Analysis on Inheritance in Open Source Systems

Edward McNealy

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

DEPT. OF COMPUTER AND INFORMATION SCIENCES

UNIVERSITY OF STRATHCLYDE

SEPTEMBER 2016

McNealy

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc Advanced Software Engineering 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

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

Signature:

Date:

August 29, 2016

Abstract

Inheritance is a key component in object-oriented programming, providing many of the benefits that are available through this programming paradigm. These benefits include increased modularity and code reusability, and the freedom to make changes to a single base class and have those changes applied to all who inherit that class. Inheritance also comes with some disadvantages, including a loss of encapsulation when the state of a class is shared among multiple children, and an increase in code complexity. Knowing when to correctly and efficiently use inheritance, and when to consider an alternative, is a valuable skill that any developer should strive towards.

By utilizing established software code metrics as a basis for analysis, this thesis aims to provide detailed observations on the use of inheritance in open source Java applications. These metrics help to reveal information about the depth of inheritance hierarchies, the risks associated with a class having many children and methods, the growth and complexity of complete inheritance hierarchies, and the effects of inheritance on methods implemented in both parent and child classes. This information can be helpful in comparing the ways different open source systems use inheritance.

An analysis tool has been developed for the purpose of investigating inheritance use in open source systems. The metrics mentioned above are used by this tool to analyze different aspects of inheritance and provide data that is easily consumed through charts and graphs. With the help of this tool and the code metrics, examples of both decent and poor design practices regarding inheritance will be revealed.

Acknowledgements

I would like to thank my thesis advisor Dr. Murray Wood of the Department of Computer and Information Sciences at the University of Strathclyde. Dr. Wood was extremely helpful during the process of completing the thesis, contributing many useful resources and helping to push the project in the right direction. Most importantly, he was always available to review my work and provide constructive feedback that kept me thinking on ways to improve.

I would also like to thank my parents, who have always encouraged me to continue learning, and who support me whenever I need help. Finally, I'd like to thank my brothers, for keeping me entertained and distracted (in a good way) during my time in university.

Edward McNealy

Table of Contents

Figures

Code Snippets

Tables

Chapter 1: Introduction to Inheritance

Inheritance plays a significant role in many aspects of software related technology today, such as artificial intelligence, object-oriented databases, and object-oriented programming (Thirunarayan, 2009). Taivalsaari claims that the ability of objects to inherit from one another in object-oriented programming is considered one of the main features that distinguishes this programming model from others. Many of the benefits achieved through object-oriented programming, such as improved conceptual modeling and reusability, are possible thanks to the concept of inheritance (Taivalsaari, 1996, p. 438).

So what is inheritance? This feature will be explained in full detail shortly, but the basic concept is that inheritance supports the ability of new object definitions to be modeled after ones that have already been defined, and gives these new objects properties of the parent that can be used in the same manner, given additional functionality, or removed altogether (Taivalsaari, 1996, p. 447). This promotes the reuse of code that has already been written, and provides additional benefits, such as the ability to interchangeably use objects that inherit from the same parent.

The use of inheritance can effect objects in many different ways, some beneficial, and some that cause increased complexity and loss of freedom to make compatible changes to existing classes (Snyder, 1986, p. 39). These problems might not always be apparent to a developer, and they might often use inheritance simply because it is available and provides some advantages, while not considering the negative side effects that may follow later in an application's lifetime. In order to better understand how inheritance might affect an application, there are certain code metrics and analysis methods that can be used to provide information on the impact inheritance has on related objects. The major questions to consider when doing this analysis are the following:

- Is there any risk or additional complexity for objects that are deep in an inheritance hierarchy?
- What is the correlation between an object that is inherited many times, and the methods that are implemented in that object?

- How does the hierarchy structure of one object change and grow as more separate objects inherit from the parent?
- What happens to methods that are defined in a parent object, when they are also implemented in child objects?

To help with the process of answering these analysis questions, a tool has been developed using the Java programming language and the **Eclipse JDT**. This application is designed to take Java software systems and parse the code into a tree-like structure that allows for inspecting the sections of code that relate to inheritance use. Then a number of calculations will be made that will provide data that can be analyzed to provide a deeper comprehension on the answer to the proposed questions.

The eleven systems that are to be analyzed are all open source Java projects. All but one of those systems come from the **Qualitas Corpus** (Tempero, 2013), which is a collection of software systems maintained by Ewan Tempero of the University of Auckland, and intended to be used for empirical studies of code artifacts. The final system is **JHotDraw** (Gamma & Eggenschwiler, 2007), which was developed by Erich Gamma and Thomas Eggenschwiler as a way of promoting well-known design patterns, identifying new patterns, and to be an example of a well-designed and flexible framework (Gamma & Eggenschwiler, 2007). The versions of each of these systems can be seen in *Table 1.1*.

System	Version
Ant	1.8.4
ANTLR	4.0
ArgoUML	0.34
Azureus	4.8.1.2
FreeCol	0.9.5
Hibernate	4.2.2
JHotDraw	6.0.1
JUNG	2.0.1
Junit	4.9
Lucene	4.2.1
Weka	3.7.9

Table 1.1 Versions of analyzed systems

The remainder of this introductory chapter will establish the terminology used throughout this thesis and provide details on the history of inheritance in object-oriented programming along with the definition of what inheritance actually means. There will also be information regarding two different types of inheritance available in programming languages today, and explain how inheritance is used in the language chosen for the application supporting the research for this thesis. Finally, there will be a section on some precautions to consider when using inheritance in any application.

1.1 Terminology

Throughout the scope of this paper there are some terms that will be used to refer to certain aspects of programming. When referring to an item that inherits from another, the words *subclass*, *child*, and *descendant* may be used. When referring to an item that is being inherited by another, the words *superclass, parent,* and *ancestor* may be used. The term *type* is used to define what kind of generic object is being used in Java, be it a class or an interface. For example, a *List<String>* is a *List* of objects of the type *String*.

Certain words and terms have been marked in order to recognize their type. When referencing a class or interface, the word is italicized, for example: *java.lang.Object*, or *AbstractFigure*. Method signatures or method names will also be in italics, for instance, *moveBy(int, int)* or *draw*. Reserved keywords in the Java programming language are in bold, such as **extends** or **super**. Applications or systems that were used for the purposes of the analysis are also in bold, like **Hibernate** or **Inheritance Inquiry**.

For examples of an application that uses inheritance, the project **JHotDraw** will be used. **JHotDraw** is an open source project created by Erich Gamma and Thomas Eggenschwiler. This application provides many representations of well-known design patterns that were popularized by Gamma, along with Richard Helm, Ralph Johnson, and John Vlissides. There are also many uses of inheritance throughout the program that might be used to as examples.

There are three design patterns that are mentioned throughout this thesis. These concepts are defined by Gamma et al. in their book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, et al., 1995, pp. 139, 175, 325, 331). In order to understand the patterns when they are used, the definitions are provided here.

- **Adapter –** "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."
- **Decorator –** "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

- **Template –** "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."
- **Visitor –** "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

1.2 Principles of Inheritance

Inheritance is one of the fundamental concepts of object-oriented programming, alongside encapsulation and polymorphism. One of the earliest introductions to inheritance was in the programming language Simula, developed by Ole-Johan Dahl and Kristen Nygaard in the 1960s. This language introduced the concept of "classes" and "objects", and can be seen as the first object-oriented programming language (Dahl, 2004). The technique of inheritance was defined by Dahl and Nygaard as *concatenation*, and could be described as the process of merging attributes from two different components (classes or objects), and the composition of their actions (Dahl, et al., 1972). By allowing new objects to be defined based on existing objects, only those properties that are not already declared in the existing class need to be declared in the new object. The properties of the base object will be available to the new object, and automatically included, without the need for redefinition (Taivalsaari, 1996). Based on this information, inheritance can be defined as the following:

Inheritance is the process through which objects are able to reuse the properties and behavior of already existing objects, thereby limiting the amount of code that needs to be rewritten, and providing a more modular approach to development.

Through the use of inheritance, a class will receive certain methods and properties from another class, allowing the inheriting class to have a similar state and structure as the parent class. Gamma, et al. note that "(w)hen a subclass inherits from a parent class, it includes the definition of all the data and operations that the parent class defines" (Gamma, et al., 1995). Methods and variables that are defined inside a parent class will be available to use in any subclasses, and objects that reference those subclasses can also call methods from the parent.

This is very useful in achieving code reuse throughout an application, as code can be defined in one class, and then simply inherited in others, avoiding multiple definitions of the same code. This type of code reuse is often recognized as “white-box reuse” (Gamma, et al., 1995, p. 31). As noted by Crnogorac et al, the main benefit received through the use of inheritance is the reduced amount of effort needed in implementation, as classes that are already developed and tested can be branched into new classes, and flexibility is obtained since code changes can be limited to only one class in certain circumstances (Crnogorac, et al., 1998, pp. 572-573).

With inheritance now defined, it is important to understand another concept of OO programming: encapsulation. Alan Snyder defines encapsulation as “a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces” (Snyder, 1986, p. 39). Using encapsulation allows a developer to update the implementation details of an object, without having to change the clients that use that object (Micallef, 1988, p. 10). While inheritance can be seen as a white-box, in that the internal details of a class are known to its children, encapsulation can be seen as a black-box. The details of an object’s implementation are unknown to clients, all the client is aware of is the interface that is **public** from that object.

In addition to the reusability benefits of inheritance, another important feature is type substitution. When one class inherits from another, the subclass is allowed to be used in the code in place of the superclass. The benefit afforded through type substitution is that the details of an object’s implementation can be hidden away, and the client that uses that object only needs to be aware that the object is of the desired type. The code also becomes more modular as many different types can be passed as a reference that is asking for one specific type of class. An example of type substitution can be seen in **JHotDraw**. Consider a **JHotDraw** application that requires a tool which can only create *Figures* that are based on a rectangle. The details of how this would work are shown in *Code Snippet 1.1* and explained in the following paragraph.

First, a *RectangleBasedTool* can be created, which **extends** *CreationTool*, and accepts any *object* of the type *RectangleFigure* as a parameter for its constructor. The **super** constructor is called, passing in the *RectangleFigure*. *CreationTool* requires a *Figure* as the parameter, and because

RectangleFigure **implements** *Figure*, this is valid and the class is complete. Now in any **JHotDraw** app, the *RectangleBasedTool* can be used for figure creation. However, unlike a *CreationTool* which can be used with any *Figure*, this new tool will only accept a class that inherits from *RectangleFigure*. Any type of *RectangleFigure* can be substituted as a parameter for this tool, and because the tool is not restricted to one **final** class, it is possible to switch between those types at run-time.

```
public class RectangleBasedTool extends CreationTool {

  // Only accepts classes that inherit from RectangleFigure
  public RectangleBasedTool(DrawingView view, RectangleFigure prototype) {
    super(view, prototype);    // super constructor accepts any Figure
  }
}

public class CreationTool extends AbstractTool {
  private Figure fPrototype;

  public CreationTool(DrawingView view, Figure prototype) {
    super(view);
    fPrototype = prototype;
  }
}

public class RecatangleExampleApp extends DrawingApplication {
  // ....
  @Override
  protected void createTools(JPanel palette) {
    Tool rectTool = new RectangleBasedTool(view(), new RectangleFigure());
    Tool diamondTool = new RectangleBasedTool(view(), new DiamondFigure());
    Tool triangleTool = new RectangleBasedTool(view(), new TriangleFigure());
    // Add tools...
  }
}
```

Code Snippet 1.1 RectangeBasedTool.java example for JHotDraw

In contrast to the benefits of inheritance mentioned above, there are also considerations to make regarding how to safely use inheritance. With inheritance, the implementation details of a subclasses are based on the parent class, forming a contract between the parent and any children. This contract will limit the rate at which a developer can safely make changes to the parent class, as those changes may have unintended side effects on the subclasses (Snyder, 1986, p. 39). Modifications to a parent class will often force the children classes to change as well (Gamma, et al., 1995, p. 31). Because classes using inheritance are defined at compile-time,

another restriction associated with its use is that the implementation details from a parent class are fixed and cannot be changed at run-time (Gamma, et al., 1995, p. 31).

When studying classes in a system, is it important to consider the relationship between those classes. According to Bertrand Meyer, "inheritance represents the *is* relation, also known as *is-a*" (Meyer, 1988, p. 498). Using **JHotDraw** as an example, there is a *ConnectedTextTool* class which **extends** *TextTool*, so it can be said that "every *ConnectedTextTool* is a *TextTool*". This relationship is a contrast against the one provided by composition. Object composition is an alternative to inheritance, in which a class contains a reference to another object (preferably an interface), and any actions required by the containing class are delegated to the composition object. The relationship with composition is a *has-a* relation. Using this alternative provides a "more workable and flexible extension mechanism" where objects can be easily added or removed at run-time of an application (Gamma, et al., 1995, pp. 43-44).

There are generally two types of inheritance: *single inheritance* and *multiple inheritance*.

1.3 Single Inheritance

With single inheritance, a class is only able to inherit information from one parent. That parent can then also inherit from another, and so on for each subsequent parent. In Java, this inheritance path can theoretically continue on an infinite amount of times, although in a study on the maximum depth of inheritance, Dr. Heinz Kabutz found that a *StackOverflow* error was thrown after compiling and running a class with 61 parents, and that a class with 1001 parents would not compile (Kabutz, 2006).

In examination of JHotDraw, the figure classes can be used as an example on the use of inheritance in a Java application. Most of the visual components of a JHotDraw application will be based around some type of *Figure*. All of these classes will implement the *Figure* interface, which defines the contract for the basic methods needed to display, move, connect to, and modify the component. There is an abstract base class for figures available for use through inheritance, called *AbstractFigure*. This contains default implementations of many of the methods defined by the *Figure* interface, as well as some abstract methods that child classes will be expected to implement. Because *AbstractFigure* already **implements** the *Figure*

interface, any subclasses of *AbstractFigure* will naturally implement *Figure* as well, through inheritance. This causes any classes using this inheritance to also be of the *Figure* type, creating the possibility for type substitution of those classes wherever a *Figure* interface is required.

There are additional abstract figures that extend the base *AbstractFigure* class, such as *CompositeFigure* and *AttributeFigure*, adding additional functionality while still being flexible for changes. Whenever a figure class needs to be developed, it should inherit from one of these classes. An additional interface for joining figures, *ConnectionFigure*, is also inheriting from the *Figure* interface. The hierarchy structure for *Figure* can be seen in *Figure 1.1*, with the *Figure* interface being the root node of the tree. In this diagram, interfaces are gray, and classes are blue. This hierarchy was generated through the use of services available with the tool developed for the purpose of this research, which will be discussed in further detail in Chapter 3. Note that the labels for the selected 5 nodes were added after the graph's creation, and only provided for the 5 objects mentioned here, ignoring the remaining *Figure* classes.

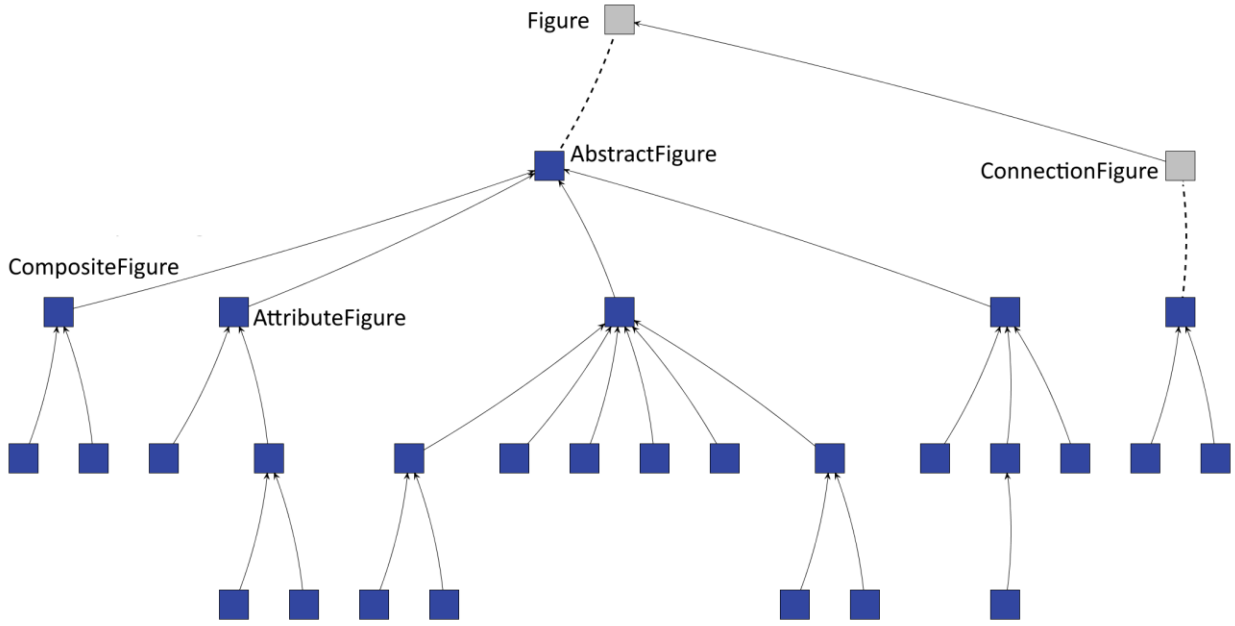

Figure 1.1 Figure hierarchy from JHotDraw

1.4 Multiple Inheritance

The definitive difference between single and multiple inheritance is simple: multiple inheritance allows a class to inherit from an arbitrary number of parents. In actual programming practice, the difference can be much more complicated, as are some special considerations that must be taken into account. One of the main problems that arises from allowing multiple inheritance is termed the "*deadly diamond of death*" by Robert Martin. This results from a situation in which a class *A* is the top-level of a hierarchy, with two classes *B* and *C* inheriting from *A*, and class *D* inheriting from *B* and *C*. When *B* and *C* override a method *foo* defined in *A*, and *D* does not provide an overridden implementation, then the version of *foo* that will be used is unclear (Martin, 1997). There are ways to resolve this issue, such as feature renaming (Meyer, 1988, pp. 535-540), as well as individual solutions provided for those languages that do support multiple inheritance.

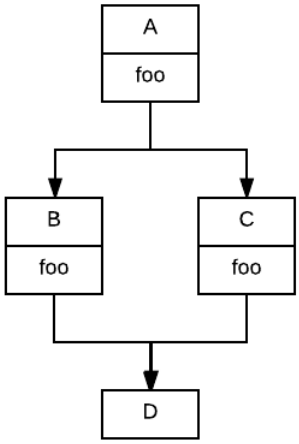

Figure 1.2 Deadly diamond of death

1.5 Inheritance in Java

Before inheritance in the Java programming language can be explained, access modifiers must also first be understood. An access modifier is the rule used by Java to declare how methods and variables can be given or denied access to other classes. When decorated with the **private** modifier, an item will be only available to the class that declares it. The **public** modifier is the opposite, as **public** items are available at a global scope. The **protected** modifier gives access of an item to a subclass, or to another class in the same package. When no modifier is given, a default known as package-private is used, which gives access of an item to other classes in the same package. Inheritance in Java relies heavily on the **protected** access modifier in order to allow subclasses to use methods and variables that are declared in the parent.

In the Java programming language, inheritance of a class is declared using the **extends** keyword. By saying that class *A* **extends** class *B*, it can be inferred that class *B* is the parent class, and class *A* is the child class. *A* will now be able to make use of any properties from *B*,

provided they are decorated with the **protected** or **public** keyword. Properties in the child class can override ones that are declared in the parent. This is achieved by using the same name or method signature as a variable or method in the parent. The *@Override* annotation can be used to ensure at compile-time that a method is in fact overriding another. Inheritance is the way that parents can share their state with any of their children. That state can be **protected**, and only available between the parent and its children, or **public**, and available to any client that uses either the parent or child.

To access a property that is declared in a parent class, the child class can make use of the **super** keyword. When creating a constructor for a subclass, the **super** keyword can be used to first call the parent class constructor. This is required when a parent has a constructor with parameters and does not provide a no-argument constructor, or the no-argument constructor is **private** to the parent. A JHotDraw example of using **super** in a constructor can be seen in *Code Snippet 1.2*. When the *RadiusHandle* calls *super(owner)*, the constructor in *AbstractHandle* is invoked.

```
public abstract class AbstractHandle implements Handle {

  private Figure fOwner;

   * Initializes the owner of the figure.
  public AbstractHandle(Figure owner) {
    fOwner = owner;
  }
  //...
}

public class RadiusHandle extends AbstractHandle {

  private RoundRectangleFigure fOwner;

  public RadiusHandle(RoundRectangeFigure owner) {
    super(owner);
    fOwner = owner;
  }
  //...
}
```

Code Snippet 1.2 RadiusHandle.java and AbstractHandle.java from JHotDraw

Methods that are overridden by a subclass are the ones that will be accessed when that subclass is used as a reference in another class. To explain this further, the *RadiusHandle* class

from JHotDraw can again be examined. This class inherits from *AbstractHandle*, which provides a default implementation for the *draw()* method, used to display the handle on the GUI. The *AbstractHandle* class draws handles with a rectangular shape. The *RadiusHandle* is used to create handles who have a circular shape. Because of this, the default implementation of *draw()* cannot be used in *RadiusHandle*. Instead, this class will override the *draw()* method. Wherever a reference to a *RadiusHandle* is used, this overridden method will be used in place of the *AbstractHandle draw()* method. In JHotDraw, Handles are created within the drawing view by looping through each instance of Handle and calling the *draw()* method. The code in *Code Snippet 1.3* shows the details of how this works.

```
public abstract class AbstractHandle implements Handle {

  public AbstractHandle(Figure owner) {
    // ...
  }

  //Default draw method Template method
  @Override
  public void draw(Graphics g) {
    Rectangle r = displayBox();     // Gets the display bounds for the handle

    g.setColor(Color.white);
    g.fillRect(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawRect(r.x, r.y, r.width, r.height);
  }
}

public class RadiusHandle extends AbstractHandle {

  public RadiusHandle(RoundRectangleFigure owner) {
    super(owner);
    // ...
  }

  @Override
  public void draw(Graphics g) {
    Rectangle r = displayBox();     // Gets the display bounds for the handle

    g.setColor(Color.yellow);
    g.fillOval(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawOval(r.x, r.y, r.width, r.height);
  }
}

public class StandardDrawingView extends // ...
                                                                    {
  /**
   * Draws the currently active handles.
   */
  @Override
  public void drawHandles(Graphics g) {
    // All the handles
    // In this case, the handles we are using are RadiusHandles
    Enumeration<Handle> k = selectionHandles();
    while (k.hasMoreElements()){
      // Calling the draw() method in RadiusHandle
      (k.nextElement()).draw(g);
    }
}
```

Code Snippet 1.3 Method overriding from JHotDraw

In addition to simply overriding a method, a subclass is able to extend a method from its superclass. For cases when a subclass wants to continue using a superclass's method, but also needs to provide additional functionality, the subclass might extend the method functionality. The method will be overridden in the subclass, but still call the **super** implementation (Lorenz & Kidd, 1994, p. 69). This is similar to the way a subclass might include a call to the parent's constructor when implementing its own constructor. However, the **super** call is not restricted to being the first statement in the extended method, and can be called at any time inside the method body.

Code Snippet 1.4 shows an example of how method extending is accomplished. A *BorderDecorator* is a type of *Figure* that uses the **Decorator** design pattern to add a border around any other *Figure*. This is done by first rendering the actual *Figure* using the call to *super.draw(g)*, which calls the *draw()* method in *DecoratorFigure*. After this, *BorderDecorator* continues with the extended functionality of rendering a border around the boundaries of the *Figure* to be decorated.

```
public class BorderDecorator extends DecoratorFigure {

  public BorderDecorator(Figure figure) {
    super(figure);
  }

   * Draws the figure and decorates it with a border.
  @Override
  public void draw(Graphics g) {
    Rectangle r = displayBox();
    super.draw(g);
    g.setColor(Color.white);
    g.drawLine(r.x, r.y, r.x, r.y + r.height);
    g.drawLine(r.x, r.y, r.x + r.width, r.y);
    g.setColor(Color.gray);
    g.drawLine(r.x + r.width, r.y, r.x + r.width, r.y + r.height);
    g.drawLine(r.x , r.y + r.height, r.x + r.width, r.y + r.height);
  }
}

public abstract class DecoratorFigure
                extends AbstractFigure
                implements FigureChangeListener {

   * The decorated figure.
  protected Figure fComponent;

   * Constructs a DecoratorFigure and decorates the passed in figure.
  public DecoratorFigure(Figure figure) {
    // ...Sets figure = fComponent and sets up listeners
  }

   * Forwards draw to its contained figure.
  @Override
  public void draw(Graphics g) {
    fComponent.draw(g);
  }
}
```

Code Snippet 1.4 Method extending from JHotDraw

All classes in Java have as their top level ancestor the *java.lang.Object* class. This class does not need to be explicitly declared; instead it is automatically assigned by Java. *Object* provides all classes in Java with some standard methods, such as *equals()* and *toString()*. While these methods have default functionality, it is usually helpful to override these methods and provide a custom implementation, with *equals()* determining if an object's variables are the same as another, and *toString()* providing a more informative description of a class. The inheritance structure that is provided by *Object* to all classes in Java ensures that every class will have *Object* as the element at the very top level of their hierarchy. Because of this, the hierarchy

level for *Object* can be noted as 0, with all other classes starting with the hierarchy depth being at 1, unless they inherit from another class.

Java does not allow for a class to use multiple inheritance, so this will not be a concern for the scope of this research. The closest that Java comes to multiple inheritance is through the design of interfaces. A class is allowed to implement more than one interface, which will create a contract on that class that expects the class to contain certain method signatures. Since interfaces only contain abstract contractual method signatures and cannot be initialized as an object, they do not typically contain the dangers associated with multiple inheritance. In Java 8, interfaces are allowed to contain **default** methods, which do provide an implementation in the interface. This can introduce a diamond problem, however classes with this problem will cause compiler errors in order to prevent this.

1.6 Dangers of Inheritance

While inheritance is beneficial to the structure of an OO system, there are also drawbacks that are included, and considerations that must be made towards the effects against other components of the system. Snyder explains that "(p)ermitting direct access to inherited instance variables weakens one of the major benefits of object-oriented programming, the freedom of the designer to change the representation of a class without impacting its clients" (Snyder, 1986). This means that because classes that inherit from another have access to properties and methods from their parent, and use them among their own local properties and methods, changes to the parent can result in unexpected changes to the children. For an example of this, consider an abstract class that is implementing a **Template** method. A **Template** method delegates part of an algorithm to subclasses. The work done in the delegated steps usually depends on the operations being performed in the skeleton, which is defined in the parent. If those parent operations are altered, the actions done by the children may have unintended side-effects. Developers need to consider this before making changes to the parent, and include tests that ensure the behavior is unchanged after the **Template** is altered.

A problem with inheritance identified by Snyder is that "the instance variable operations defined on a class for the benefit of its descendants may not necessarily be appropriate for

users of instances of the class" (Snyder, 1986). In order to correct this problem, Java provides access modifiers that should be used in the appropriate situations. The use of "getters" and "setters" as operations to access instance variables from a class and its parent should be avoided when possible. Allen Holub explains that while the use of these operations is somewhat standard in many Java applications, it is in fact not quite a good object-oriented practice (Holub, 2003a). Inheritance must work alongside encapsulation in order for a system to follow OO guidelines. As the primary goal of encapsulation is to contain the scope of an object to solely that object, and prevent other objects from being aware of any details of the variables in a class, getters and setters should be used sparingly. According to Holub, "you should avoid getter and setter functions since they mostly provide access to implementation details" (Holub, 2003a). In an article discussing the issues of getters and setters, Greg Jorgensen states that "getters and setters should be avoided because they break the encapsulation OOP offers" (Jorgensen, 2008).

Snyder notes that, when using inheritance and a local instance variable and a variable inside a parent have the same name, then changing the name of the parent variable will potentially affect the behavior of the child. He also notes that raising an error at compile-time for instance variables in a child and parent with the same name would not be appropriate, as these variables should be separate instances (Snyder, 1986). In Java, this is avoided because access to variables in a parent class are only available when that variable has the **protected** or **public** access modifier, and unless the variable is accessed through the use of the **super** keyword, it is assumed that the variable is the local instance.

Inheritance is often used to define a "base" class that contains functionality that is common among many classes. Those classes can then extend the base class to gain that functionality, avoiding the problem of rewriting code in multiple locations. However, this results in what is recognized as the "fragile base-class problem", which means that modifications to a parent may seem to have no adverse effect on their children, but can cause problems in those classes. In the article "Extends is Evil" by Holub, he states that "(y)ou can't tell whether a base-class change is safe simply by examining the base class's methods in isolation; you must look at (and test) all derived classes as well" (Holub, 2003b). He also notes that you must examine other

classes that contain references to both the parent and the child classes that are being modified, since this coupling between classes can result in unintended side-effects in these classes as well. The solution to his proposal that "extends is evil" is to avoid inheritance through superclasses with the **extends** keyword, and instead prefer to use inheritance through interfaces, with the **implements** keyword. This is directly related to OO design principle of "*program to an interface, not an implementation"* (Gamma, et al., 1995). The advantages gained through the use of interface inheritance are the following, again provided by Gamma, et al.:

- "Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect."
- Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface."

One way to achieve this interface based inheritance is by using the **Adapter** design pattern. The **Adapter** will "convert the interface of a class into another interface clients expect" (Gamma, et al., 1995). Object composition can be used to delegate the actions of the **Adapter** to another class, circumventing the need for the **Adapter** to inherit from another class. In this way, the **Adapter** can still adhere to the interface that the client expects, without inheriting any extra properties through a parent class.

1.7 Inheritance Summary

Inheritance can be defined as *a process through which objects are able to reuse the properties and behavior of already existing objects*, which is also known as white-box reuse. Some of the features available through the use of single inheritance is the sharing of properties and behavior from one class with many children, and the ability to substitute one type of subclass with another, as long as they have the same parent. Inheritance use can also include inconvenient side effects. This includes restrictions on the changes that can be made to a parent class, as those changes can alter the behavior of any subclasses, and may require those subclasses to also be modified. There might also be conflicts with names of properties and methods in a child class that match ones defined in the parent. However, this can be handled

by the programming language used, such as Java's use of the **super** keyword to distinguish between child and parent properties and methods.

Chapter 2: Software Code Metrics

The software quality-measurement values that are used in this research will be described in this chapter. This will include the definition for each code metric, viewpoints that should be considered before making use of a metric, and information on how to interpret each metric. While interfaces in Java are able to use inheritance, and are even able to inherit from multiple parents, they are unable to have concrete definitions of methods, and cannot be initialized as an *Object*. Because they have no effect on the inheritance hierarchy of *Objects*, **interfaces will not be considered** in any of the code metrics used throughout the scope of this project.

2.1 Overview on Established Metrics

In order to measure the degrees to which a software system meets certain qualifications, code metrics can be used to extract information about the system. When inspecting an element in an application to determine the quality of design choices regarding the use of that element throughout the scope of the application, there are two important metrics that may be considered. These metrics are the *depth of inheritance* and the *number of children*. These metrics were conceptualized by Shyam Chidamber and Chris Kemerer, and are part of a software metrics suite consisting of four other metrics, for a total of six (Chidamber & Kemerer, 1994). Another useful metric is the *weighted methods per class*, which will be helpful in this thesis by revealing information about how class inheritance impacts the amount of methods cascading down the inheritance tree.

The *Figure* classes used in **JHotDraw** are good examples for showing the use of inheritance in an application. Since **JHotDraw** is a graphical-based application used for drawing diagrams and shapes, there are many instances of *Figures* that can be used. All *Figures* in **JHotDraw** extend the *AbstractFigure* class, making this class the top-level of their inheritance hierarchy. The *AbstractFigure* implements the *Figure* interface, which forms a contract that declares the methods that all *Figure* classes will be expected to implement. *AbstractFigure* will be useful in explaining the details on *number of children*.

In the same manner that Chidamber and Kemerer make assumptions about the distribution of properties for their metrics, there is a fact that is apparent in this thesis (Chidamber & Kemerer, 1994). For each of the analyzed systems that are in use for the purposes of this research, we can be certain about the fact that the system *does* make some use of inheritance, and therefore is composed of both classes who have no parents (by making no use of inheritance at all, or by being at the root of the inheritance hierarchy tree) and classes that extend another class (using inheritance).

2.2 Depth of Inheritance Tree

A basic definition for the *depth of inheritance tree* (DIT) for an application that only allows for single inheritance would be the amount of elements that a single element is inheriting from, all the way to the root of the inheritance hierarchy tree. According to Chidamber and Kemerer, "(t)he depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree" (Chidamber & Kemerer, 1994, p. 480). The DIT is a useful tool for measuring the complexity of a class and the application that class is a component of by providing a strong overview of the various levels of inheritance hierarchies in an application.

When the depth of a class's hierarchy tree is realized, a developer can determine how complex that class might be. With a deep hierarchy, there may be many classes that have to be navigated through before a property or method can be reached. Taking the *TriangleFigure* from **JHotDraw** as an example, we can inspect the amount of work necessary to trace a function for this class. A developer might be interested in seeing how this *Figure* is moved around the screen. Searching through *TriangleFigure*, there is no method having to do with movement, so the superclass *RectangleFigure* must then be inspected. This class contains a *basicMoveBy(int,*

int) method, but the name implies that there is more to this function. However, in the next superclass, *AttributeFigure*, there is again no definition for a method with behavior that would move a *Figure*. So one more superclass, *AbstractFigure*, must be examined. Here we can find the abstract definition for *basicMoveBy()*, as well as a method *moveBy(int, int)*. In this method statement, the *basicMoveBy()* method is called. Taking one final step into the *Figure* interface will show that the *moveBy(int, int)* method is defined here, so the developer knows that the *AbstractFigure* implementation is where the method originates from. This process can be viewed in the UML diagram in *Figure 2.1*. Tracing a method through an inheritance hierarchy can be simplified in some ways, such as using breakpoints to step through code execution while debugging, or through the use of an IDE such as Eclipse, which allows for quickly jumping to an implementation.

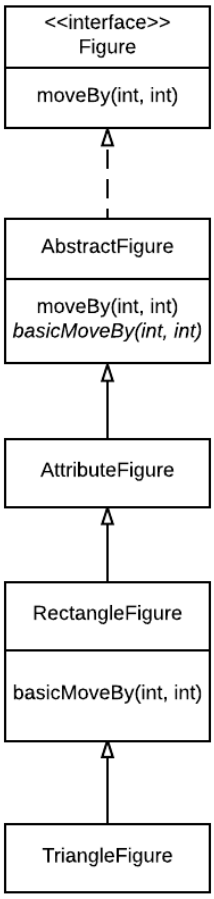

Figure 2.1 TriangleFigure UML diagram

As the root of all classes, *java.lang.Object* has a depth of inheritance of 0. So in the case of a class that does not extend any other classes, the depth of inheritance would be 1. A subclass of that class would have a depth of inheritance of 2, and the number continues to increase by one for each subsequent child. Since we know that each system that is being analyzed for this project is using inheritance in some way, it can be recognized that there will be classes whose DIT is greater than 1. Since the inheritance trees will consist of branching nodes, it can also be realized that there will be at least two classes who share the same level of depth of inheritance. For example, in **JHotDraw** the *AbstractTool* is a base class that does not inherit from any other class, so its DIT is 1. Both *CreationTool* and *SelectionTool* inherit from *AbstractTool*, so they have a DIT of 2.

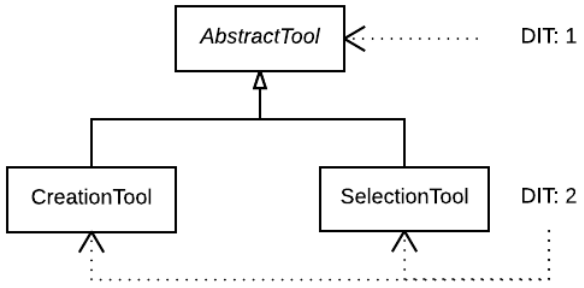

Figure 2.2 DIT levels of tools from JHotDraw

There are three main viewpoints to consider in regards to depth of inheritance. According to Chidamber and Kemerer (Chidamber & Kemerer, 1994, p. 483) those viewpoints are:

1) "The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior."
2) "Deeper trees constitute greater design complexity, since more methods and classes are involved."
3) "The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods."

Methods and variables declared in a base class will be available to any classes that extend the base, provided those methods or variables are **protected**, **public**, or package-private (and the inheriting class is in the same package). As a class becomes deeper in an inheritance hierarchy, it becomes more complex because of the additional methods and properties already defined in each of its parents, all the way to root class. This increased complexity is the trade-off that results from the white-box reuse that inheritance provides.

In the analysis of the depth of inheritance levels in a project, the DIT number provides information about how well the project is making use of the OO fundamental concept of inheritance. A lower number implies that a class is less complex, but possibly not taking advantage of minimal code reuse through inheritance. A higher number implies that the project is exploiting the benefits of inheritance code reuse, but that there will most likely be an increase in complexity and possible code errors (Naboulsi, 2011). While there is no concrete ideal level for depth of inheritance, many suggestions are that the limit on the DIT should be no greater than 5. (Kabutz, 2006), (Tandon, 2010) A pilot study conducted by Harrison et al with the assistance of final year BSc. Computer Science students in the University of Southampton reached a similar conclusion. Their study on the effect of code changes to classes using inheritance found that between 3 and 5 levels of inheritance is ideal (Harrison, et al., 2000, p. 6).

2.3 Number of Children

The *number of children* (NOC) can be defined as the amount of elements that directly inherit from another single element, or the "(n)umber of immediate descendants of the class" (Chidamber & Kemerer, 1994). According to Radu Marinescu, the metric for number of children "represents the number of immediate subclasses subordinated to a class in the class hierarchy" (Marinescu, 1998, p. 3). Ramanath Subramanyam and M.S. Krishnan described the number of children as "a count of the number of immediate child classes that have inherited from a given class" (Subramanyam & Krishnan, 2003, p. 298). Every time that a class inherits from another class, the number of children for the superclass increases. In contrast to the depth of inheritance, which describes how deep a hierarchy tree travels from the root node (or the height of the hierarchy), the number of children measures the width of a hierarchy, or how far the branches of the tree will spread.

Chidamber and Kemerer again give three viewpoints on the NOC metric (Chidamber & Kemerer, 1994, p. 485):

1) "Greater the number of children, greater the reuse, since inheritance is a form of reuse."
2) "Greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing."
3) "The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class."

By allowing many classes to inherit from one parent, the amount of code needed can be reduced, as methods that are common between similar classes is contained in the parent. When the number of children for a class reaches a high amount, this can signify that more abstraction may be needed in the children classes, and perhaps more abstract classes are needed to bridge between the parent and the many children. Arti Chhikara and R.S Chhillar state that "if there are a large number of children of a class, then the abstraction level of that

parent class is reduced." (Chhikara & Chhillar, 2012, p. 365) They recognize this as an indication of the misuse of inheritance. The high number of children will also have an impact on the effectiveness of testing against that class, as the scope that the methods reach is wider than the scope of those classes who have fewer descendants.

The metric for the number of children only includes the immediate classes that inherit from a parent. This means that the graphs of the hierarchy trees for NOC will only extend two node levels deep. The width of these trees, or the number of branches extending from the root node, can be any number greater than 0 (meaning there will be at least 1 subclass).

Through the analysis of the number of children for each class in a project, it can be determined if that project is making good use of the concept of inheritance. If there are many classes who have children, then it is obvious that there are many classes who are inheriting from another class. Marinescu notes that this metric is helpful in determining the misuses of subclassing, and that this may call for a restructuring of a class's hierarchy. He determined that there are two situations in which a high NOC value may prompt a redesign (Marinescu, 1998, p. 3):

- *Insufficient exploitation of common characteristics* – When there are multiple classes who share similar functionality, such as graphical-based classes, or repository classes that access a database, there is a high potential for reuse. If there is a pattern of classes with these characteristics who have a high number of children, then the system may need to be refactored to have some of the common functionality abstracted to another class to be placed higher in the hierarchy.
- *Root of the class hierarchy* – When the root class in a hierarchy has a very large amount of children compared to other classes in the system, refactoring should be considered for that class. The hierarchy may currently be represented as a *tree*, where there is one class at the root, with many children branching off from that root. Instead, it may be wise to consider splitting this large tree into a *forest*, where there will be a number of trees with a smaller amount of branches. This could result in less complexity, as the trees are smaller and easier to navigate and test.

The studies done by Chidamber and Kemerer suggested that classes usually have only a small number of children, with a small deviation of classes who are inherited by many children. Two explanations given for this are that inheritance is not being considered during the planning and designing of new classes, and that there is a lack of communication between programmers of different classes, resulting in missed opportunities for code reuse (Chidamber & Kemerer, 1994, p. 486).

2.4 Weighted Methods per Class

The metric of *weighted methods per class* (WMC) for an object provides the number of methods defined in a class. Chidamber and Kemerer provide the following formula for determining the WMC:

- "Consider a Class C_1 with methods $M_1,\ldots,M_n$ that are defined in the class. Let $C_1,\ldots,C_n$ be the complexity of methods. Then: $WMC = \sum_{i=1}^{n} c_i$ "

In addition to this formula, Chidamber and Kemerer provide 2 key viewpoints on the use of the WMC metric:

- "The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class".
- "The larger the number of methods in a class the greater the potential impact on children, since children will inherit all methods defined in the class".
    - Note: In Java, and for the purposes of this research, this is not always the case, as methods can be declared as **private**, restricting access to any other class.
- "Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse".

Classes with a high method count become much more complicated, and require more time to work through. A developer who is unfamiliar with a class will need to spend time analyzing the methods in that class in order to understand how they relate to the application. As the method count grows, it is likely that those methods will only be useful within the scope of the

application they are defined, unless they are specifically designed to be a part of an external library or application programming interface (API). In a study on detecting design flaws detection in large scale systems, Marinescu notes that "classes with very high *WMC* values are *critical in respect of the maintenance effort* and they might be therefore redesigned in order to reduce the overall complexity level of the class" (Marinescu, 1998, p. 2). His study suggests that in order to reduce WMC, a class can be split into multiple classes.

Most importantly for this research, methods in a class can have a direct influence on the complexity of any subclasses. When one class uses methods that are defined in another class, those classes form a relationship, or become coupled (Chidamber & Kemerer, 1994, p. 479). As more classes have access to an increasing number of methods, there will be more coupling between objects. This will reduce the modularity of an application and increase its complexity, as it may become difficult to trace the origin of method definitions. This might also discourage encapsulation, as objects are sharing their state and allowing access of methods to more objects.

When calculating the WMC of a class, the value for the metric will increase by one for each method defined within that class. This includes any getter or setter methods that are used for accessing class variables. In addition to any method implementations specific to a class, the WMC also includes any overridden methods defined in a parent class. These methods count as ones unique to the implementing class. However, any methods that are defined in a superclass will *not* be included in this metric. The WMC is limited to only those methods in the class which is being analyzed. When creating hierarchies for each of the classes in a project, the method count will then be used to provide details on how those inherited methods affect subclasses. These guidelines for calculating the weighted method count are based on those used by the Project Analyzer developed by Aviosto (Aviosto, 1997).

2.5 Code Metrics Summary

Chidamber and Kemerer have defined a suite of metrics that can be helpful in providing information on the quality of software. The metrics that are important to the purpose of this

thesis are the depth of inheritance tree, the number of children, and the weighted methods per class.

As the DIT of a class increases, the behavior of that class can become more difficult to control, and indicates a higher number of methods and superclasses are involved. The potential reuse of the inherited methods increases as well, which can mean less code being written, as well as more coupling between classes.

A high number of children indicates a large amount of reuse, but can also be an indication that classes are not being designed properly, as common characteristics of the children could possibly be refactored to another class higher in the hierarchy. In addition, classes with many children might require more testing of the methods defined in those classes.

The WMC helps to keep track of the amount of methods that each class has defined, which becomes useful in determining how the method count will be affected by classes using inheritance. Coupling of methods between classes can have an impact on the complexity of those classes.

Chapter 3: Inheritance Inquiry Application

This chapter will discuss the software used during the development process and the implementation details of the analysis program created for the purposes of this research. The application is developed as a library named **Inheritance Inquiry**. The code metric results described in the next chapter are obtained using this library, and most of the graphs used as examples in this research are generated by the services available in the library.

3.1 Main Application Software

The **Inheritance Inquiry** application was developed using the Java programming language. The latest release version of Java, Java 8, was chosen in order to provide the most current Java APIs, to gain experience working with the new lambda expressions feature, and to ensure that any external libraries were compatible. The Eclipse IDE was used for writing the code for the application. In order to easily include dependencies on the open source projects that were used for support in the tool, the Maven nature was enabled on the project. Any libraries available in the Maven Repository could simply be added to a pom.xml file in the project, and were then available for use in the application.

The tool is based on the analysis of software systems, so there needs to be support for analyzing lines of source code as nodes in a stack and viewing the relationships between different nodes in all sections of the system. This is the same functionality that is utilized by IDEs to provide features such as advanced searching of type usages, jumping between declarations and implementations of classes and methods, and class outlines, among other features. One such open source project that supports this process is **JavaParser** (Viswanadha & Gesser, 2016). This project was originally considered for this research, and was used in the early stages of development. However, the **Eclipse JDT** (The Eclipse Foundation, 2016) was eventually chosen to replace **JavaParser**, as **Eclipse JDT** contains more advanced features and offers better support in the long term, for any future use of the **Inheritance Inquiry** tool. Since the two libraries are similar in some ways, much of the code used in the development process using **JavaParser** was able to be converted for use with **Eclipse JDT**.

3.2 Eclipse JDT

The **Eclipse Java development tools (JDT)** is an open source set of tool plug-ins that assist in building the Eclipse IDE. These tools are what allow Eclipse to have a Java project nature and a development perspective focused towards Java applications. The many different views, editors, and wizards are all built with the **Eclipse JDT**. With these tools, any developer can create a plugin for Eclipse that can aid in their development process, and that can be shared with other users of Eclipse. Thanks to these tools, Eclipse is able to speed up the development process of an application, through the use of helpful functions such as advanced searching, jumping between code definitions and implementations, easy project building, and code refactoring. According to Fuhrer et al, Eclipse was one of the first IDEs that popularized the use of refactoring during development of an application (Fuhrer, et al., 2007, p. 31).

The **Eclipse JDT** consists of four main components: the APT, Core, Debug, and UI. Core is the component that will be focused on for this project. The JDT Core provides the basic infrastructure necessary to support a Java application in an IDE. This includes the APIs that are used to inspect and to manipulate Java source code, which are the Java Model and the Java Document Model, respectively.

When developing a Java project in Eclipse, there are features available in the IDE that assist with quickly navigating between resources in the project and with viewing details of the classes in the program. The Java Model contains the hierarchal information needed to represent these details to the user of the IDE. These elements are all inheriting from the *IJavaElement* interface, and include (among others):

- *IJavaProject*: Representation of a single Java application, containing all the resources in that app.
- *ICompilationUnit*: A Java source file containing the code for an application, which is in a file with the **.java** extension
- *IMethod*: A declaration of a method or constructor inside of a Java type.
- *IClassFile*: The **.class** file of a compiled Java type.

By using these elements, the Package view of the Eclipse IDE can display a graphical representation of a Java project, which assists with keeping track of and accessing all the resources in a program, as well as allowing for creating new resources. *Figure 3.1* shows the details provided by the Java Model in a standard Java development environment within Eclipse.

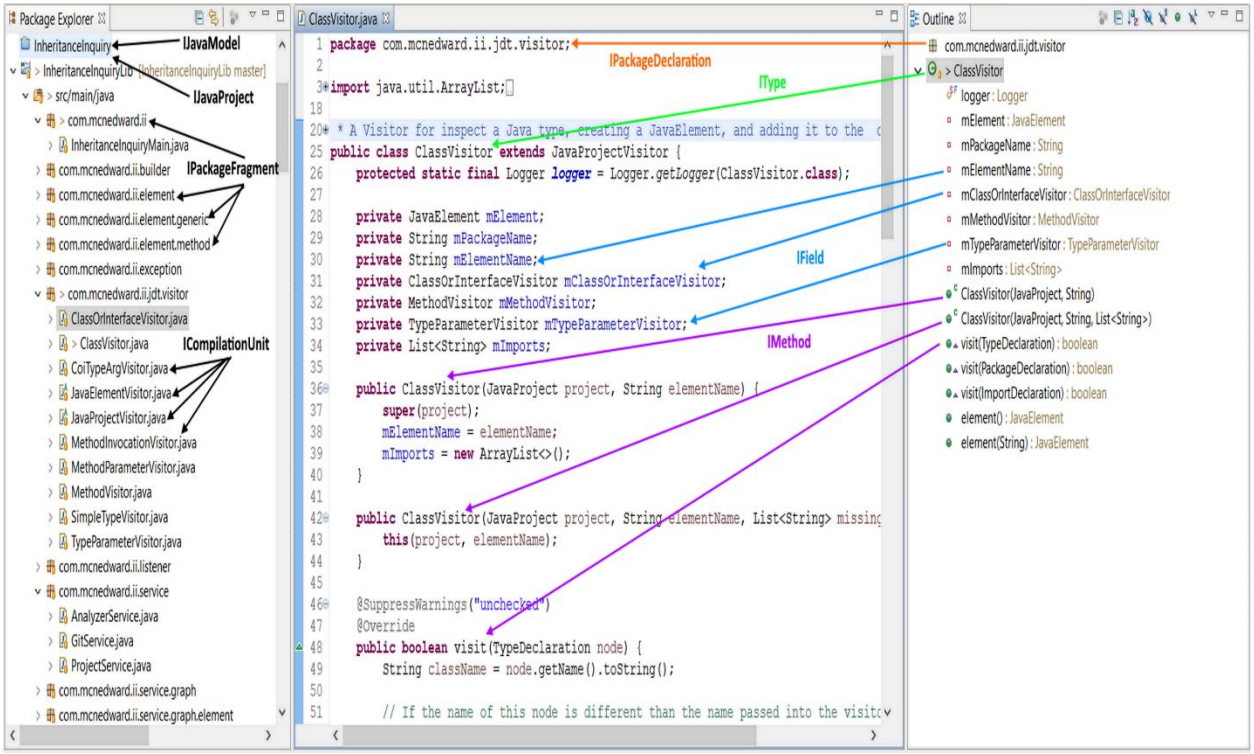

Figure 3.1 Use of Java Model in Eclipse

In order to use the elements of the Java Model, access to an Eclipse workspace is required. There are internal methods in the JDT API that build the elements, and in order to use them they must be accesses through the interface methods of an *IWorkspaceRoot*. Because of this, the Java Model elements can only be used through an Eclipse plug-in. The Java Model is not needed as part of the **Inheritance Inquiry** tool, and is only described to provide background information regarding the **Eclipse JDT**.

The Java Document Model is used to manipulate the source files for a Java project. This includes refactoring properties in a class, creating classes and interfaces, and adding new methods or properties. These modifications are accomplished be manipulating a

CompilationUnit, and the elements contained within the *CompilationUnit*. This is the root node of an Abstract Syntax Tree.

3.2.1 Abstract Syntax Tree

In the same manner that an XML file contains many different nodes that make up a tree structure, the Abstract Syntax Tree (AST) represents the nodes of a Java source file as a tree model. Changes made to the AST will be reflected as modifications to the Java code (Kuhn & Thomann, 2006), such as removing or updating a property in a class or moving the declaration of some property. The AST also provides essential information about how each node is being referenced within the scope of the entire Java project. This information is referred to as "bindings". A binding can be used to determine if one node is the same instance as another node. This can be useful in distinguishing between variable references, and methods declared and used in various classes.

There is a standard workflow for generating and using an AST. This usually involves 5 steps:

1. Provide the source code that will be represented by the AST, which can be in one of two forms: a **.java** file or Java source code in a *char[]*.
2. Parse the source code using the *ASTParser* provided by the JDT.
3. Retrieve the AST from the parsed source, along with any bindings, if they are requested in options specified to the *ASTParser*.
4. Make any modifications to the AST, such as renaming properties or moving declarations.
5. Apply those modifications and save the changes to the source code.

The details of this process can be viewed in *Figure 3.2*. For the purposes of this research, the **Inheritance Inquiry** tool only makes use of steps 1-3. This is because the app has no need for modifying the source code of its analyzed systems, only studying the ways in which inheritance is being used.

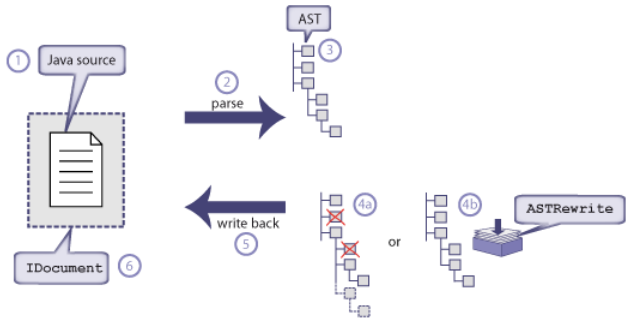

Figure 3.2 Standard AST workflow process (Kuhn & Thomann, 2006)

Every node within the AST is using inheritance by being a subclass of the type *ASTNode*. These subclasses are all designed to represent a specific element in Java, and provide methods that help to access information about the functionality of their respective element. Any binding generated in the AST inherits from the *IBinding* interface. This interface provides information on the kind of binding (package, type, variable, method, annotation, or member-value pair), and provides the contract for the methods necessary for an *IBinding*, such as the name, the modifiers, or the type. The details of the *ASTNode* subclasses and the *IBinding* interfaces that were utilized for the **Inheritance Inquiry** tool are displayed in *Table 3.1*. For the Documentation, anything underlined is taken from the JavaDoc comments for that type.

Type Name	**Documentation**
SimpleType	Type node for a named class type, a named interface type, or a type variable.
TypeParameter	Type parameter node (added in JLS3 API).
TypeDeclaration	Type declaration AST node type.
PackageDeclaration	Package declaration AST node type. Node where the package of a class or interface is declared.
ImportDeclaration	Import declaration AST node type. Node where a separate package is imported inside a class or interface.
MethodDeclaration	Method declaration AST node type. Node where a method is declared inside a class or interface.
SuperMethodDeclaration	Simple or qualified "super" method invocation expression AST node type.
MethodInvocation	Method invocation expression AST node type. Node where a method is invoked by an object.
IMethodBinding	A method binding represents a method or constructor of a class or interface.
ITypeBinding	A type binding represents fully-resolved type.
Expression	Abstract base class of AST nodes that represent expressions.

Table 3.1 ASTNodes and IBindings used for Inheritance Inquiry

3.2.2 Visitors

The **Visitor** design pattern is an essential component for analyzing an object structure that contains many different types of elements. This design pattern allows for performing operations on various objects, based on the concrete implementation of those objects. The basic intention of the **Visitor** design pattern, according to Gamma et al (Gamma, et al., 1995, p. 331), is the following:

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Each node in the AST represents a Java element that has unique behavior and properties. There are specific operations that must be performed on each node for the **Inheritance Inquiry** tool, and these operations are dependent on the concrete type of the node. They cannot rely simply on an interface, as these concrete types may not all implement the same interface. It would not be practical to have these operations defined inside of the class, as they may be distinct and unrelated to the behavior of the class, and they will pollute that class with unnecessary complications (Gamma, et al., 1995, p. 333). The **Visitor** will help to circumvent this problem.

There are two key components in the **Visitor** pattern: the Visitor and the Element. In this application, the Element will be the abstract *ASTNode* class. Every node is able to "accept" a Visitor, which means that the Element is allowing the Visitor to perform operations on that Element. This is handled through the *accept(ASTVisitor)* method in *ASTNode*. This method takes as a parameter a subclass of the abstract type *ASTVisitor*, performs some initial setup (if implemented by the subclass), accepts the Visitor subclass (which is where the main operation is performed), and then performs a final conclusive method (if implemented by the subclass). This is further explained by *Code Snippet 3.1*.

```
public abstract class ASTNode {

   * Accepts the given visitor on a visit of the current node.
  public final void accept(ASTVisitor visitor) {
    if (visitor == null) {
      throw new IllegalArgumentException();
    }
    // begin with the generic pre-visit
    if (visitor.preVisit2(this)) {
    // dynamic dispatch to internal method for type-specific visit/endVisit
      accept0(visitor);
    }
    // end with the generic post-visit
    visitor.postVisit(this);
  }

    /**
     * Accepts the given visitor on type-specific visit of the current node.
     * This method must be implemented in all concrete AST node types.
     */
    abstract void accept0(ASTVisitor visitor);
}
```

Code Snippet 3.1 ASTNode.java from Eclipse JDT

The *ASTVisitor* contains all of the default method implementations for visiting the *ASTNode* Elements, all named *visit()* and doing nothing but return **true** as a default, and taking as a parameter a concrete type of *ASTNode*, for example, *visit(SimpleType), visit(TypeDeclaration), visit(MethodInvocation)*. This is shown in a small sample of the code for the *ASTVisitor* in *Code Snippet 3.2*. When there is a need for certain operations against a type of node, a new Visitor can be created, inheriting from *ASTVisitor*, and then overriding any of the *visit()* methods for the *ASTNodes* that are needed for those operations. In the context of the **Inheritance Inquiry** application, the *ClassVisitor* can be previewed as a prime example. **Visitors** in the **Inheritance Inquiry** tool will be a subclass of either *JavaProjectVisitor* or *JavaElementVisitor*. These abstract base classes inherit from *ASTVisitor* and simply store a reference to the *JavaProject* or the *JavaElement* that is being visited. Shown in *Code Snippet 3.3*, the *ClassVisitor* is used at the top level of the class or interface hierarchy (the *CompilationUnit*) to create and setup a *JavaElement* (which will be discussed in the following section), and then delegate more nodes in the class to be visited. These nodes are: the type parameters that may be declared as part of this type's declaration, the superclasses (the most important part for this research!), the interfaces, and the method declarations in the type.

```
public abstract class ASTVisitor {

  public void preVisit(ASTNode node) {
    // default implementation: do nothing
  }

  public void postVisit(ASTNode node) {
    // default implementation: do nothing
  }

  public boolean visit(SimpleType node) {
    return true;
  }

  public boolean visit(TypeDeclaration node) {
    return true;
  }

  public boolean visit(MethodInvocation node) {
    return true;
  }
}
```

Code Snippet 3.2 ASTVisitor.java from Eclipse JDT

```
/**
 * A Visitor for inspecting a Java type, creating a JavaElement, and adding
it to the correct JavaPackage.
 */
public class ClassVisitor extends JavaProjectVisitor {
  // Visitors used within this Visitor
  private ClassOrInterfaceVisitor mClassOrInterfaceVisitor;
  private MethodVisitor mMethodVisitor;
  private TypeParameterVisitor mTypeParameterVisitor;

  public ClassVisitor(JavaProject project, String elementName) {
    super(project);
    // ...
  }

  @Override
  public boolean visit(TypeDeclaration node) {
    // ...
    // Visit all the interfaces
    List<ASTNode> interfaces = node.superInterfaceTypes();
    for (ASTNode inter : interfaces) {
      mClassOrInterfaceVisitor.setIsInterface(true);
      inter.accept(mClassOrInterfaceVisitor);
    }

    // Visit the super class
    Type superClassType = node.getSuperclassType();
    if (superClassType != null) {
      // If this node is an interface, then it's "extends" will be as well
      mClassOrInterfaceVisitor.setIsInterface(node.isInterface());
      superClassType.accept(mClassOrInterfaceVisitor);
    }

    // Visit the methods
    for (Object declaration : node.bodyDeclarations()) {
      if (declaration instanceof MethodDeclaration) {
        ((MethodDeclaration) declaration).accept(mMethodVisitor);
      }
    }
}
```

Code Snippet 3.3 ClassVisitor.java from Inheritance Inquiry

In *Code Snippet 3.3*, the *ClassVisitor* contains 3 additional visitors: *ClassOrInterfaceVisitor* for visiting classes that are being inherited and interfaces being implemented, *MethodVisitor* for visiting method statements, and *TypeParameterVisitor* for visiting generic type parameters of a class (this one is not shown being used in this example). In the *visit(TypeDeclaration)* method, first all the interfaces are visited, then the superclass, then any statements in the class declaration that are a method. After visiting each of these nodes in the AST, the *JavaElement* that represents the class will be composed.

3.3 Implementation Details

The purpose of the **Inheritance Inquiry** library is to analyze a number of Java applications to extract information on the ways that they make use of inheritance. The library is designed to be able to process a single Java project, a system containing multiple versions of a single Java project, or one project for each of many different systems. This is accomplished by using the *ASTParser* from **Eclipse JDT** to create *CompilationUnits* for every **.java** file in an application, then visiting those *CompilationUnits* and inspecting the nodes for inheritance use, and finally calculating the code metrics for the application. Once this process is complete, additional services can be utilized to provide further analysis and visual representations of the data. There are a few distinct components that work together as a part of this build process:

- Elements
- Builders
- Tasks
- Services

3.3.1 Elements

The Elements are the way that the **Inheritance Inquiry** tool represents aspects of the Java programming language, much in the same manner as the Java Models of **Eclipse JDT** does. However, instead of being used to help build plugins for Eclipse, the sole purpose of the Elements are to store information about the Java objects for later use in analysis.

3.3.1.1 JavaProject

The *JavaProject* is used as the root of all other Elements, with the same hierarchal structure as a standard Java application. This contains the name of the project, the path to the project on the file system, the name of the system which the project is a part of, and the version of the project. In addition to this, all of the *JavaPackages* in the project are kept as a reference here.

There are some helpful methods within the *JavaProject* that are used throughout the tool. The most important of these methods are:

- *find(String)*: Searches the project for an existing Java class or interface

- *findPackage(String)*: Searches the project for an existing Java package
- *findOrCreateElement(String, String, boolean)*: Returns an existing Java class or interface, or creates one and adds it to the appropriate package if it does not exist
- *findNumberOfChildrenFor(JavaElement)*: Creates a list of all the immediate subclasses of a Java class or interface
- *getClasses()*: Finds and caches a list of all the Java classes and interfaces in the project

3.3.1.2 JavaPackage

The *JavaPackage* contains a list of all the Java classes and interfaces that are declared in a specific Java **package**. It also contains an important method for use in the searching for *JavaElements*:

- *find(String)*: Searches the package for an existing Java class or interface

3.3.1.3 JavaElement

A *JavaElement* is the representation of a Java type, either a class or interface. This includes the name, the *JavaPackage* it is contained in, a list of all packages the type is importing, and a flag to determine if the type is an interface or not. There is also a list of all the other *JavaElements* that the type makes use of, either as a parent, or an implemented interface, and finally, there is a list of all the *JavaMethods* that the type declares. The *JavaElement* also contains cached lists of both the interfaces and superclasses that it uses, since these are accessed often in the analysis component of the tool.

3.3.1.4 JavaSolution

The *JavaSolution* contains the analyzed results of the code metrics and hierarchy tree generations from the tool. The name of the project, name of the system that the project is a part of, and the version of the project are all kept in the *JavaSolution*. In addition to this, lists of the DIT, NOC, and WMC metrics are available in the form of their respective Java representation. These metrics simply contain the value for the metric and the name of the Java object they are generated from. Finally, there are lists of the hierarchy trees for DIT, NOC, and the entire object tree structure of an element.

3.3.2 Builders

The **Inheritance Inquiry** tool contains different processes by which a project can be built and analyzed. These processes are controlled through the Builders. There is a base class named *Builder*, which contains some helper methods used in the Build task. For every Java project that needs to be analyzed, a concurrent task is started and ran in a separate thread. The *Builder* contains the methods for managing these tasks. Each Builder that inherits from this class simply needs to provide an implementation for where to find the project files, then submit tasks for each of those files. *Code Snippet 3.4* (explained below) shows the *SystemBuilder*, which is used to build and analyze all versions of one project in a system. In addition to this Builder, there is the abstract *QCBuilder*, designed for building systems from the **Qualitas Corpus**. The Builders that inherit from this are the *GraphBuilder* and the *MetricAnalysisBuilder*, used for building graphs and building Excel charts, respectively. The *QCBuilder* contains an additional method that must be implemented, *handleSolutions(List<JavaSolution>)*, which can be used to perform additional work against a set of *JavaSolutions*. This is helpful in comparing how different systems might be using inheritance.

```
public final class SystemBuilder extends Builder {

  public SystemBuilder() {
    super();
  }

  @Override
  protected void buildProcess() throws TaskBuildException {
    // Create a File[] for all the projects in a system
    // ...
    for (File projectFile : projects) {
      submit(new StandardBuildTask(projectFile, system.getName()));
    }
    waitForTasks();
  }
}
```

Code Snippet 3.4 SystemBuilder.java from Inheritance Inquiry

In the *SystemBuilder*, all that is required is to override the *buildProcess()* method. In this method, the location for a system with multiple versions is given, and then for each *File* in that system, a new *StandardBuildTask* (defined in the next section), is submitted, using the *submit(Job)* method. Finally, the *waitForTasks()* method is called, which is defined in the base

Builder class. This method will just pause the method execution until all of the *StandardBuildTasks* have completed. Once they are all complete, the entire build and analysis process will be complete, and any graphs, charts, or other analysis information will be produced.

3.3.3 Tasks

The tasks for the **Inheritance Inquiry** tool all inherit from the parent *IIJob* class, which implements *java.util.concurrent.Callable* through the *Job* interface. This class contains references to each of the Services that are needed throughout the building and analysis process. The *JavaProject* is built by this class, and then analyzed and converted to a *JavaSolution*. The concrete implementations of *IIJob* are then responsible for determining what type of analysis to run on that solution. These concrete classes include the *StandardBuildTask*, which runs through all the processes and services, and Tasks aimed towards only processing solutions for graphs and Excel charts.

The code shown in *Code Snippet 3.5* explains how this process works in the *StandardBuildTask*. The *call()* method is from *java.util.concurrent.Callable*, and this calls the *buildProjectAndAnalyzeForSolution()* method. Inside this method, the *ProjectService* builds the *JavaProject*, and then the *AnalyzerService* analyzes that project. Once this is finished, the abstract *processSolution(JavaSolution)* method is called, which is implemented in each concrete *IIJob* class. This can be seen in the *StandardBuildTask*, which calls both the *MetricService* and the *GraphService* to produce the Excel data and the graphs for the hierarchies.

```
public abstract class IIJob<T> implements Job<T> {
  private ProjectService mProjectService;
  private AnalyzerService mAnalyzerService;
  private MetricService mMetricService;
  private GraphService mGraphService;
  // ...
  @Override
  public T call() {
    JavaSolution solution = buildProjectAndAnalyzeForSolution();
    T result = processSolution(solution);
    // ...
    return result;
  }

  private JavaSolution buildProjectAndAnalyzeForSolution() {
    JavaProject project = mProjectService.build(mProjectFile, mSystemName, mListener);
    return analyze(project);
  }

  protected abstract T processSolution(JavaSolution solution) throws // ...

  protected JavaSolution analyze(JavaProject project) {
    return mAnalyzerService.analyze(project);
  }
}

public class StandardBuildTask extends IIJob<Void> {
  @Override
  public Void processSolution(JavaSolution solution) throws GraphBuildException {
    metricService().buildMetrics(solution);
    graphService().buildGraphs(solution);
    return null;
  }
}
```

Code Snippet 3.5 IIJob.java and StandardBuildTask.java from Inheritance Inquiry

3.3.4 Services

Services are used as the main area of work for each of the steps in the build and analysis process. There are four main Services: *ProjectService, AnalyzerService, GraphService,* and *MetricService*. A Service is also available for downloading projects from remote **Git** repositories, though this Service is not currently being utilized in the tool.

3.3.4.1 ProjectService

The *ProjectService* contains three methods for building a project, based on the different options that might be used, such as passing a *String* as the file path, or using an actual *java.io.File* object instead. This Service is where **Eclipse JDT** is needed, as the *JavaProject* is built here. The *ASTParser* is setup for each project that needs to be built, and *CompilationUnits* are generated from the parser. Once these are created, the required nodes are then visited, using the

appropriate Visitors to create *JavaElements* for each *CompilationUnit*. The result of each of the three **public** methods in this Service is a complete *JavaProject*.

3.3.4.2 AnalyzerService

The *AnalyzerService* is used to calculate all of the metrics and hierarchy structures for each *JavaProject*, and return a *JavaSolution* with that information. This is done through one of the following methods:

- *analyze(JavaProject)*: Calculates everything; all metrics and the hierarchy structures for both DIT and NOC. This also provides additional information about how methods are used in the *JavaElements*.
- *analyzeMetrics(JavaProject)*: Calculates just the metrics and hierarchy structures.
- *analyzeForDIT(JavaProject)*: Calculates just the metrics and hierarchy structures for DIT.
- *analyzeForNOC(JavaProject)*: Calculates just the metrics and hierarchy structures for NOC.
- *analyzeForFullHierarchy(JavaProject)*: Calculates just the complete hierarchy structures. This is mainly used for generating graphs of the complete hierarchy of objects.

This Service calculates the metrics for DIT and NOC by building Java objects to represent the inheritance hierarchy of a class. All of the subclasses in the hierarchy are stored in a *Stack*, and the metric can be found by examining the size of that *Stack*. This is also used for building the graphs and charts in the *GraphService* and *MetricService*.

```
private List<DitHierarchy> travelHierarchies(JavaElement element) {
  List<JavaElement> parents = element.isInterface() ?
                        element.getInterfaces() : element.getSuperClasses();

  List<DitHierarchy> hierarchies = new ArrayList<>();
  if (!parents.isEmpty()) {
    for (JavaElement parent : parents) {
      DitHierarchy parentHierarchy = new DitHierarchy(parent, true);
      hierarchies.add(parentHierarchy);

      List<DitHierarchy> parentHierarchies = travelHierarchies(parent);
      parentHierarchy.tree.add(parentHierarchies);

      // Add to the inherited method count
      inheritedMethodCount += parentHierarchy.elementMethodCount;
    }
    tree.add(hierarchies);
  }
  return hierarchies;
}
```

Code Snippet 3.6 Method for building DIT hierarchies

The code shown above in *Code Snippet 3.6* is the recursive method that is responsible for building the hierarchy for a DIT metric. The *JavaElement* that is passed into the method is the class or interface that this hierarchy will be built for. A *List<JavaElement>* is retrieved, which contains the superclasses for that *JavaElement*. This is a *List* only because an interface may have multiple parents; since this thesis is only concerned with class inheritance, this *List* will only have one element in it during the analysis. Next, a new hierarchy (the *DitHierarchy* object) will be initialized using the superclass, and then the superclass will also be traversed recursively for more hierarchies. This continues until all the classes in the hierarchy have been added, and the DIT can then be found as the size of the hierarchy tree stack (the *tree* variable in the code snippet).

3.3.4.3 GraphService

The *GraphService* generates a graph that displays the hierarchy structure of either the DIT or NOC metrics. This Service is also able to create graphs for entire object hierarchy trees, and graphs that show the effects of overriding or extending methods. The images for the graphs are created using the **Java Universal Network/Graph Framework**, or **JUNG** (SourceForge, 2010). This library generates graphs and network structures based on a set of nodes and edges that are defined by the user of the library. **JUNG** utilizes some components of the *javax.swing*

framework for creating the visual aspects of the graphs. Once the graphs have been defined and plotted, an image of the graphs can be copied and saved. The **JHotDraw** *Figure* hierarchy structure graph shown in *Chapter 1* in *Figure 1.1* was created by first using the *analyzeForFullHierarchy(JavaProject)* method from the *AnalyzerService*, then using the resulting *JavaSolution* in the *buildFullHierarchyTreeGraph(JavaSolution, String)* method from the *GraphService*. The diagram in *Figure 3.3*, showing the hierarchy of the *Assert* class from the **JUnit** system was also generated by the *GraphService*.

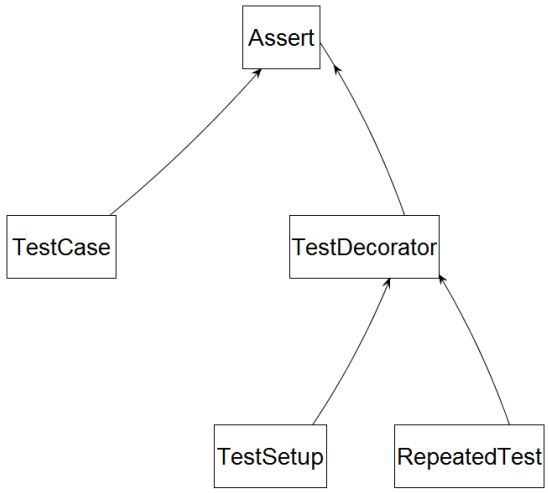

Figure 3.3 Assert.java hierarchy from JUnit

3.3.4.4 MetricService

The purpose of the *MetricService* is to generate files that can be read by Excel to display tables of data that might be converted into charts. This Service has a main function where it is able to accept a *JavaSolution*, then build valid Excel files containing information on each of the metrics of every *JavaElement* in that solution. An example of this is shown in *Table 3.2* for the NOC metrics in the **ArgoUML** system.

ArgoUML - Number of Children	
Class or Interface	**Number of Children**
org.argouml.application.api.AbstractArgoJPanel	12
org.argouml.application.api.org.argouml.application.api.AbstractArgoJPanel	1
java.lang.Exception	9
java.lang.RuntimeException	6
org.tigris.swidgets.Dialog	1
org.tigris.swidgets.PopupButton	1
javax.swing.JPanel	37
org.argouml.util.ArgoDialog	17
org.argouml.ui.UndoableAction	77

Table 3.2 ArgoUML NOC metric from MetricService

The *MetricService* also contains methods that will build an Excel acceptable file for the DIT, NOC, and WMC metrics of all the *JavaElements* in a *List<JavaSolution>*. The data for these files consists of a count of the number of *JavaElements* that have a value for certain metric. For example, *Table 3.3* shows the results of the DIT metrics in the different systems from the **Qualitas Corpus** and **JHotDraw**. Taking the **Ant** system as an example, we can see the differing values for depth of inheritance throughout this system. The Java types that are analyzed for these metrics are just the classes; interfaces are ignored. There are 666 classes who have a DIT of 1 (meaning those classes have no parent except for *java.lang.Object*), 261 who have a DIT of 2, and so on up to 3 classes with a DIT of 7. By looking at *Figure 3.4*, we can see one of those classes that has a DIT level of 7.

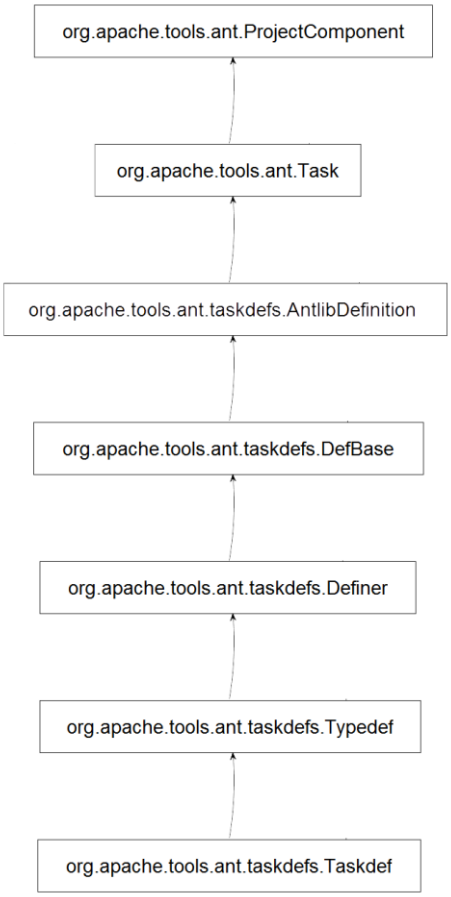

Figure 3.4 Taskdef.java from Ant system in Qualitas Corpus

System	1	2	3	4	5	6	7	8	9	Total Elements
Ant	666	261	387	139	44	10	3	0	0	1510
ANTLR	395	110	53	44	17	12	1	0	0	632
ArgoUML	1161	515	476	283	72	13	2	0	0	2522
Azureus	2894	514	294	56	19	6	0	0	0	3783
FreeCol	744	139	164	100	26	0	0	0	0	1173
Hibernate	4695	899	1019	268	78	37	32	5	4	7037
JHotDraw	96	58	44	9	0	0	0	0	0	207
JUNG	669	128	36	15	2	0	0	0	0	850
JUnit	179	56	8	7	0	0	0	0	0	250
Lucene	2159	717	865	562	38	5	3	0	0	4349
Weka	853	494	308	61	5	3	0	0	0	1724

Table 3.3 DIT levels for Qualitas Corpus systems and JHotDraw

3.4 Inheritance Inquiry Summary

The **Inheritance Inquiry** library utilizes the **Eclipse JDT** in order to parse Java source code and visit the nodes in the Abstract Syntax Tree of the Java object. The code metrics discussed in *Chapter 2* can then be calculated and processed through the use of different Services available from the library. This results in graphs displaying the hierarchy trees of different metrics and Excel readable files that can be used to visualize the data. The build and analysis process of the **Inheritance Inquiry** tool completes the workflow shown in *Figure 3.5*, explained as:

- The Builder locates the files for the projects to process
- A Task is submitted for each of the projects
- Within those tasks, the Services are used to complete the work necessary for the processing of the building and analyzing the Elements
    - *ProjectService* parses and visits the **.java** files and creates a *JavaProject.*
    - *AnalyzerService* calculates the code metrics and hierarchy trees for every *JavaElement* in the *JavaProject*, and creates a *JavaSolution* for further processing
    - *GraphService* creates the **JUNG** graphs for visualizing hierarchies and the effects of inheritance on methods.
    - *MetricService* creates Excel readable files for analyzing the data from the *JavaSolution* metrics.

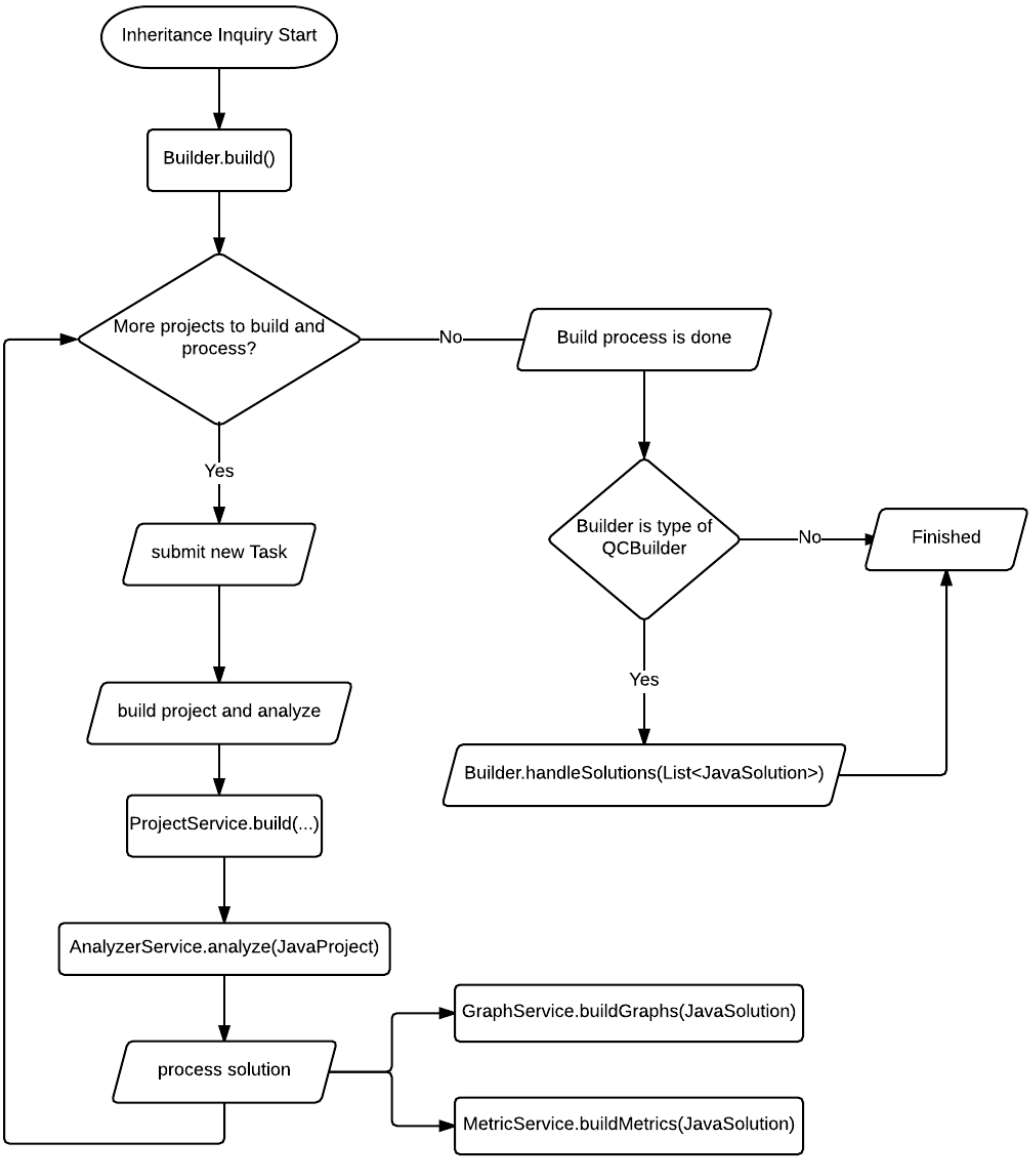

Figure 3.5 Inheritance Inquiry build workflow

Chapter 4: Analysis

In *Chapter 1*, four questions were raised, that are intended to be answered through the analysis provided by this thesis. Those questions are:

- Is there any risk or additional complexity for objects that are deep in an inheritance hierarchy?
- What is the correlation between an object that is inherited many times, and the methods that are implemented in that object?
- How does the hierarchy structure of one object change and grow as more separate objects inherit from the parent?
- What happens to methods that are defined in a parent object, when they are also implemented in child objects?

Each of the eleven proposed systems has been investigated to gather data in order to find the answers to these questions. For the systems coming from the **Qualitas Corpus**, the latest versions available are being used. For **JHotDraw**, the version being used is 5.1. The following sections will first examine the details on inheritance use in these systems, including the metrics discussed in *Chapter 2*, the inheritance hierarchy structure, and how methods are effected by the use of inheritance. Finally, there will be a quick overview on each individual system, with deeper investigation being provided for a few of the systems with more interesting results.

In the following analysis of the 11 systems, it is important to note that the average values for DIT **do not consider classes with a DIT of 1**. This means that the averages provided are **only** calculated for those classes that inherit from a class other than *java.lang.Object*. As mentioned at the start of *Chapter 2*, interfaces are not being considered for their inheritance use in any of the analysis provided by this thesis. The amount of classes with a DIT of 1 account for more than 60% of all classes for 6 systems, and more than 44% for the other 5. As this is a study on inheritance use, and having a DIT of 1 means no inheritance, those classes do not need to be considered.

This adjustment does not need to be considered for NOC, since this metric is applied to any class that uses the **extends** keyword to inherit from another class, and Java classes do not

inherit from *Object* in this way. In the same manner, when **inheritance use** is mentioned, *Object* is also not taken into account. Only those classes that explicitly inherit from another by using the **extends** keyword are considered.

4.1 Multi System Analysis

To begin the analysis on each of these systems as a group, the overall usage of inheritance will be examined. *Table 4.1* shows this information, with first the total amount of classes defined for each system, and then the percentage of those classes that are using inheritance. More than half of the systems have a lower percentage, with the ranges varying from 21% to 37%. The other 5 systems use inheritance in over half of their classes, but never reaching higher than 56%.

System	Classes	% of Inheritance Usage
Ant	1510	55.89%
ANTLR	632	37.50%
ArgoUML	2522	53.97%
Azureus	3783	23.50%
FreeCol	1173	36.57%
Hibernate	7037	33.28%
JHotDraw	207	53.62%
JUNG	850	21.29%
JUnit	250	28.40%
Lucene	4349	50.36%
Weka	1724	50.52%

Table 4.1 Total classes and percentage of inheritance usage for analyzed systems

The system **JUnit** uses inheritance among its class only 28% of the time. Since **JUnit** is a testing framework, this may actually be preferred. In an article on the reasons against the use of inheritance in tests, Petri Kainulainen claims that inheritance can have negative performance side-effects on a test suite, as well as causing tests to become harder to understand. He examines how **JUnit** handles class hierarchies for each test in the suite. The entire hierarchy must be walked through, by using reflection, to find all methods with specific annotations, and then invoking those methods. This traversal must occur before and after a test class is ran, and then before and after every test in that class. (Kainulainen, 2014) In order to reduce the strain on processing tests, and to keep the complexity of a test suite low, **JUnit** has kept the usage of inheritance to a minimum.

In contrast to the low usage of **JUnit**, **JHotDraw** has one of the highest percentages of inheritance use out of the analyzed systems. One of the purposes of **JHotDraw** is to promote well-known design patterns, many of which were authored in part by Erich Gamma, who was also one of the main developers of **JHotDraw**. As many of those design patterns rely heavily on

inheritance, it makes sense that the usage of inheritance in this system would be high. For instance, all of the figures that **JHotDraw** renders to the display are inheriting from *AbstractFigure*. Many other components also have a base parent class, such as the locators with *AbstractLocator*, handles with *AbstractHandle*, and tools with *AbstractTool*. All of these instances are good examples on the use of inheritance. They promote code reusability by declaring common behavior in parent classes. The system is designed with the intent for using type substitution to allow any concrete class to be switched for another, making the application very flexible in terms of creating new types of objects and being able to use them immediately.

4.1.1 DIT Ranges

By reviewing the ranges of the levels of the depth of inheritance trees for each of the analyzed systems, some patterns in design choices are shown. Each of the systems contain a large amount of classes whose DIT is 1, meaning that they do not inherit from any other concrete class (besides *java.lang.Object*, which is disregarded in this study). The chart in *Figure 4.1* shows the variance in DIT levels. The bars in gray represent those classes who do not use inheritance, the bars in blue represent those classes who are extending one class for a DIT of 2, the orange bars shows the classes that have a DIT of 3, and so on for every DIT level in the system. The standalone classes are a majority for 5 out of the 11 systems. For the other 6, they still represent a large portion of the classes. However, those 6 have a better range for the DIT levels present, as can be most easily recognized for **Ant**, **ArgoUML**, and ***Lucene***, whose chart bars have a diverse spread of colors representing their DIT levels.

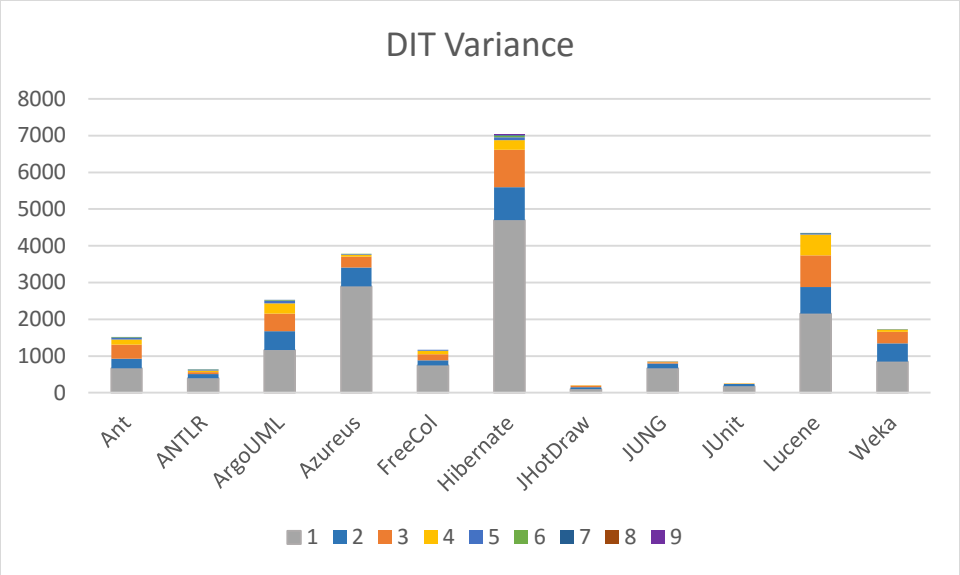

Figure 4.1 DIT Variance in analyzed systems

As was noted in *Chapter 2,* at the end of *Section 2* on the *Depth of Inheritance Tree*, there is no set definitive level for DIT, but some suggestions are that the depth should be within the range of 3 and 5. Lower DIT values are not indicative of any sort of problems, but as the depth increases, it can possibly signify added complexity and higher defects (Subramanyam & Krishnan, 2003, p. 307). In order to gain an overview on how deep inheritance hierarchies range for each system, the standard depth of classes can be found. This will only include those classes that inherit from another, or those whose DIT is greater than 1. This can be referred to as the *DIT Standard* for a system.

Before the *DIT Standard* can be found, it must be noted that in order to create an inheritance tree, a class must be inherited at least once, meaning that there will always be a class whose DIT level is at least 2. The DIT level will always increment by one, and can never skip a level. What this means is that for a system that has a max DIT of 6, there will be at least one class whose DIT level is in the range of 2 – 6, and it is **impossible** to have, for example, **no classes** with a DIT of 4.

The *DIT Standard* is different from a simple average. Instead, to calculate this, one must first find the summation of the amount of classes at a certain DIT level (2 or greater) multiplied by their DIT level. That value is then divided by the total amount of all classes with a DIT of 2 or

greater, and the result is the *DIT Standard*. This calculation can be represented using the following formula:

- For a system with DIT levels in the range of 2,...,*n*
- Let *a* be the number of classes at the current DIT level in the range
- Let *x* be the sum of all classes with a DIT of 2 or more
- *DIT Standard* = $\frac{\sum_{i=2}^{n} a * i}{x}$

System	Version
Ant	1.8.4
ANTLR	4.0
ArgoUML	0.34
Azureus	4.8.1.2
FreeCol	0.9.5
Hibernate	4.2.2
JHotDraw	6.0.1
JUNG	2.0.1
Junit	4.9
Lucene	4.2.1
Weka	3.7.9

Table 4.2 DIT Standard for analyzed systems

Using this formula, we can see the *DIT Standard* for each of the analyzed systems in *Table 4.2*. If these values were to be rounded up to the nearest whole number, there would be 9 of the 11 systems who have a *DIT Standard* of 3. Thinking back on the suggestions for DIT values being no higher than 3-5, it seems that all of the systems are able to maintain lower depths, for the most part. A few systems do have values higher than this, and while that is not necessarily an immediate problem, those classes that have these lower DIT levels will be easier to maintain and understand. This might also suggest that these systems are making good use of inheritance, while keeping the complexity of a majority of their classes to a minimum.

Because there is no accepted number for a “bad” DIT, the analysis for this thesis will simply use a value of 4 as an “acceptable” DIT. This was chosen as it is the middle number from the 3-5 range suggested during *Chapter 2 Section 2*. This value can be scaled to fit any other systems based on the discretion of the researcher. With that in mind, a DIT greater than 4 can be considered to be dangerous because of the increased complexity and a possibility for more defects. This was shown in the results of an empirical analysis by Subramanyam and Krishnan on how the Chidamber and Kemerer metrics relate to the number of defects in any given system. Their results confirmed their hypothesis that classes with a higher DIT are associated with higher defects (Subramanyam & Krishnan, 2003, p. 307). To determine how dangerous our analyzed systems are, we will take the percentage of classes with an acceptable DIT (less than

or equal to 4), and compare that to the percentage of classes with a dangerous level of DIT (5 or more). These results are shown in *Table 4.3*.

System	% of Acceptable DIT	% of Dangerous DIT	Classes
Ant	96.23%	3.77%	1510
ANTLR	95.25%	4.75%	632
ArgoUML	96.55%	3.45%	2522
Azureus	99.34%	0.66%	3783
FreeCol	97.78%	2.22%	1173
Hibernate	97.78%	2.22%	7037
JHotDraw	100.00%	0.00%	207
JUNG	99.76%	0.24%	850
JUnit	100.00%	0.00%	250
Lucene	98.94%	1.06%	4349
Weka	99.54%	0.46%	1724

Table 4.3 Acceptable and dangerous DIT levels

Each of the 11 systems were able to maintain acceptable DIT levels for over 95% of all their classes. Based solely on this information, it would appear that these systems should have a low number of defects. Many of the systems were able to contain a high number of classes and still maintain a very high percentage of acceptable DIT levels, such as **Azureus** with 99% of 3,783, **Lucene** with almost 99% of 4,349, and even **Hibernate** with the most amount of classes and almost 98% of those classes being acceptable.

In contrast to this, there is **ANTLR**, who has the third lowest amount of classes at 632, but the highest percentage of dangerous DIT levels at 4.75%. This would suggest that **ANTLR** is creating hierarchy trees that are much too deep, and perhaps they should consider refactoring some of the subclasses in these trees into separate classes. Doing so would result in higher ranges for the number of children, so this must also be considered before any major restructuring of code.

Only 2 of the systems had a 100% acceptable rate in their DIT levels: **JHotDraw** and **Junit**. Looking back to *Table 4*, **JUnit** only has a 28% of inheritance use from its classes. On the other hand, **JHotDraw** makes much greater use of inheritance, at 53% of its classes taking advantage of inheritance, making **JHotDraw** among the top 3 of 11 inheritance using systems. This solidifies the fact that **JHotDraw** is a prime example to use for examining inheritance use.

To end the analysis on DIT ranges for the systems, *Figure 4.2* shows the number of classes at a specific DIT value for each system. As mentioned in the introduction to this chapter, each system has at least 44% of its classes that have a DIT of 1 (no inheritance). Because of that, those classes without inheritance are not included in this chart.

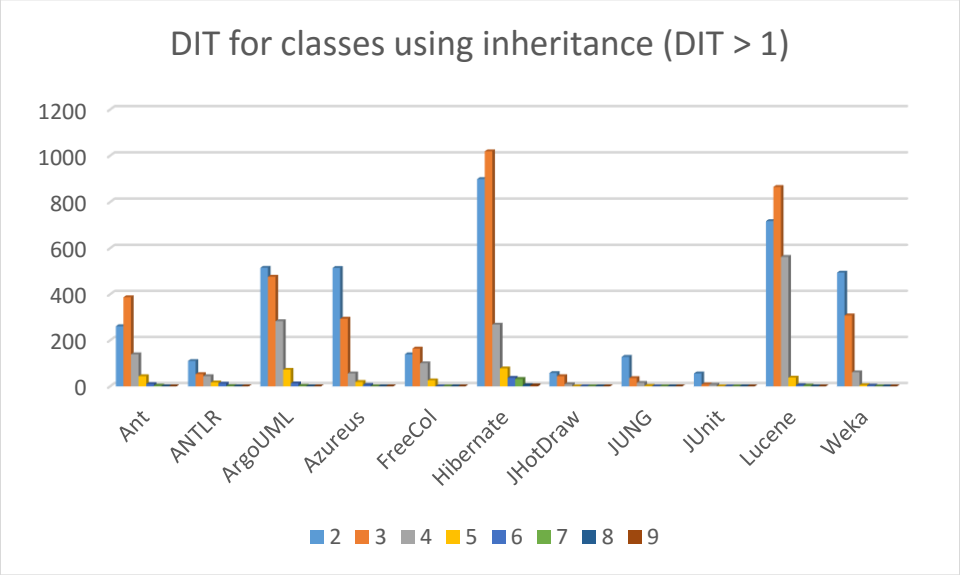

Figure 4.2 DIT ranges for inheritance use

4.1.2 NOC and WMC

The number of children and the weighted method count for a class, when reviewed together, can provide an interesting synopsis on the complexity of that class. According to Marinescu, high values for NOC and WMC together in a class indicate high complexity, and he claims that they are "potentially influencing all the classes derived from them and therefore it is not at all desirable to have such classes in a project" (Marinescu, 1998). If at all possible, these classes should be restructured to fit better design practices. Analysis on the 11 proposed system was done to determine how many classes should potentially be refactored based on their NOC and WMC values. This will be referred to as the *Inherited Method Risk*. To find this, a value must first be selected to use as the limit for NOC, as well as one for the limit on WMC.

The selected value for WMC is found by comparing values for WMC in the 11 proposed systems against values found in the research by Marinescu (Marinescu, 1998, p. 2). It should be noted that Marinescu is using McCabe's Cyclomatic Complexity Metric (McCabe, 1976), in addition to

just finding the number of methods in a class, which is the way the **Inheritance Inquiry** tool finds the WMC. The average of the values from these 2 datasets was found, and then those 2 averages were again averaged. These values and their averages are seen in *Table 4.4* and *Table 4.5*. From these WMC values, our limit for WMC is **21**.

System	WMC Average
Ant	17.5
ANTLR	6
ArgoUML	24
Azureus	20.5
FreeCol	9.3
Hibernate	53.8
JHotDraw	4.6
JUNG	10.4
JUnit	4.7
Lucene	45.9
Weka	15.3
Total Average	**19.3**

Table 4.4 WMC average for McNealy research

Site	Complexity Definition	WMC Average
Site A	McCabe	21
Site A	Unitary	13
Site B	McCabe	21
Site B	Unitary	10
Site C	McCabe	56
Site C	Unitary	16
Total Average		**22.8**

Table 4.5 WMC average for Marinescu research

There is no definitive value to use in determining a "best" amount for NOC. While having classes with a large number of children is indicative of high reuse of a class (Chhikara & Chhillar, 2012, p. 365; Chidamber & Kemerer, 1994, p. 485), it may also be a sign that the complexity of that class and its children is too great, and should be restructured (Marinescu, 1998, p. 3; Aviosto, 1997) . Because there is no standard for NOC, the average number of children for each of the proposed systems will be taken and used as the limit for the *Inherited Method Risk*. This average is **10.3**, as shown by the values in *Table 4.6*.

System	NOC Average
Ant	8.1
ANTLR	8.3
ArgoUML	13.2
Azureus	11.6
FreeCol	26.3
Hibernate	5.6
JHotDraw	3.7
JUNG	7.5
JUnit	5.3
Lucene	13.9
Weka	10.3
Total Average	**10.3**

Table 4.6 NOC Averages for analyzed systems

With the values select for limits on NOC and WMC, the **Inheritance Inquiry** tool can be run for analysis on the systems, and when the *AnalyzerService* is calculating the NOC with the *analyzeForNoc(JavaProject)* method, any calculated hierarchy trees that have **10** children and

21 inheritable methods (methods that are not **private**), will be flagged as a risk. The results of this can be seen in *Table 4.7* and *Figure 4.3*.

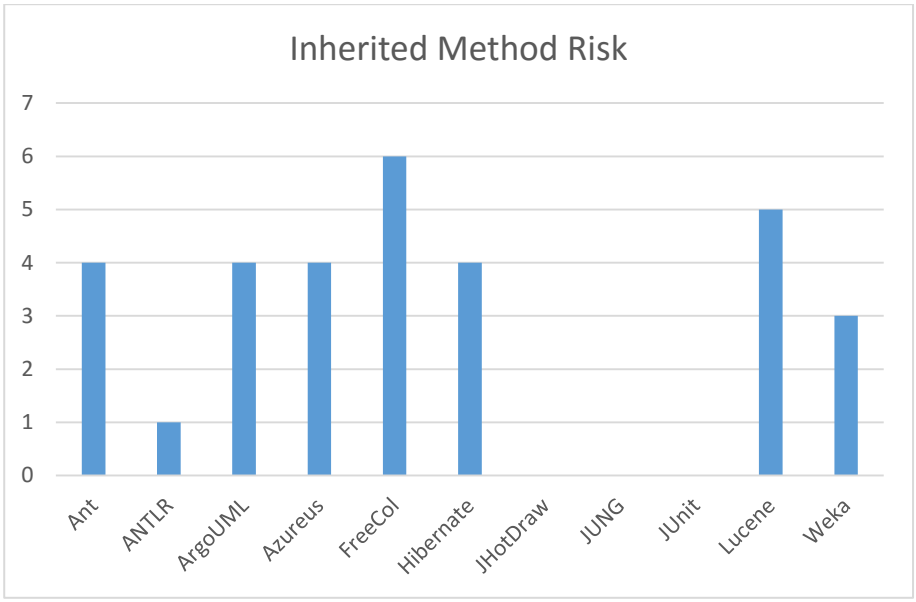

Figure 4.3 Inherited Method Risk

System	Inherited Method Risk
Ant	4
ANTLR	1
ArgoUML	4
Azureus	4
FreeCol	6
Hibernate	4
JHotDraw	0
JUNG	0
JUnit	0
Lucene	5
Weka	3

Table 4.7 Inherited Method Risk

As a side note, it is important to realize that these values are not indicative of an absolute standard to be used for all research purposes. The **Inheritance Inquiry** tool is designed to be flexible with the values for NOC and WMC when calculating the *Inherited Method Risk*, and these values can be adjusted based on the judgement of the user in regards to the scale of whatever system is being analyzed.

JHotDraw, **JUNG**, and **Junit** are the only systems who seem to have no risk of increased complexity through the inherited methods of their classes. These systems also had the highest percentage of acceptable DIT ranges, with **JHotDraw** and **Junit** at 100% and **JUNG** at 99.76%.

FreeCol has the highest amount of risky classes, even though it has a low percentage of inheritance use throughout the entire system, at only 36.57% of classes using inheritance. Upon closer inspection of one of the offending classes, *FreeColObject*, more details will be revealed about how there may be a risk to any classes that are inheriting from *FreeColObject*, either by directly extending the class, or by inheriting from further down the hierarchy. *Table 4.8* shows the immediate subclasses of *FreeColObject*, along with information on their method counts. *FreeColObject* contains 50 inheritable methods, meaning those methods are either **protected** or

public, and therefore accessible by any instance of an object that inherits from *FreeColObject*. The WMC column shows the count of methods that are defined in the child class. The WMC + Inheritable Methods column displays the count of those methods plus the count of the methods inherited from *FreeColObject*. Of the 18 direct subclasses of *FreeColObject*, 8 of those classes have a WMC higher than the WMC limit of **21** that was used to find the *Inherited Method Risk*.

FreeColObject		
Children	**WMC**	**WMC + Inheritable Methods**
AIObject	7	57
AIMain	19	69
AbstractOption	7	57
AbstractUnit	10	60
NationOptions	13	63
ExportData	11	61
ModelMessage	21	71
HistoryEvent	10	60
Feature	20	70
MarketData	24	74
FreeColGameObjectType	16	66
Scope	20	70
UnitTypeChange	12	62
AbstractGoods	8	58
HighScore	30	80
TradeItem	11	61
FreeColGameObject	30	80
DiplomaticTrade	20	70

Table 4.8 FreeColObject.java immediate subclasses and their WMC

The developers of **FreeCol** might need to consider restructuring this class into more **abstract** subclasses, or find methods that could be refactored into only those classes that rely on them, or removed altogether. However, the name for *FreeColObject* implies that this class has a similar purpose as *java.lang.Object*, in that many objects in the **FreeCol** system should be an instance of *FreColObject*, by inheriting from it in some way. After examining the full hierarchy tree, shown in *Figure 4.4*, it is apparent that many other objects in the **FreeCol** system *do* inherit from this class. While the graph is skewed because of the large size, the depth and width of the tree are both large in size. This only increases the amount of methods that each subsequent child inherits, further increasing the complexity of each subclass.

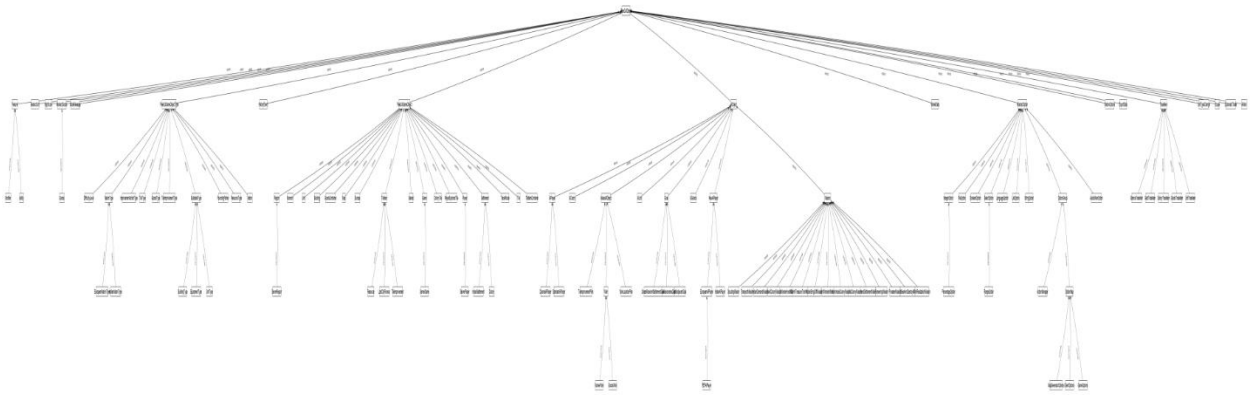

Figure 4.4 FreeCol.java inheritance hierarchy

4.1.3 Inheritance Hierarchy

The width of the inheritance hierarchies in a system can be examined to provide insight into how often a system practiced code reusability. The number of a children at a certain level in the hierarchy are used to determine the width of that level. As each level in the hierarchy advances deeper, the width of those levels tends to grow. This is because a higher number of children provides more opportunities for subclassing of those children. The data for the inheritance hierarchies of each of the analyzed systems is presented in *Table 4.9*. Also included in this table is the NOC average and max for each of these systems.

System	NOC Average	Average Width	Max NOC	Max Width	Max width class
Ant	8.1	11	157	175	junit.framework.TestCase
ANTLR	8.3	8	33	35	org.antlr.v4.codegen.model.OutputModelObject
ArgoUML	13.2	10	91	91	java.util.Observable
Azureus	11.6	9	172	172	com.aelitis.azureus.ui.common.table.impl.TableColumnImpl
FreeCol	26.3	11	49	52	net.sf.freecol.common.model.FreeColObject
Hibernate	5.6	4	394	501	org.hibernate.testing.junit4.BaseUnitTestCase
JHotDraw	3.7	3	12	14	CH.ifa.draw.handle.AbstractHandle
JUNG	7.5	2	28	28	javax.swing.JApplet
JUnit	5.3	2	7	7	org.junit.runners.model.RunnerBuilder
Lucene	13.9	8	437	437	org.apache.lucene.util.LuceneTestCase
Weka	10.3	5	113	163	junit.framework.TestCase

Table 4.9 Inheritance width ranges for analyzed systems

The NOC average and average width are similar only for 3 systems: **ANTLR**, **Hibernate**, and **JHotDraw**. Around half of the max NOC values are the same as the max hierarchy width:

ArgoUML, **Azureus**, **JUNG**, **JUnit**, and **Lucene**. At first thought, it may seem suspicious that the NOC and width do not match more closely. However, it must be remembered that the NOC calculates only the **immediate** subclasses of an element, while the width will take the entire class hierarchy structure into account when determining the size. This means that the width can be a sum of all the subclasses at a certain level in the hierarchy, thus allowing the width to occasionally be higher than the NOC. The range of values for the NOC will also be much broader, as there will be more instances of low counts of subclasses. This explains why the NOC has a higher average than the hierarchy width.

The distinction between the NOC and width can be further visualized by the graphs displayed in *Figure 4.5*. On the left, the hierarchy tree for the class *AbstractHandle* from **JHotDraw** is displayed, and the right side shows the hierarchy tree for *LocatorHandle*. The max width of *AbstractHandle* is **14**, because it includes the subclasses of both *LocatorHandle* and *ChangeConnectionHandle*. However, the NOC of *AbstractHandle* is only **7**, as that is the number of **immediate** subclasses. *LocatorHandle* has a lower hierarchy width, at **12**, but its NOC is higher, with a value of **12**. This explains why a system is able to have a very wide hierarchy tree when its max NOC may not be quite as high (see **Hibernate** in *Table 4.9*).

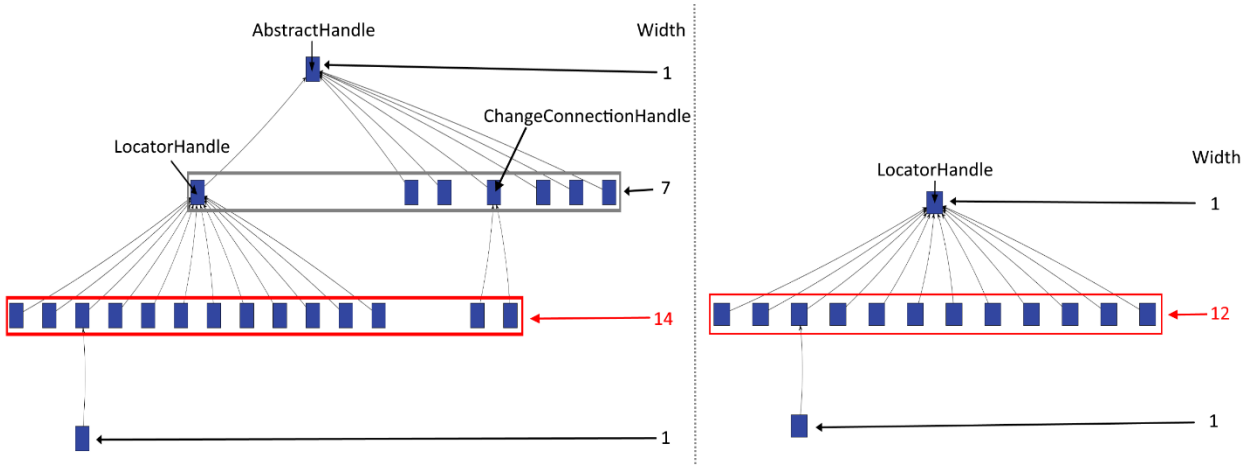

Figure 4.5 AbstractHandle.java and LocatorHandle.java widths

Two of the systems, **Ant** and **Weka**, have *junit.framework.TestCase* as the class which has the highest hierarchy tree width. This class comes from the **JUnit** system. Looking again at *Table 4.9*, **JUnit** only has a max width of **7**, while **Ant** has **175** and **Weka** has **163**. The reason that the

width is so high in **Ant** and **Weka**, and low in **JUnit**, is that the class *TestCase* is an **abstract** class that is meant to be subclassed in an application's test suite and used as the base class for any tests. **Ant** and **Weka** must therefore both have a very large test suite of classes that inherit from *TestCase*. This is an instance when having a large number of children with a deep inheritance tree is acceptable, as much of the code in a test suite is repeatable, and so having a significant amount of reusability is preferable.

In the same way that **Ant** and **Weka** have a class from an external library as the one with the largest hierarchy tree, **ArgoUML** and **JUNG** also inherit from another system's class at the top level of their highest hierarchy. The classes that these systems inherit are both from official Java APIs, with **ArgoUML** inheriting from *java.util.Observable*, and **JUNG** making use of *javax.swing.JApplet*. This might suggest that the most efficient class to use is already available to an application from the programming language that is used for development. The programmer of that system can take advantage of this through inheritance, and avoid having to write code for a feature that is already being widely used.

4.1.4 Effects on Methods

When a class inherits from another, certain methods defined in the parent class become available to the child. The subclass can choose to leave these methods as they are defined in the parent, or they can **override** the methods to provide a new implementation. Overriding can be helpful when a developer needs to have a class of a certain type, but unique behavior for a method that already has been defined. Looking back on *Section 1.5* in the discussion on *Inheritance in Java*, the *draw()* method for a *RadiusHandle* in the **JHotDraw** system is overriding the default *draw()* method from *AbstractHandle* in order to give *RadiusHandle* its unique circular shape. This is an example of when overriding an already implemented method is necessary.

System	Overridden Method Average	Overridden Method Max	Class with Max Overridden Methods
Ant	1	29	org.apache.tools.ant.taskdefs.Delete
ANTLR	2	11	org.antlr.v4.automata.LexerATNFactory
ArgoUML	2	95	org.argouml.model.mdr.UndoCoreHelperDecorator
Azureus	2	38	com.aelitis.azureus.core.networkmanager .admin.impl.NetworkAdminImpl
FreeCol	2	17	net.sf.freecol.server.ai.ColonialAIPlayer
Hibernate	2	51	org.hibernate.dialect.PostgreSQL81Dialect
JHotDraw	3	20	CH.ifa.draw.figure.DecoratorFigure
JUNG	2	6	edu.uci.ics.jung.visualization.control.EditingModalGraphMouse
Junit	1	5	org.junit.experimental.theories.Theories
Lucene	2	28	org.apache.lucene.index.IndexWriterConfig
Weka	3	16	weka.experiment.RemoteExperiment

Table 4.10 Overridden Methods for analyzed systems

Information about how each analyzed system overrides methods in classes using inheritance is shown in *Table 4.10*. The max amount of times that methods are overridden varies greatly between each system, with no real pattern emerging. However, when examining the average, 64% of the systems had an average of 2 overridden methods for each class. The reason for this is most likely explained as those classes overriding the same 2 methods: *equals()* and *hashCode()*, both from *java.util.Object*. As noted by William Pugh and David Hovemeyer, when overriding the *equals()* method, it is important to also override the *hashCode()* method. This is because the two methods together form a contract that determines whether objects in Java have equal values, and objects that violate this contract will not work in hash-based collections (Pugh & Hovemeyer, 2004). And since these methods are defined in *java.lang.Object*, even those classes who have a DIT of 1 (only inheriting from *Object*) will often be required to override them.

DecoratorFigure will be further examined to provide more insight into the ways a class might override methods from its parent. *DecoratorFigure* is an **abstract** class that inherits from *AbstractFigure*, and has a total of **29** methods, with **20** overridden methods. Of those 20 methods, **17** are overriding existing implementations defined in *AbstractFigure*, and the remaining **3** are concrete implementations for **abstract** methods from *AbstractFigure*. The reason that *DecoratorFigure* has such a high count for overriding methods is because this object is employing the **Decorator** design pattern. This pattern is used to dynamically add extra features

to an object, and to “provide a flexible alternative to subclassing for extending functionality” (Gamma, et al., 1995, p. 175). *Figures* that inherit from *DecoratorFigure* will take a separate concrete *Figure,* and delegate the execution of the *AbstractFigure* overridden methods to the *Figure* that is to be decorated. As shown by *Code Snippet 4.1*, the *fComponent* is the *Figure* to be decorated, and *DecoratorFigure* delegates the overridden *displayBox()* method to the implementation provided by whatever the concrete class of *fComponent* happens to be.

```
public abstract class DecoratorFigure
                extends AbstractFigure
                implements FigureChangeListener {

       /**
        * The decorated figure.
        */
       protected Figure fComponent;

       /**
        * Forwards displayBox to its contained figure.
        */
       @Override
       public Rectangle displayBox() {
           return fComponent.displayBox();
       }
}
```

Code Snippet 4.1 DecoratorFigure.java in JHotDraw

The ways in which the analyzed systems are extending methods is also examined in this thesis. Extending a method is when a subclass overrides a method from its superclass in order to provide a new implementation, but also still calls the superclass method through the use of the **super** keyword. This is beneficial by allowing the subclass to provide additional functionality to a method, while still using the default base class implementation. *Table 4.11* shows the details of how extended methods are used in the analyzed systems.

System	Extended Method Average	Extended Method Max	Class with Max Extended Methods
Ant	2	29	org.apache.tools.ant.taskdefs.Delete
ANTLR	1	2	org.antlr.v4.runtime.atn.NotSetTransition
ArgoUML	1	75	org.argouml.model.mdr.UndoCoreHelperDecorator
Azureus	1	14	org.gudy.azureus2.ui.swt.views.table.impl.TableViewSWTImpl
FreeCol	1	4	net.sf.freecol.common.model.Modifier
Hibernate	1	51	org.hibernate.mapping.Subclass
JHotDraw	2	5	CH.ifa.draw.figure.connection.LineConnection
JUNG	1	4	edu.uci.ics.jung.graph.ObservableGraph
Junit	1	3	junit.extensions.RepeatedTest
Lucene	2	16	org.apache.lucene.store.BaseDirectoryWrapper
Weka	2	19	weka.experiment.RemoteExperiment

Table 4.11 Extended Methods for analyzed systems

The average for extended method use is less than the average for overriding methods, showing that extending methods is not quite as necessary for inheritance as overriding. This is mainly due to the fact that overridden methods do not **need** to include a call to the **super** method, but to extend a method, the class must override the method **and** call the parent implementation. Many classes need to override *equals()* and *hashCode()*, and while these methods can be extended by making use of the superclass implementation to include attributes from the parent, they are not always required to do so.

To further examine how one of these systems is extending methods, the **final** class *Modifier* from the **FreeCol** system is shown in *Code Snippet 4.2*. In the *writeAttribues()* method, the **super** implementation is first called, and then additional operations are performed on the *XMLStreamWriter* in order to write out attributes that are unique to the *Modifier* class.

```
public final class Modifier extends Feature {

 public void writeAttributes(XMLStreamWriter out) throws XMLStreamException {
    super.writeAttributes(out);
    out.writeAttribute("value", String.valueOf(value));
    out.writeAttribute("type", type.toString());
    if (incrementType != null) {
      out.writeAttribute("incrementType", incrementType.toString());
      out.writeAttribute("increment", String.valueOf(increment));
    }
  }
}
```

Code Snippet 4.2 Modifier.java in FreeCol

Figure 4.6 gives an overview of how often methods are overridden in each of the analyzed systems, on average. According to Mark Lorenz and Jeff Kidd, "(a) large number of overridden methods indicates a design problem." (Lorenz & Kidd, 1994, p. 67) They suggest that subclasses should be used as a specialized type of the superclass, and that these subclasses should implement methods that are unique to the intended purpose of that class. In order to determine if subclass method overrides are being used safely in a system, Lorenz and Kidd have proposed a threshold for the amount of times this should be acceptable. For those methods that are **not meant to be overridden** (for Java applications, this means methods that are **not** declared as **abstract** in the superclass) the tolerable number of times for overrides in a subclass is three or less (Lorenz & Kidd, 1994, p. 68). For any subclass with a high DIT, this amount should be even fewer. As can be seen in *Figure 4.6*, almost all of the systems acknowledge this rule. **JUnit** is just over the edge of the acceptable amount, with an average of 3.47, but **ANTLR** and **JUNG** both exceed this threshold by 2, with averages of 5.18 and 5.78, respectively. Both of these systems should inspect their code for overridden methods, and consider refactoring instances that might be unnecessary, or better suited to a separate, unique method.

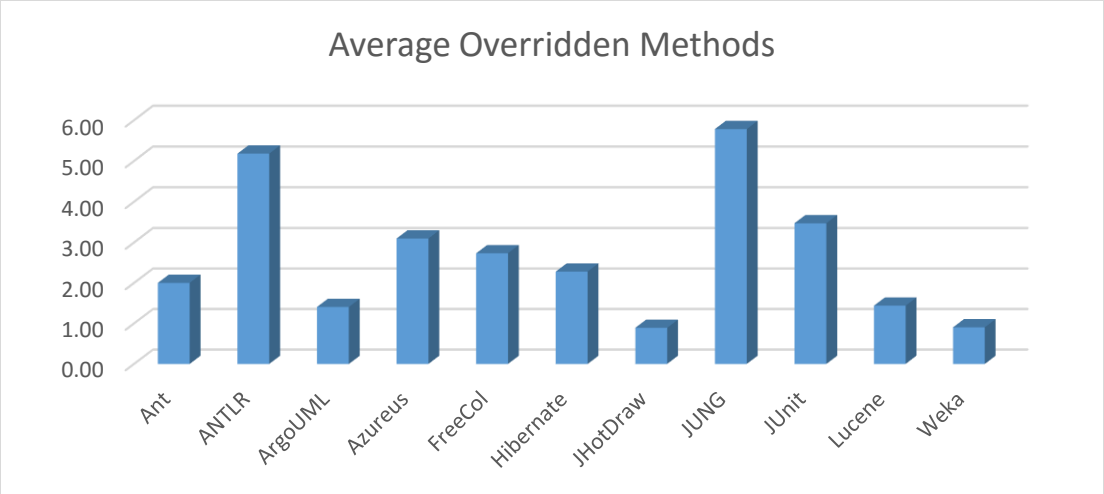

Figure 4.6 Average overridden methods in analyzed systems

4.2 Individual System Investigation

This section will provide a brief overview for each of the individual systems that were analyzed. This mostly includes the values for the DIT and NOC, with graphs to detail those metrics. A few of the systems contained more interesting results than the others, and will therefore be given a closer inspection.

4.2.1 Ant

Of all the analyzed systems, **Ant** has the highest percentage of inheritance use, with 55.89% of the 1510 classes inheriting from another class. The average DIT value is 3.01, and the deepest inheritance tree is 7, for 3 different classes. The average value for NOC is 8.3, with the highest NOC being 157 for the class *BuildFileTest*. According to the JavaDocs for this class, the main purpose of this class is to provide utility methods to all subclasses that inherit from the class. As was mentioned with **JUnit**, this is not a very efficient way of building tests, so using inheritance in this way is not ideal. The recent developers of **Ant** seem to agree with this. The version of **Ant** that was analyzed for this thesis is 1.8.4, and *BuildFileTest* has been deprecated as of version 1.9.4, and replaced with a new class and annotations for current testing uses.

Ant seems to be a very good example of inheritance use, as it has the highest overall percentage of use, and the ranges of depth and number of children are spread evenly throughout the system. And compared to some other inheritance heavy systems, the number

of overridden and extended methods is not excessive, and seems to be within a reaonable amount.

4.2.2 ANTLR

ANTLR has an average percentage of inheritance use, with 37.5% of its 632 classes using inheritance in some way. The average value for DIT is 3.03, and the class with the highest DIT is *LL1OptionalBlock* with a DIT of 7. For NOC, the average is 8.1, with the class *BaseTest* having the highest number of children at 33. **ANTLR** has one of the second lowest usage of method overrides at 122, and the lowest amount of extended methods at 12.

4.2.3 ArgoUML

At 53.97% of 2522 classes using inheritance, **ArgoUML** is the second most inheritance heavy system. With an average DIT of 2.97, most of the inheritance hierarchies stay below 3, with only 2 classes having a maximum depth of 7. For the NOC, the ranges are spread out more than the previous 2 systems, with **ArgoUML** having an average NOC value of 13.2, and *CrUML* having the most children at 91.

4.2.4 Azureus

The **Azureus** system has the second lowest percentage of inheritance use, at only 23.5% of its 3783 classes. The DIT average is 2.55, and this system manages to keep the hierarchy depths below 7, with the maximum being 6 for 6 classes. The NOC is leaning towards a high value when compared with other systems, with an average of 11.6 and a maximum of 91 for the class *CoreTableColumn*.

Considering that this system has the third highest number of classes, this may be an indication that the developers of **Azureus** are not taking full advantage of the benefits provided by inheritance. This system also has a very high amount of method overrides at 1221 and extensions at 288, being among the median for inherited method usage in the analyzed systems. This is questionable when accounting for the low average inheritance usage, as this means that many of these method overrides and extensions are occurring in within a small subset of the system's classes. However, while the version of **Azureus** that was analyzed is the latest available from the **Qualitas Corpus** (4.8.1.2), the system **Azureus** is now available under a

new name, **Vuze** (Azureus Software, Inc, 2016), and is on version 5.7.1.0. The current version may have included refactoring, and hierarchy restructuring during the upgrade process.

4.2.5 FreeCol

FreeCol is near the average percentage for the systems in terms of inheritance use, at 36.57% of the 1173 classes using inheritance. Along with **ANTLR**, **FreeCol** has the highest average for DIT at 3.03. This system also manages to avoid creating hierarchies that are unnecessarily deep, with 26 classes having the max DIT of 5. The average number of children for **FreeCol** is also on the low side, at only 5.6, and the highest NOC being 49 for the class *FreeColTestCase*.

4.2.6 Hibernate

Hibernate is by far the largest system that was analyzed, with 7037 classes, and 2342 of those classes using inheritance. Despite this, the percentage of overall inheritance use is only 33.28%, given this system a low ratio of class to inheritance usage when compared to the other systems. While the average range for DIT values was just barely under three, at 2.92, **Hibernate** has the deepest hierarchy trees of any system, being the only one to have a DIT of 8 (with 5 classes) and a DIT of 9 (with 4). The average NOC was also the largest for **Hibernate** at 26.3, which is almost double the second highest average of 13.9 for **Lucene**. The class *BaseCoreFunctionalTestCase* has a very high number of children, at 394; however, this is not the system with the highest NOC, which will be mentioned shortly.

Hibernate is an object-relational mapping (ORM) framework, which means it is used to generate and manage Java objects based off of many different SQL dialects. Because of this, the configurations for each of the different dialects can be quite extensive, and thus span multiple classes, building a large inheritance hierarchy as those configurations grow in scope. These dialects are all based of the *Dialect* class, which is an abstract class with a DIT of 1. While this class does not have an overly excessive number of children, at only 22, it was flagged as an *Inherited Method Risk* by the **Inheritance Inquiry** tool. As mentioned in *Section 4.1.3*, for the purposes of this thesis an *Inherited Method Risk* is when a class has **10** children and **21** inheritable methods, and means that there is a high risk of complexity for any children that inherit from *Dialect*. As seen in *Figure 4.7*, there are many subclasses of *Dialect*, and the depth of the hierarchy is deep, reaching 6 in one class.

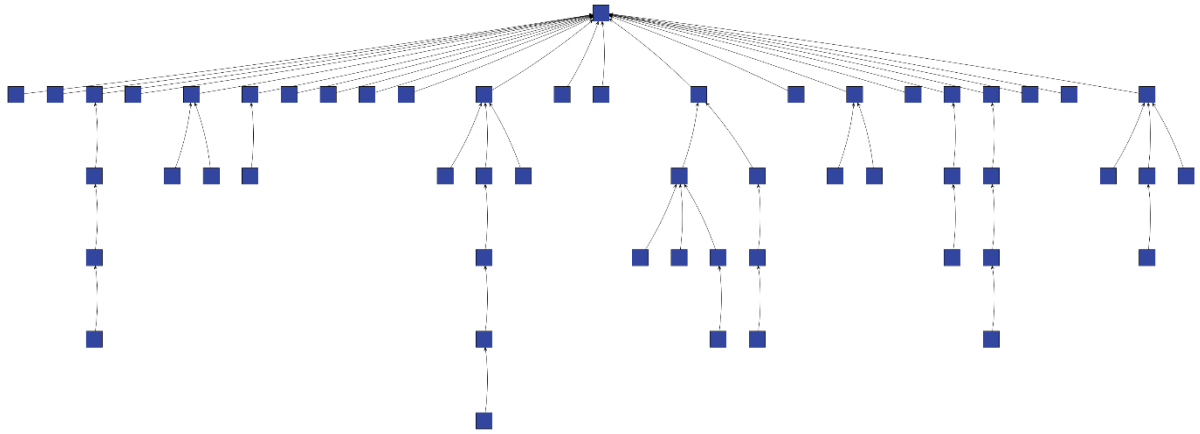

Figure 4.7 Dialect.java inheritance hierarchy from Hibernate

The version of **Hibernate** that was analyzed here is the latest available in the **Qualitas Corpus**, at 4.2.2, and the latest stable release is version 5.2.1. Unless updates have been made to the structure of this hierarchy in the time between these releases, this version's method of handling the dialects may be the most efficient way possible, and in that case would be acceptable.

4.2.7 JHotDraw

Of all the analyzed systems, **JHotDraw** has the lowest amount of classes, at 207. However, with a percentage rate of 53.62%, it is one of the most inheritance heavy systems. **JHotDraw** manages to keep the depth of all of its inheritance trees under 5, with 9 classes having a DIT of 4, and an average range of 2.56. The average NOC is lowest in **JHotDraw**, at 3.7, and the maximum child count is 12 for *LocatorHandle* (which was examined in *Section 4.1.4*). As previously mentioned in *Section 4.1.1*, **JHotDraw** is an excellent illustration of how inheritance should be used, and this is the reason many of the examples in earlier chapters are based off this system.

4.2.8 JUNG

With only 21.29% of its 850 classes, **JUNG** has the lowest percentage of inheritance use out of all of the analyzed systems. The DIT average is 2.4, and the maximum level of depth is 5, for 2 classes. The average NOC is 7.5, with the highest number of children being 28. However, the class that has the highest number of children is from a Java API, *javax.swing.JApplet*. Considering that this system such a low use of inheritance, it would possibly make more sense for the subclasses of *JApplet* to be placed into separate hierarchies. Looking at the list of these subclasses in *Table 4.12*, a possibility for this could be one abstract class for *LayoutDemo*, one for *ShaperDemo*, and maybe another for any standard Demo sample

Subclasses of JApplet in JUNG
edu.uci.ics.jung.samples.EdgeLabelDemo
edu.uci.ics.jung.samples.RadialTreeLensDemo
edu.uci.ics.jung.samples.TwoModelDemo
edu.uci.ics.jung.samples.PluggableRendererDemo
edu.uci.ics.jung.samples.TreeLayoutDemo
edu.uci.ics.jung.samples.ClusteringDemo
edu.uci.ics.jung.samples.ImageEdgeLabelDemo
edu.uci.ics.jung.samples.LensDemo
edu.uci.ics.jung.samples.ShowLayouts
edu.uci.ics.jung.samples.MultiViewDemo
edu.uci.ics.jung.samples.PerspectiveTransformerDemo
edu.uci.ics.jung.samples.SubLayoutDemo
edu.uci.ics.jung.samples.HyperbolicVertexImageShaperDemo
edu.uci.ics.jung.samples.LensVertexImageShaperDemo
edu.uci.ics.jung.samples.VertexLabelAsShapeDemo
edu.uci.ics.jung.samples.VertexCollapseDemo
edu.uci.ics.jung.samples.L2RTreeLayoutDemo
edu.uci.ics.jung.samples.BalloonLayoutDemo
edu.uci.ics.jung.samples.AnnotationsDemo
edu.uci.ics.jung.samples.VertexImageShaperDemo
edu.uci.ics.jung.samples.MinimumSpanningTreeDemo
edu.uci.ics.jung.samples.GraphEditorDemo
edu.uci.ics.jung.samples.SatelliteViewDemo
edu.uci.ics.jung.samples.PerspectiveVertexImageShaperDemo
edu.uci.ics.jung.samples.VertexCollapseDemoWithLayouts
edu.uci.ics.jung.samples.VertexLabelPositionDemo
edu.uci.ics.jung.samples.TreeCollapseDemo
edu.uci.ics.jung.samples.WorldMapGraphDemo

Table 4.12 Subclasses of JApplet.java in JUNG

4.2.9 JUnit

JUnit also has a low amount of classes at 250, as well as a low percentage of inheritance use, with only 28.4% of those 250 classes being subclasses. The average for the depth of inheritance tree in **JUnit** is 2.31. Of the 11 analyzed systems, **JUnit** and **JHotDraw** are the only ones whose DIT levels remained under 5 for all classes; **JUnit** has 7 classes with a DIT of 4. The NOC average for this system is 5.3, and the maximum NOC of 7 is the lowest for a max NOC compared to the other systems. As was discussed in *Section 4.1.1*, the metrics for inheritance use in **JUnit** are acceptable, as this system is a testing framework, and should therefore keep inheritance to a minimum.

4.2.10 Lucene

The **Lucene** system is one that makes extensive use of inheritance throughout its classes. This system has the second highest amount of classes at 4349, and uses inheritance in over half of

those classes (50.36%). Like most other systems, the DIT average for **Lucene** was almost 3, at 2.98. The highest value for depth was 7 for 3 classes. **Lucene** has the second highest average for NOC, at 13.9. The class with the highest number of children is *LuceneTestCase*, with 437, which is higher than the max NOC for **Hibernate**.

4.2.11 Weka

The final system, **Weka**, is among the middle range for class count and inheritance use, with 50.52% of 1724 classes using inheritance. **Weka** has an average DIT of 2.52, and manages to keep the depth below 7 for all classes, with only 3 having a DIT of 6. The average for NOC in this system is 10.3. Like **JUNG**, the maximum for NOC comes from a Java API class, *javax.swing.JPanel*, which is inherited from 113 times. The same suggestion that was given for **JUNG** can apply here: each GUI element that is a subclass of *JPanel* should be reevaluated to determine if there is a more efficient class that could hold the common behavior and properties of these classes.

Chapter 5: Conclusion and Future Developments

As inheritance is such a significant aspect of object-oriented programming, it is important that a developer programs with both the benefits and side effects of inheritance in mind. The benefits that come from the use of inheritance include increased code reusability, type substitution between classes that share a parent, and advanced relationship models for classes with similar behavior. While these are important features that aid the development process and can contribute to better code, a developer should not use inheritance simply because it is available. They should consider the effects it may have on existing code, such as how complex the code will be to understand, and if it will become harder to maintain.

Alternatives to inheritance can also be considered, such as using object composition to pass a reference to a class (or interface) of a certain type, and then delegate actions to that class. This would also promote a more interface-based approach, as developers could program their classes around a certain interface instead of a parent class, allowing the composite object to be of any type, as long as that type implements the required interface. This is known as *programming to an interface*, and requires that clients do not need to have any knowledge of specific types being used, as long as those classes implement the correct interface (Gamma, et al., 1995).

This chapter will review the conclusions gained from the results of using the **Inheritance Inquiry** tool to analysis the **Qualitas Corpus** and **JHotDraw** systems. The future of this tool, including improvements that might be made and further development plans, will also be discussed.

5.1 Analysis Results

After completing the analysis on each of the proposed systems, some conclusions may be drawn on how inheritance was used throughout those applications. As proposed in *Chapter 1*, there were four main questions to consider:

- Is there any risk or additional complexity for objects that are deep in an inheritance hierarchy?

- What is the correlation between an object that is inherited many times, and the methods that are implemented in that object?
- How does the hierarchy structure of one object change and grow as more separate objects inherit from the parent?
- What happens to methods that are defined in a parent object, when they are also implemented in child objects?

There is no defined "best" value for DIT, only suggestions that can be given based off the scale of the classes that make up the inheritance hierarchy. As was found through the analysis of this thesis, most of the classes in each of the systems maintained a DIT around 3, with the highest *DIT Standard* being 3.03, and the lowest 2.31. For about a third of the systems, the maximum DIT for their classes stopped at 7, with only **Hibernate** going above that, to 9.

To determine the dangers that might be associated between classes that have a high NOC and WMC, the *Inherited Method Risk* was calculated. This metric can be used to tell a developer that they should perhaps consider refactoring a class to move common behavior between certain subclasses into more appropriate classes. The analysis by the **Inheritance Inquiry** tool showed that more than half of the systems had four or more classes that could be seen as a complex risk, and that are candidates for refactoring.

As more classes inherit from one parent class, the hierarchy structure for that parent grows. With each new subclass, there is a chance that those subclasses will also be inherited from, which expands the hierarchy tree of the parent. This tree includes not only the immediate children of the parent, but also any subclasses of those children. The results that were provided by the analysis of this system show that the class with the largest hierarchy is sometimes a class from a separate library or API, as was the case with **Ant** and **Weka** having large hierarchies spanning from a **JUnit** test class, and **ArgoUML** and **JUNG** using Java APIs. This suggests that a more convenient class to inherit from might already be developed, and programmers should take advantage of that existing code.

The final section of analysis was performed on inherited methods, and how subclasses might override or extend methods that are defined in their parent class. Methods can be overridden

for a number of reasons, such as when the default implementation is not quite right for a certain subclass, or when a particular design pattern needs to be employed (as was the case for *DecoratorFigure* in **JHotDraw**). Methods can also be extended to provide additional functionality, while still calling the parent's implementation. Having a large amount of method overrides can reveal design issues with an inheritance hierarchy, and may demonstrate that a class hierarchy should be restructured. The analysis done on the 11 proposed systems found that all but 2 systems were within an acceptable threshold regarding their method overrides. This indicates that the majority of these systems were defining proper methods for their subclasses, and using overrides sparingly.

5.2 Future Developments

There are a few improvements that might be made to increase the efficiency and appeal in using the **Inheritance Inquiry** tool. At the time of the research for this thesis, the tool is possibly more complex than was necessary for the project. There are a number of different Tasks in the tool that were deprecated by the time a majority of the analysis was being performed. Some of the Tasks that were not used include a Task that targets different versions of a system, and tasks that were intended to only calculate the metrics or to build graphs. These Tasks were ignored in favor of either a Task that analyzed everything from all systems at once, or a Task that built only a single system for analysis. There was also a service that could be used for download projects through **Git**, and this was ignored in favor of simply focusing on the **Qualitas Corpus** systems and **JHotDraw**.

Initially, the tool was developed with the intention of being published to a website, where any developer would be able to upload their project and have it analyzed. While some development time was spent on this, the website was eventually scaled down to the code base that is the current state of the **Inheritance Inquiry** tool. The tool is now a library that can be imported into any application and used in conjunction with the existing code of that program. There are some improvements needed on the API for the library, and that refactoring will be the next step in the development process of the tool. The API is already open to include listeners and multiple services that will allow for easy integration with a graphical based application, and

development on a more user friendly application for the **Inheritance Inquiry** tool has already been started.

5.3 Closing Thoughts

Inheritance is a powerful feature of object-oriented programming that can be used to develop flexible and reusable code. There are many benefits to be gained from utilizing this concept, but developers have to be mindful of the proper times to use inheritance, and when they should consider alternatives. There are instances when a large class hierarchy is necessary for a feature to function properly, and the architecture of an application can be designed around the concept of inheritance, as is the case with many graphical based applications that have similar components. Inheritance is fundamentally designed around the concept of allowing existing programs to be expanded and improved without the need to modify existing code (Taivalsaari, 1996, p. 474). By remaining mindful of this fact and the benefits and dangers of its use, inheritance can be used in conjunction with other programming concepts and design practices to produce more efficient code.

References

Aviosto, 1997. *Chidamber & Kemerer object-oriented metrics suite.* [Online]
Available at: http://www.aivosto.com/project/help/pm-oo-ck.html
[Accessed 9 8 2016].

Azureus Software, Inc, 2016. *Vuze.* [Online]
Available at: http://www.vuze.com/
[Accessed 26 8 2016].

Chhikara, A. & Chhillar, R., 2012. Analyzing the Complexity of Java Programs using Object-Oriented Softare Metrics. *International Journal of Computer Science Issues,* January, 9(1), pp. 364-372.

Chidamber, S. & Kemerer, C., 1994. A Metrics Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering.* s.l.:s.n., pp. 476-493.

Crnogorac, L., Rao, A. S. & Ramamohanarao, K., 1998. *Classifying Inheritance Mechanisms in Concurrent Object-Oriented Programming.* s.l.:Springer Berlin Heidelberg.

Dahl, O.-J., 2004. The Birth of Object Orientation: the Simula Languages. In: *From Object-Orientation to Formal Methods.* s.l.:Springer Berlin Heidelberg, pp. 15-25.

Dahl, O.-J., Dijkstra, E. W. & Hoare, C. A. R., 1972. *Structured Programming.* London: Academic Press.

Fuhrer, R., Keller, M. & Kiezan, A., 2007. Advanced refactoring in the Eclipse JDT: Past, present, and future. In: *Proc. ECOOP Workshop on Refactoring Tools (WRT).* s.l.:s.n., pp. 31-32.

Gamma, E. & Eggenschwiler, T., 2007. *JHotDraw.* [Online]
Available at: http://www.jhotdraw.org/
[Accessed 23 8 2016].

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l.:Pearson Education India.

Harrison, R., Counsell, S. & Nithi, R., 2000. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software,* 52(2), pp. 173-179.

Holub, A., 2003a. *Why getter and setter methods are evil.* [Online]
Available at: http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html
[Accessed 5 8 2016].

Holub, A., 2003b. *Why extends is evil.* [Online]
Available at: http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html
[Accessed 5 8 2016].

Jorgensen, G., 2008. *Typical Programmer.* [Online]
Available at: http://typicalprogrammer.com/doing-it-wrong-getters-and-setters/
[Accessed 5 August 2016].

Kabutz, H., 2006. *How Deep is Your Hierarchy.* [Online]
Available at: http://www.javaspecialists.eu/archive/Issue121.html
[Accessed 7 8 2016].

Kainulainen, P., 2014. *Petri Kainulainen.* [Online]
Available at: https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/
[Accessed 20 8 2016].

Kuhn, T. & Thomann, O., 2006. *eclipse.* [Online]
Available at: https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html
[Accessed 11 8 2016].

Lorenz, M. & Kidd, J., 1994. *Object-Oriented Software Metrics.* s.l.:Prentice-Hall.

Marinescu, R., 1998. Using Object-Oriented Metrics for Automatic Design Flaws Detection in Large Scale Systems. In: *ECOOP Workshops.* s.l.:s.n., pp. 252-255.

Martin, R., 1997. *Java and C++: A Critical Comparision,* s.l.: s.n.

McCabe, T. J., 1976. A complexity measure. *IEEE Transactions on software Engineering,* pp. 308-320.

Meyer, B., 1988. *Object-Oriented Software Construction.* 2nd ed. New York: Prentice Hall.

Micallef, J., 1988. Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming,* 1(1), pp. 12-36.

Naboulsi, Z., 2011. *Code Metrics - Depth of Inheritance (DIT).* [Online]
Available at: https://blogs.msdn.microsoft.com/zainnab/2011/05/19/code-metrics-depth-of-inheritance-dit/
[Accessed 8 8 2016].

Pugh, W. & Hovemeyer, D., 2004. Finding bugs is easy. *ACM SIGPLAN Notices,* 39(12).

Snyder, A., 1986. Encapsulation and Inheritance in Object-Oriented Programming Language. *ACM Sigplan Notices,* 21(11), pp. 38-45.

SourceForge, 2010. *JUNG.* 2.0.1 ed. s.l.:SourceFourge.

Subramanyam, R. & Krishnan, M., 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. In: *IEEE Transactions on Software Engineering.* s.l.:s.n., pp. 297-310.

Taivalsaari, A., 1996. *On the Notion of Inheritance.* 3 ed. s.l.:ACM Computing Surveys (CSUR).

Tandon, S., 2010. *Improve the Quality Of Java-Based Projects Using Metrics.* [Online]
Available at: http://www.devx.com/architect/Article/45611
[Accessed 6 8 2016].

Tempero, E., 2013. *Qualitas Corpus.* s.l.:s.n.

The Eclipse Foundation, 2016. *Eclipse Java development tools.* 3.7.0 ed. s.l.:The Eclipse Foundation.

Thirunarayan, K., 2009. Inheritance in Programming Languages. *Department of Computer Science and Engineering, Wright State University, Dayton, OH-45435.*

Viswanadha, S. & Gesser, J. V., 2016. *JavaParser.* 2.5.1 ed. s.l.:https://github.com/javaparser/javaparser.