralization?
if (target instanceof PathContainer) {
    PathContainer pc = (PathContainer) getPanelTarget();
    pathCheckBox.setSelected(pc.isPathVisible());
}
```

Figure 34 - ArgoUML declaring variables as type Object

Although Figure 34 shows the object being cast to 'PathContainer', this only occurs a few times. Most cases by far are cast to one of the following data types:

- FigCompartmentBox
- FigInterface
- StereoTypeContainer
- Visibilitycontainer
- FigText
- FigAssociationClass
- PathContainer
- FigRRect

Additionally, all of the casts that include this 'getPanelTarget()' method are in a specific type of file.

- StylePanelFigClass.java
- StylePanelFigAssociation.java
- StylePanelFigInterface.java
- StylePanelFigText.java
- StylePanelFigPackage.java
- StylePanelFigRRect.java
- StylePanelFigNodeModelElement.java

All of which extends the 'StylePanelFig' class. Following some background research, all of these files involved in giving the user the ability to adjust the common attributes of a Fig. A Fig class is simply used instead of the console for input and output, this can simply be a pop-up box that displays a message in the form of a String.

4.2.4 Conclusion

As a much bigger system, ArgoUML was expected to use the casting operator more than Apache-Ant but not quite double the average casts per file. It was observed that ArgoUML employs a technique that uses Object data types more frequently across the entire system. Resulting in many dynamic type checks required within the system itself. Although this approach can be practical, it forces the use of casting, which can be considered unnecessary if the correct parameter is originally passed in. It can also cause confusion during maintainability as it can be difficult to understand what the method does from its signature as well as limiting the possibilities of overloading. Users and developers may assume that any object type can be passed in and the method will work, which is usually not the case. From the findings of the tool and manual inspection, a number of refactoring techniques discussed could increase the overall code quality of ArgoUML.

4.3 JHotDraw 7.5.1

JHotDraw can be considered a similar system to ArgoUML in the sense that it is used to aid development and annotation of GUI based applications and tools. However, JHotDraw is more specific to applications that intend to be used for drawing technical graphics such as network layouts. JHotDraw is the Java version of HotDraw and has impressive capabilities that can be extended to incorporate missing functionality that may be desirable for specific projects. The software was originally developed by the man who first defined what a 'code smell' was, Kent Beck, and was the first package labelled a 'framework' as it was specifically designed for reuse. JHotDraw version 7.5.1 was released in 2010 and is the most recent version within the Qualitas Corpus. It is written in Java and has had continuous updates throughout its lifetime. The system is suited for a study such as this and will be interesting to compare with ArgoUML to observe the differences or similarities in their approach to casting.

4.3.1 Results

	Value
Total .java files analysed	613
Total number of cast instances	2685
Average casts per file	4.380
Total 'instanceof' instances	362
Total casts that involve project Class	117
Total casts that involve Object Class	680
Total casts that involve primitives	705
Total errors that occur	495
Total errors due to message chains	130

Table 5 - JHotDraw analysis Results

4.3.2 Most Cast Dense File – 'Base64.java'

'Base64.java' uses the cast operator the most out of the whole system with a total of 81 instances. Upon manual inspection, the file uses the function to convert 'char' data types into the data type 'byte' while filling a corresponding byte[] array during declaration. The full declaration is illustrated in Figure 35.

```
private final static byte[] _NATIVE_ALPHABET = /* May be something funny like EBCDIC */ {  
    (byte) 'A', (byte) 'B', (byte) 'C', (byte) 'D', (byte) 'E', (byte) 'F', (byte) 'G',  
    (byte) 'H', (byte) 'I', (byte) 'J', (byte) 'K', (byte) 'L', (byte) 'M', (byte) 'N',  
    (byte) 'O', (byte) 'P', (byte) 'Q', (byte) 'R', (byte) 'S', (byte) 'T', (byte) 'U',  
    (byte) 'V', (byte) 'W', (byte) 'X', (byte) 'Y', (byte) 'Z',  
    (byte) 'a', (byte) 'b', (byte) 'c', (byte) 'd', (byte) 'e', (byte) 'f', (byte) 'g',  
    (byte) 'h', (byte) 'i', (byte) 'j', (byte) 'k', (byte) 'l', (byte) 'm', (byte) 'n',  
    (byte) 'o', (byte) 'p', (byte) 'q', (byte) 'r', (byte) 's', (byte) 't', (byte) 'u',  
    (byte) 'v', (byte) 'w', (byte) 'x', (byte) 'y', (byte) 'z',  
    (byte) '0', (byte) '1', (byte) '2', (byte) '3', (byte) '4', (byte) '5',  
    (byte) '6', (byte) '7', (byte) '8', (byte) '9', (byte) '+', (byte) '/'  
};
```

Figure 35 - JHotDraw converting char types to byte types

This employment of casting is used in the process of encoding and decoding from base64 notation. Base64 is a Java class used to deal with encryption (JavaTPoint, 2016). The file provides the necessary methods to both encrypt and decrypt the data within a system. A few methods used for encoding include:

- encodeBytes(byte[] source) – Used to encode an array of bytes into Base64 notation.
- encodeToFile(byte[] dataToEncode, String filename) – Used to encode data to a file.
- encodeFromFile(String filename) – Used to encode data in a file to Base64 notation.

The reason for JHotDraw using Base64 notation for characters is to maintain its ability to save, load, print and share figures and diagrams without being corrupted. Base64 notation ensures this. However, this comes at a cost as four bytes are produced from every three bytes of data, which can greatly increase the size of storage required if encoding a large data set.

4.3.3 Most 'Instanceof' Dense File - 'JavaPrimitivesDOMFactory.java'

'JavaPrimitivesDOMFactory.java' contains 39 use cases of the 'instanceof' operator. Although this is not many, the fact that it is more than half of the maximum number of casts in a file may be an issue. The reasoning being that the system may be using the casting function without carrying out necessary type checks first. The purpose of the class itself, 'JavaPrimitivesDOMFactory', is to serialise Java primitive objects and 'DomStoreable' objects. Upon manual inspection, the 'instanceof' operator is used to identify the type of an object being passed into a method and then return the type of the object as a String. This is a perfectly acceptable situation. However, the implementation of the method is similar to that of Figure 26, that there is a large chain of 'if' and 'else if' statements that make the body of the method. Figure 36 displays a section of one of these methods.

```
public String getName(Object o) {  
    if (o == null) {  
        return "null";  
    } else if (o instanceof Boolean) {  
        return "boolean";  
    } else if (o instanceof Byte) {  
        return "byte";  
    } else if (o instanceof Character) {  
        return "char";  
    } else if (o instanceof Short) {  
        return "short";  
    } else if (o instanceof Integer) {  
        return "int";  
    } else if (o instanceof Long) {  
        return "long";  
    } else if (o instanceof Float) {  
        return "float";  
    } else if (o instanceof Double) {  
        return "double";  
    } else if (o instanceof Color) {  
        return "color";  
    } else if (o instanceof Font) {  
        return "font";  
    } else if (o instanceof byte[]) {  
        return "byteArray";  
    }  
}
```

Figure 36 - JHotDraw 'else if' chain

Which again could possibly be replaced with a more manageable and aesthetically pleasing 'switch' statement. The same approach is used in other methods within the class. Additionally, the operator is used when the object passed in is an Array type. This is followed by a cast to a new array and then filled as illustrated in Figure 37.

```
else if (o instanceof double[]) {  
    double[] a = (double[]) o;  
    for (int i = 0; i < a.length; i++) {  
        out.openElement("double");  
        write(out, a[i]);  
        out.closeElement();  
    }  
}
```

Figure 37 - JHotDraw using both instanceof and cast operator

Other than these observations, the 'instanceof' operator is not used an excessive number of times and does not provide the user with a greater understanding of the system.

4.3.4 Summary

The final output values of the tool analysis have some surprising results. Despite the fact that there were only a total of 2685 instances of casting, the total number of '.java' files analysed was only 613. Which means the average number of casts per file is 4.380, higher than that of ArgoUML which had a total of 8367 instances of casting. Suggesting that the developers of JHotDraw do not, in fact, view casting as a code smell at all, which might actually be the case.

Once more, the results of JHotDraw are not what they seem. With many cast instances, one would expect that a similar approach to development would be adopted as ArgoUML. To keep an object of the general Object data type and then cast to specific types when required. However, JHotDraw is the first system that has more casts involving primitive types than that of Object data types.

There are a total of 705 casts that are either casting to or from primitives compared to a total of 680 casts of type Object. Other than the primitive conversions discussed earlier in Figure 35, the others are spread out fairly evenly. For example, if you follow the majority of the primitive casts to their original file. In most cases, there are only two casts in the file, others only have a single instance. Therefore, other than the conversions required for Base64 notation, there is no substantial data that can aid users in refactoring.

The program casts between system classes a total of 680 times. Although less than that of primitives, it is still a substantial amount with respect to the size of the whole system. In fact project class casts account for 25.3% of the total casts. Many of which occur in the file 'FontFamilyNode.java'. Referencing the JHotDraw API, this file implements the 'MutableTreeNode' and only allows 'FontFaceNode' as child nodes (JHotDraw7API, n.d.). MutableTreeNode is an interface that specifies the requirements for a tree node object that may be subject to change. FontFaceNode also implements MutableTreeNode and is a class that does not allow children. The reason for using these classes within JHotDraw may be to efficiently traverse through a Java tree but limit the number of children produced from each node. Figure 38 illustrates a method within 'FontFamilyNode' and how it uses the cast function to ensure that the child nodes are not split again into children of their own.

```
public void insert(MutableTreeNode newChild, int index) {
    FontFamilyNode oldParent = (FontFamilyNode) newChild.getParent();
    if (oldParent != null) {
        oldParent.remove(newChild);
    }
    newChild.setParent(this);
    children.add(index, (FontFaceNode) newChild);
}
```

Figure 38 - Project class casts in JHotDraw

Other casts that regularly occur in JHotDraw, is the cast from type Component. Component is a Java class in which its instances have a graphical representation that can be displayed and interacted with by the user. In a system like JHotDraw that is mainly used to create and customise graphical user interfaces for the user, there will be many different types of Component types such as scrollbars, checkboxes and other custom types set up by the system. As a result of manual inspection, it can be stated that JHotDraw uses the Component class much like other systems use Object.

For example, objects of type `Component` are passed into a method. Followed by a type check and a cast if the check returns true. This is a common theme throughout JHotDraw, an example is shown in Figure 39 from the file 'ReOpenApplicationAction.java'.

```
Component c = SwingUtilities.getRootPane(v.getComponent()).getParent()
if (c instanceof JFrame) {
    JFrame f = (JFrame) c;
    if ((f.getExtendedState() & JFrame.ICONIFIED) != 0) {
        f.setExtendedState(f.getExtendedState() ^ JFrame.ICONIFIED);
    }
    f.requestFocus();
}
```

Figure 39 - JHotDraw casting from type `Component`

Unfortunately, there were a total of 495 instances in which the binding of the type being converted could not be resolved. 130 of these were due to message chains. However, there were various other reasons that errors occurred such as the `Java Clone()` method that the analysis tool consistently struggles with. This will be discussed in the analysis of the tool itself. The specific method in JHotDraw that threw the tool off was 'createUI()'. Many different object types call upon this method such as:

- `PaletteButtonUI`
- `PaletteLabelUI`
- `PaletteSliderUI`
- `PaletteFormattedTextFieldUI`

All of which extend the corresponding `BasicUI` class and have been customised to enable palette specific functionality. JHotDraw uses the cast function along with the method call, inside a set method. This is so that the property being set is of the correct type. An example of '`PaletteButtonUI.createUI()`' does exactly this in Figure 40.

```
btn.setUI((PaletteButtonUI) PaletteButtonUI.createUI(btn));
```

Figure 40 - JHotDraw casting within a set method

4.3.5 Conclusion

JHotDraw is a considerably smaller system than that previously analysed. However, the results of both output from the tool and manual inspection drew out some interesting aspects of the system. The casting operator was frequently used throughout the system for various reasons. This particular analysis disclosed the need for casting when using Base64 notation, which was not considered at the beginning of this investigation.

This and many other cast dense files greatly increased the average of the system to 4.38 casts per file. Which can be expected if Kent Beck does not personally consider casting to be a code smell. The system has an efficient approach to creating different characteristics of user interfaces by using objects of type Component. However, it forces the use of the cast operator which may be unnecessary if an alternative method was used. Additionally, the use of the 'instanceof' operator in Figure 36 implies that there may be a need for other refactorings in the system that cast analysis alone wouldn't uncover.

4.4 Azureus (Vuze)

The Vuze software, previously known as Azureus, is a Java-based package that is used to share and transfer files via the BitTorrent protocol. BitTorrent protocol is a peer-to-peer method of transferring data over the internet. The most recent version included in the Qualitas Corpus is still Azureus but has since been renamed to Vuze in more recent versions. It is also a free open-source package but has many features similar to premium clients, which is why it is so popular among both aspiring and advanced developers. Vuze is compatible with many IDE's and has its own user interface functionality that can be accessed through the Eclipse Standard Widget Toolkit (SWT).

4.4.1 Results

	<u>Value</u>
Total .java files analysed	3319
Total number of cast instances	12915
Average casts per file	3.89
Total 'instanceof' instances	2130
Total casts that involve project Class	1688
Total casts that involve Object Class	5880
Total casts that involve primitives	3051
Total errors that occur	1836
Total errors due to message chains	110

Table 6 - Azureus analysis Results

'MD2Digest.java'

The cryptographic hash function employs a public key infrastructure that is used to produce hash values of text. The file 'MD2Digest.java' is used within Azureus to implement MD2 hash function.

```
private static final byte[] S = {
    (byte)61, (byte)54, (byte)84, (byte)161, (byte)236, (byte)240,
    (byte)6, (byte)19, (byte)98, (byte)167, (byte)5, (byte)243, (byte)192,
    (byte)199, (byte)115, (byte)140, (byte)152, (byte)147, (byte)43, (byte)217,
    (byte)188, (byte)76, (byte)130, (byte)202, (byte)30, (byte)155, (byte)87,
    (byte)60, (byte)253, (byte)212, (byte)224, (byte)22, (byte)103, (byte)66,
    (byte)111, (byte)24, (byte)138, (byte)23, (byte)229, (byte)18, (byte)190,
```

There is not much information disclosed about this file and its purpose within Azureus. Therefore, another cast dense file will also be included in this section to develop a deeper understanding of Azureus as a whole.

The second file behind 'MD2Digest.java' with the most cast instances is 'DesParameters.java'. However, this file is from the Bouncy Castle Crypto package and is also used to implement cryptographic algorithms. In fact, DES in the file name represents Data Encryption Standard. As a result, the casting operator is used in a similar fashion to that of 'MD2Digest.java'.

```
/* semi-weak keys */
(byte)0x01, (byte)0xfe, (byte)0x01, (byte)0xfe, (byte)0x01, (byte)0xfe, (byte)0x01, (byte)0xfe,
(byte)0x1f, (byte)0xe0, (byte)0x1f, (byte)0xe0, (byte)0x0e, (byte)0xf1, (byte)0x0e, (byte)0xf1,
(byte)0x01, (byte)0xe0, (byte)0x01, (byte)0xe0, (byte)0x01, (byte)0xf1, (byte)0x01, (byte)0xf1,
(byte)0x1f, (byte)0xfe, (byte)0x1f, (byte)0xfe, (byte)0x0e, (byte)0xfe, (byte)0x0e, (byte)0xfe,
(byte)0x01, (byte)0x1f, (byte)0x01, (byte)0x1f, (byte)0x01, (byte)0x0e, (byte)0x01, (byte)0x0e,
(byte)0xe0, (byte)0xfe, (byte)0xe0, (byte)0xfe, (byte)0xf1, (byte)0xfe, (byte)0xf1, (byte)0xfe,
(byte)0xfe, (byte)0x01, (byte)0xfe, (byte)0x01, (byte)0xfe, (byte)0x01, (byte)0xfe, (byte)0x01,
(byte)0xe0, (byte)0x1f, (byte)0xe0, (byte)0x1f, (byte)0xf1, (byte)0x0e, (byte)0xf1, (byte)0x0e,
(byte)0xe0, (byte)0x01, (byte)0xe0, (byte)0x01, (byte)0xf1, (byte)0x01, (byte)0xf1, (byte)0x01,
(byte)0xfe, (byte)0x1f, (byte)0xfe, (byte)0x1f, (byte)0xfe, (byte)0x0e, (byte)0xfe, (byte)0x0e,
(byte)0x1f, (byte)0x01, (byte)0x1f, (byte)0x01, (byte)0x0e, (byte)0x01, (byte)0x0e, (byte)0x01,
(byte)0xfe, (byte)0xe0, (byte)0xfe, (byte)0xe0, (byte)0xfe, (byte)0xf1, (byte)0xfe, (byte)0xf1
```

54

Figure 42 shows how the file casts to type byte when developing data encryption keys, these are required when a block cypher is used to encrypt data.

Giving the nature of Azureus and what it is used for, it is no surprise that there are so many files and casting instances that are involved in the encryption of data.

As Azureus allows for peer-to-peer sharing of data, it can be expected that it is difficult to find sources explaining the function of each file in the system. The tool does, however, highlight how the necessity of the cast operator to carry out such encryptions.

4.4.3 Most 'instanceof' Dense File – 'BEncoding.java'

Bencoding is the type of encoding used in the Azureus system and in fact most BitTorrent clients. It is also involved in the process of peer-to-peer file sharing. However, it specifically supports byte Strings, integers, lists and dictionary data types. 'BEncoding.java' uses the 'instanceof' operator to check the type of object before casting and then encoding it. The type checks range from type Map and TreeMap to String. An example is shown in Figure 43.

```
else if(o_key instanceof String)
{
    String key = (String) o_key;
    if (byte_keys)
    {
        try
        {
            encodeObject(Constants.BYTE_CHARSET.encode(key));
        }
        catch (Exception e)
        {
            // ignore
        }
    }
}
```

Figure 43 - Azureus type check before encoding

The file only has a total of 47 instances of the operator and follows a similar approach to that of Figure 43 in nearly all of the cases.

4.4.4 Summary

Being the largest system so far, the tool analysed a total of 3319 '.java' files. Although there is a total of 12915 instances of the cast operator, Azureus has an average of 3.89 casts per file. Lower than that of JHotDraw which is over five times smaller than Azureus. As Azureus is such a vast system, it would be impossible to manually inspect all popular conversion types. Therefore, a number of reoccurring cases will be inspected. However, the main figures to observe during this analysis is the average casts per file. The usefulness of these values will be discussed in the following chapter.

The system has 5880 instances of casting that converts from Object data types. Similar approaches to previous systems can be observed in Azureus, there is an abundance of methods that accept Object type parameters, check the type and then use casting to access class-specific methods. Figure 44 displays two methods that have implemented this approach.

```
// @see java.lang.Object#equals(java.lang.Object)
public boolean equals(Object obj) {
    if ((obj instanceof VuzeActivitiesEntry) && id != null) {
        return id.equals(((VuzeActivitiesEntry) obj).id);
    }
    return super.equals(obj);
}

// @see java.lang.Comparable#compareTo(java.lang.Object)
public int compareTo(Object obj) {
    if (obj instanceof VuzeActivitiesEntry) {
        VuzeActivitiesEntry otherEntry = (VuzeActivitiesEntry) obj;

        long x = (timestamp - otherEntry.timestamp);
        return x == 0 ? 0 : x > 0 ? 1 : -1;
    }
    // we are bigger
    return 1;
}
```

Figure 44 – Azureus casting from Object data types

Additionally, the system has casting when using the ‘toArray’ function to instantiate an array of a specific type. This is also a recurring theme within Azureus, an example is displayed in Figure 45.

```
if( !sending_msgs.isEmpty() ) {
    messages = (ClientMessage[])sending_msgs.toArray( new ClientMessage[sending_msgs.size()] );
}
```

Figure 45 – Azureus using casting during array instantiation

There were a total of 3051 casts that convert objects of primitive data types. As well as ‘DESParameters.java’ that uses casting in the process of data encryption, there are many other files that have similar if not the same reasons for using the cast operator. Encryption of data is a crucial aspect of a system that promises peer-to-peer sharing. However, the way it uses casting seems unnecessary and could certainly be reviewed for refactoring.

As a result of further investigating other ways Azureus uses casting, it was discovered that there are many instances in which the cast operator is used within various loops.

For example, Figure 46 shows the casting operator being used as many times as there are elements in the list 'managers'. However, like before, the tool only recognises one instance of casting.

```
for (int i=0;i<managers.size();i++){  
    DownloadManager manager = (DownloadManager)managers.get(i);
```

Figure 46 - Azureus using casting inside for loop

When inspecting the cast instances that the analysis tool could not resolve binding for, it was noticed that there was one conversion that occurred hundreds of times. 'cell.getDataSource()' occurs in multiple files such as :

- SizeItem.java
- StorageTypeItem.java
- TorrentRelativePathItem.java
- CategoryItem.java

All of which implement the interface 'TableCell'. However, having inspected the interface, the return type of '.getDataSource()' is simply an Object data type. It is not confirmed the reason the tool cannot resolve the binding for this method. Further analysis of the tool is required to pinpoint where it goes wrong in this particular example.

4.4.5 Conclusion

As previously mentioned, without the aid of an analysis tool to point users in the right direction, it would be near impossible to manually inspect the whole of Azureus for possible refactoring opportunities. The usefulness of the average value included in the analysis tool was highlighted as one may assume the more instances of casting, the more refactoring is required. However, this is not the case as shown in this chapter. So far the smallest system has had the highest average of casts per file. That said, the tool did recognise more than 12000 instances of casting and the way Azureus employs the operator as explained. Although this has been the largest system analysed by the tool so far, this is still considerably high in terms of casting being a code smell.

4.5 Marauroa

Due to the sheer size of Azureus, Marauroa was selected for analysis as it has more than half the '.java' files, allowing the project to compare systems with a wide range of sizes. Marauroa is a Java package that allows users to build their own online games by assisting with database persistence, object management and client-server communication. Marauroa is popular among developers that aim to produce 'old-school' games. Arguably, the most popular game to be produced using the package is 'Stendhal', an open-world adventure game where online players can interact and trade in-game currencies. Due to the consistent demand for these kinds of games, Marauroa has experienced continued development and updates. It will be interesting to see how this system compares to the others as it is the smallest system that is to be analysed.

4.5.1 Results

	<u>Value</u>
Total .java files analysed	207
Total number of cast instances	222
Average casts per file	1.07
Total 'instanceof' instances	18
Total casts that involve project Class	17
Total casts that involve Object Class	35
Total casts that involve primitives	76
Total errors that occur	44
Total errors due to message chains	0

Table 7 - Marauroa Analysis Results

Immediately, it is not difficult to notice the clear contrast between Marauroa and all of the other systems. Not only in terms of size but all results obtained from the analysis tool. That said, the analysis methods will remain consistent to highlight any surprising approaches Marauroa may have to the use of casting.

4.5.2 Most Cast Dense File – 'ClientFramework.java'

Within Marauroa, the file that uses the cast operator the most is 'ClientFramework.java', an abstract class that has a total of 19 casting instances. Upon inspection, there is no obvious approach to the use of casting within the file. For example, there are no large methods that contain multiple 'else-if' statements found during analysis of other programs.

There are, however, many switch statements that all include at least one instance of casting, which do not actually implement any type checks beforehand. These kinds of assumptions are not advised. An example is displayed in Figure 47.

```
/* Server replied with ACK to login operation */
case S2C_LOGIN_ACK:
    logger.debug("Login correct");

    onPreviousLogins(((MessageS2CLoginACK) msg).getPreviousLogins());
```

Figure 47 - Marauroa using casting inside SWITCH statement

Other than this, the casting operator is used sparsely, is it possible that the developers had a clear opinion when it came to its use. Unlike many of the other systems. The file is advised to be extended into the user's game to wrap actions that the online client should do.

4.5.3 Most 'Instanceof' Dense File – 'RPObject.java'

This opinion may become clearer throughout this analysis as the file that uses the 'instanceof' operator the most is 'RPObject.java'. However, it does so only **twice**. Surprisingly, this file is of considerable size with a total of 1896 lines. The two occurrences do, however, use the same approach that this project has frequently come across. Passing an object into a method, checking the type and then casting to access class-specific methods. Which is what Figure 49 and Figure 48 display.

```
public boolean equals(Object obj) {
    if (this == obj){
        return true;
    }
    if (obj instanceof RPObject) {
        RPObject object = (RPObject) obj;
```

Figure 48 - Marauroa Type check 1

```
public boolean equals(Object anotherid) {
    if (anotherid != null && anotherid instanceof RPObject.ID) {
        ID otherId = (RPObject.ID) anotherid;
```

Figure 49 - Marauroa Type Check 2

This file details what the system views as an 'object'. It explains that "everything is an object", whether It be "physical or logical". This is followed up with a large list of parameters that their custom object should have.

This may explain why there are only two use cases of 'instanceof' and why the system has more castings that involve primitive types than Object data types.

4.5.4 Summary

As briefly mentioned, Marauroa is the smallest system that has been inspected by the analysis tool. With a total of 207 '.java' files and 222 casts in total, Marauroa was included to increase the variety of systems analysed and see how they compare to each other. With an average of 1.07 casts per file, the lowest of all the systems, Marauroa does not use the function as nonchalantly as other programs. The system has 17 instances of casting that convert between internal classes. A few of these instances do not check the type first and have been used on the basis that the developer 'just knows' that the object is on that type. Which in turn, relies on a successful connection to the internet. For example, the property 'netMan' is declared in the class but instantiated in a method called 'connect()', as shown in Figure 50 and Figure 51.

```
protected INetworkClientManagerInterface netMan;
```

Figure 50 - Marauroa declaring 'netMan'

```
public void connect(String host, int port) throws IOException {  
    netMan = new TCPNetworkClientManager(host, port);  
}
```

Figure 51 - Marauroa instantiating 'netMan'

This is then followed up by a method shown in Figure 52 that requires the cast function to be the same type it was instantiated as.

```
/* Check network for new messages. */  
messages.addAll(((TCPNetworkClientManager) netMan).getMessages());
```

Figure 52 - Marauroa casting without type check

This seems like an unnecessary instance of casting and there are similar occurrences throughout other files. It is unclear why the system has done this and there are little to no comments explaining its purpose. This may be taken further in the discussion section of this report to try to understand the root of the cause.

Similarly to JHotDraw, Marauroa has more castings that involve primitives than that of Object data types, 41 instances to be exact. Again, this should be more if the tool was able to recognise the use of casting inside a for loop as shown in Figure 53.

```
for (int i = 0; i < b1.length; i++) {  
    res[i] = (byte) (b1[i] ^ b2[i]);  
}
```

Figure 53 - Marauroa casting inside FOR loop

The Message class in java also occurs regularly during conversions. However, this links back to the 'ClientFramework.java' file that uses switch statements to determine the specific type of message.

There were 44 errors that the tool could not identify binding for. Interestingly, none of them were due to message chains. Suggesting that the developers of Marauroa are reluctant to use many code smells. Many of the errors occur as a result of the Java clone() method that was discussed in section 4.3. Others include 'object._tojava_(PythonWorld.Class)' and 'ser.readObject(new RPObject())'. The first is to support the importing of Python scripts and the second is to create a new custom RPObject that was defined in 'RPObject.java' previously. Which again, is used inside a for loop shown in Figure 54.

```
for (int i = 0; i < added; ++i) {  
    RPObject object = (RPObject) ser.readObject(new RPObject());  
}
```

Figure 54 - Marauroa Cast that causes tool error

It is clear that an object of type 'RPObject' is returned but the tool throws an error when a 'new' object is instantiated and cast at the same time.

4.5.5 Conclusion

This system makes little use of both the casting operator and the 'instanceof' type check functionality. As well as it's custom object class 'RPObject', many of its classes inherit directly from type Object.

It is possible that the few '.java' files analysed contributes to the end results but the average casts per file is the lowest value obtained from all of the systems analysed. Marauroa further emphasised the improvements required in the analysis tool to be able to provide more accurate results.

4.6 Discussion

The focus point of this project as a whole was to attempt to provide an understanding of the different approaches to casting that systems may have and if this is a result of poor overall design. From the results and observations made, it was clear that each system viewed casting in a different light and required the operator for various purposes. There were no Java systems analysed, including those that were not included in this report for further inspection, which did not use a single instance of casting. Due to restrictions set by Java and its strict Object-Oriented approach, a fair point can be made that no matter how well a software systems design and architecture, it is very difficult if not impossible not to use casting at some point during development. However, as this project has found out, certain systems use casting far more incautiously than others. Table 8 illustrates the total files cast for each system alongside the average casts per file to allow a visual comparison of the two.

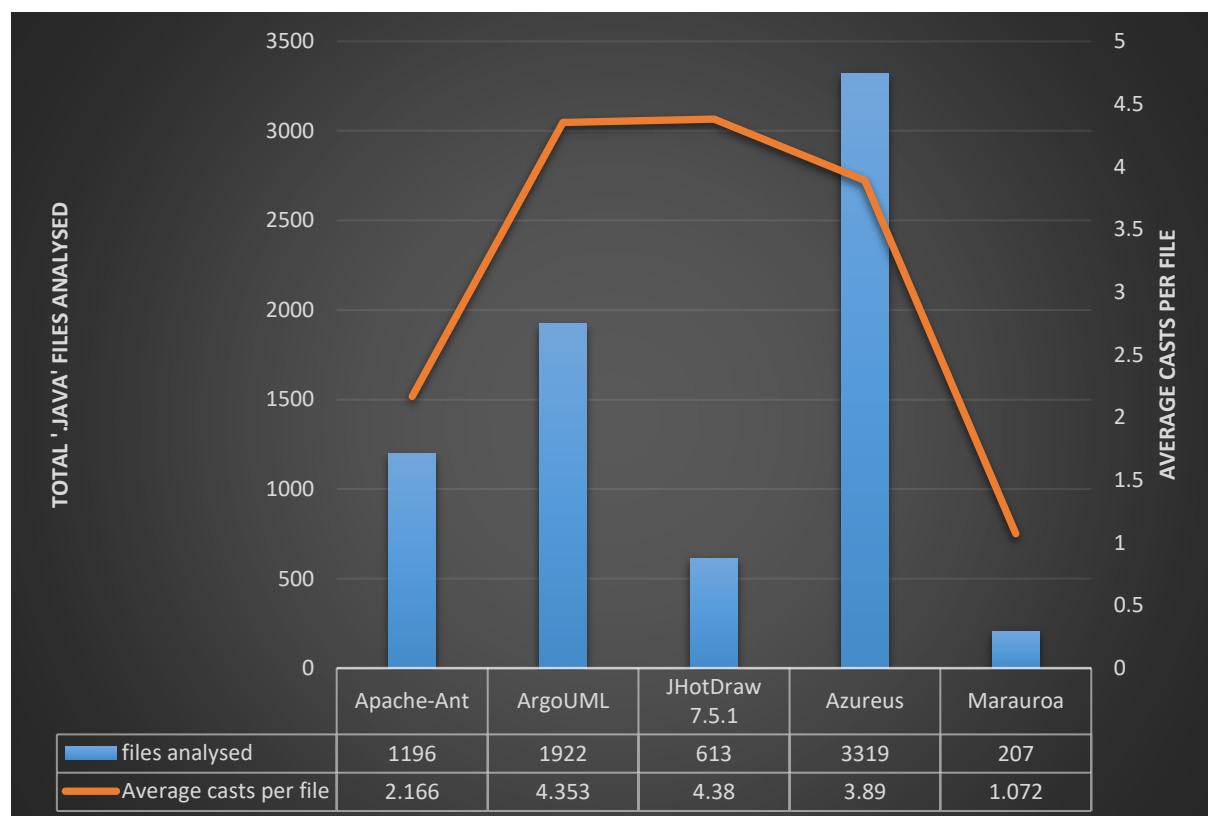


Table 8 - Total files analysed compared to average casts per file

4.6.1 Highest Cast Dense System

The system that used the casting operator the most relative to its size was JHotDraw. With an average value of 4.38 casts per file, JHotDraw surpassed systems that are more than five times its size. 613 '.java' files were analysed in which the tool identified 2685 instances of casting. This was mainly down to the implementation of encoding data, which required the system to cast 'char' data types to 'byte' data types.

4.6.2 Lowest Cast Dense System

The system that had the least amount of casting relative to its size was Marauroa. This was also the smallest system analysed and had an average of 1.07 casts per file. If additional systems were analysed in this report, it would be difficult to find a program that has an average as low as Marauroa. This may simply be down to its size or it could be possible that the developers had a negative opinion on the use of casting. So they designed the system in such a way that they would not require it. Regardless of the reason, casting analysis alone would not be able to provide adequate enough information on the system to suggest any possible refactoring.

4.6.3 Passing Object Type Parameters

There was a consistent need for casting throughout all the systems as a result of a specific design choice. Object data types being passed in as method parameters, the parameter would undergo multiple type checks, and then the object would be cast to the specific check that returned true.

However, on many occasions. The system checked for every possible return type. Resulting in an unsightly chain of 'else-if' statements. In examples like the method found in JHotDraw (Figure 36), refactoring ought to be considered. These particular instances of casting made up the majority of casts in all the systems that were analysed. Granted ArgoUML is a medium sized program with 1922 '.java' files, the system persistently utilised this design approach throughout many of its classes. The approach is considered bad practice within the software development community. Particularly, the point made during analysis that assumptions may be mistakenly made that any object type can be passed into these kinds of methods. Additionally, it forces developers to use casting throughout the system.

4.6.4 The Use of 'Instanceof' Operator

Due to the fact that most cast instances undergo a type check beforehand, it only made sense to correspondingly analyse the systems for the 'instanceof' operator. Which proved to contribute valuable data towards the project overall. It was definite that the use of the operator certainly came hand-in-hand with casting. The more the system used casting, the more type checks were required. Additionally, as the instances of casting that were converting Object data types increased, so too did the use of 'instanceof'.

4.7 Evaluation of Tool

The software analysis tool that was developed greatly increased the efficiency of the analysis process. By providing practical results, the tool proved extremely helpful during manual inspection and was used to focus on files and Java code that were of particular interest. It was clear early on in this project that the investigation would simply not be possible without the use of an automated inspection tool. Especially when dealing with systems as big as Azureus, within the time constraints of the project it would have been impossible to try and manually detect each instance of casting. Furthermore, the late addition to detect the 'instanceof' operator would not have been possible, which also contributed towards the overall investigation. All instances of casting that the tool identified were confirmed to be true positives. In terms of static analysis, there were no instances of casting that went unnoticed. As previously mentioned, the tool development was a continuous process. Once the fundamentals were established and it was capable of analysing real life systems, there were many improvements added when deemed necessary.

Other than the specific features detailed within this report, the tool also provides users with data types involved in casting that the tool was not implemented to handle. For example, if the tool was to provide details and resolve binding for every data type involved in every instance of casting, code would have to be developed to deal with every possible return type. This would simply be out of the scope of this project.

Therefore, instead of simply ignoring the fact that they exist, the tool has the option to provide all data types that were not accounted for so that the user can look for reoccurrences. This feature was incredibly helpful in allowing the user to get a quick breakdown of conversion types out with the final results of the tool.

Although there were many aspects of the tool that were a success, the tool did not perform as well as initially thought which has been highlighted throughout the analysis chapter. The first most predominant downfall is the inability to accurately count instances of casting inside a for loop. For example, a for loop may have iterated over ten objects, casting each of them to an alternate data type. However, the tool would only recognise a single instance of casting. To solve this problem, dynamic analysis would be required which is out with the scope of this investigation. This has caused an obvious decrease in accuracy of the final results. As discovered during analysis, the tool was also unable to resolve binding for many cast instances. One being for methods that created new objects during its implementation. This would be a main focus if there was additional time available.

5.0 Conclusions and Recommendations

This report documents the process of attempting to develop a static analysis tool that can aid users in recommending possible refactoring of Java code. The investigation supports Fowler and Beck by showing that human intuition cannot be beaten at the present moment when it comes to suggesting the best course of action to eliminate code smells (Fowler & Beck, 2000). However, static analysis tools such as the one developed here can provide an extremely helpful abstract of systems. By efficiently using the tool to quickly gather data, developers can then home in on areas of the program that they wish to further inspect or refactor based on personal experiences and opinions.

The report has outlined the various design aspects of numerous systems and identified that not all developers view the use of casting the same. There are various literature and teaching tools that will suggest reviewing a systems design instead of repeatedly having to cast objects. Along with many academics that have studied the topic of code smells. However, there are large systems like Azureus that use casting profusely and are still a great success and continue to be many years later. Therefore, is it possible that casting is a smell and nothing more? A mere suggestion that one could possibly redesign the system so that conversions are not required but at the same time, pose no threat to the functionality and efficiency of the final output. This may be why as mentioned in the literature review, many developers will disagree that casting should be viewed in a negative light.

5.1 Possible Future Work

There is a great scope to develop this project further. There are various different paths this investigation could go down that can provide a deeper understanding of not only casting, but code smells in general and how they exist in real life open source systems. Listed below, are numerous recommendations that have the potential to further increase the usefulness of the casting analysis tool.

- Integrate additional functions to allow for dynamic code analysis. By doing so, the tool could compare data types before and after the execution of the code as well as analysing the output data of each system. This may also enable the possibility of counting casting instances in real time, especially when the system is executing loops to repeatedly convert object data types.

- If time permits, develop code that is able to handle, if not all, more data types so that it can resolve binding for a higher percentage of expressions that are being converted. Although this could be a gruelling task, it would greatly increase the quality of the current tool. If a 100% success rate was achieved for resolving binding, the tool could then be developed to analyse the hierarchies of each data type and how the two types involved in the casting relate.
- As a follow up to the previous suggestion, the cast analysis tool developed in this project could collaborate with the tool discussed in section 2.4 that also carries out static analysis to provide information on hierarchy code smells within open source systems. If the two were able to integrate, a far more proficient tool could be produced that can first analyse the use of casting. Followed by an in depth analysis of the relationships of each data type.
- One could analyse more systems to try and identify other commonalities when it comes to casting. Moreover, there is also the option to expand not only the tool but the full investigation to other programming languages. Especially, the languages that implement explicit casting functions differently to convert data types such as C++. This allows for a comparison to be made not only between systems but also between high-level languages to observe for any commonalities.

5.2 Final Conclusion

The aim of this project was to investigate the use of casting in Java systems. It was first observed that the operator was one of few code smells that had not been studied to great depths. Some literature did not even include the function as a code smell at all. Therefore, a foundation had to be set for the project to build on which would be an unbiased view of the function. From there, the project would try to develop a deeper understanding in able to come to a conclusion on how it is used and the insight it gives to the design of a system. This was to be done through the development of a static analysis tool which would aid manual inspection of systems from the Qualitas Corpus.

Although a steep learning curve was initially off-putting, the tool was able to parse Java source code and identify nearly all instances of casting used in each system. The tool was applied to a total of five systems, followed by an in depth analysis using its output results. The final conclusions of each system were not as first expected, with the smallest system using the function the most relative to its size. It was clear that casting is a function widely used in open source systems to various degrees. It would be realistic to conclude that it is near impossible to develop a system of relative size and not use the casting function in multiple occasions.

6.0 References

Acellere, 2017. *Benefits of Static Code Analysis*. [Online]
Available at: <https://medium.com/acellere/benefits-of-static-code-analysis-a453b5d4a5e9>

[Accessed 11 07 2019].

Anon., 2011. *Visitor Design Pattern*. [Online]
Available at: <https://www.codeproject.com/Articles/186185/Visitor-Design-Pattern>

[Accessed 2019].

Anon., 2019. *Code Smells - Couplers - Message Chains*. [Online]
Available at: <https://refactoring.guru/smells/message-chains>

[Accessed 14 07 2019].

Anon., 2019. *Oxford English Dictionary*, s.l.: s.n.

Anon., 2019. *Visitor Pattern Design*. [Online]
Available at: https://sourcemaking.com/design_patterns/visitor

[Accessed 07 2019].

Anon., n.d. *Bloaters*. [Online]
Available at: <https://sourcemaking.com/refactoring/smells/bloaters>

[Accessed 06 2019].

Anon., n.d. *Object-Orientation Abuser*. [Online]
Available at: <https://sourcemaking.com/refactoring/smells/oo-abusers>

[Accessed 06 2019].

Anon., n.d. *Refactoring - Couplers*. [Online]
Available at: <https://refactoring.guru/refactoring/smells/couplers>

[Accessed 06 2019].

baeldung, 2019. [Online]
Available at: <https://www.baeldung.com/java-type-casting>

[Accessed 06 2019].

Boskovic, M., 2005. *PatternGuru: An Educational System for Software Patterns*, s.l.: s.n.

Budd, T., 1991. *An Introduction To Object-Oriented Programming*. 3 ed. s.l.:Reading, Mass. : Addison-Wesley Pub. Co. .

Corpus, Q., 2013. *Qualitas Corpus*, s.l.: s.n.

EclipseFoundation, 2019. *Eclipse Foundation*, s.l.: s.n.

Emden, E. v. & Moonen, L., 2002. *Java quality assurance by detecting code smells*. Richmond, IEEE.

Fokaefs, M., Tsantalis, N. & Chatzigeorgiou, A., 2007. *JDeodorant: Identification and Removal of Feature Envy Bad Smells*. Paris, IEEE.

Fowler, M. & Beck, K., 2000. *Refactoring : improving the design of existing code*. s.l.:Reading, MA : Addison-Wesley .

Ghahrai, A., 2018. *Static Analysis vs Dynamic Analysis in Software Testing*. [Online] Available at: <https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/> [Accessed 11 07 2019].

Girish, S., Samarthayam, G. & Sharma, T., 2015. *Refactoring for Software Design Smells*. s.l.:Elsevier Inc..

Gosling, J. et al., 2019. *The Java Language Specification*. [Online] Available at: <https://docs.oracle.com/javase/specs/jls/se12/html/index.html> [Accessed 11 07 2019].

JavaTPoint, 2016. *Java Base64 Encode and Decode*, s.l.: s.n.

JHotDraw7API, n.d. *Class FontFamilyNode*, s.l.: s.n.

Letouzey, J.-L. & Whelan, D., n.d. *Introduction to the Technical Debt Concept*. [Online] Available at: <https://www.agilealliance.org/wp-content/uploads/2016/05/IntroductiontotheTechnicalDebtConcept-V-02.pdf> [Accessed 06 2019].

Schildt, H., 2007. *Java : a beginner's guide*. s.l.:New York, N.Y. : McGraw-Hill.

Sciore, E., 2019. The Visitor Pattern. In: *Java program design principles, polymorphism, and patterns*. s.l.:Berkeley, CA : Apress L. P..

Sierra, K. & Bates, B., 2015. *Head First : Java*. s.l.:Sebastopol, CA : O'Reilly.

Spivak, R., 2015. *Let's Build A Simple Interpreter Part 7*, s.l.: s.n.

Suryanarayana, G., Samarthayam, G. & Sharma, T., 2015. *Refactoring For Software Design Smells: Managing Technical Debt*. Amsterdam ; Boston : Elsevier, Morgan Kaufmann .

Tempero, D. E., n.d. *Acquiring the Qualitas Corpus*, s.l.: s.n.

ZIAMOS, I., 2017. *Detecting Inheritance Hierarchy Smells*, Glasgow: University of Strathclyde.