Research and Creative Experiences for Undergraduate Programs: Retrospective Case Study and Potential Solutions in Regard to the Object Oriented Design of CharGer Through Various Modeling and Analysis Techniques

Jesse D. Moore

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation

Moore, Jesse D., "Research and Creative Experiences for Undergraduate Programs: Retrospective Case Study and Potential Solutions in Regard to the Object Oriented Design of CharGer Through Various Modeling and Analysis Techniques" (2014). Honors Capstone Projects and Theses. 190. https://louis.uah.edu/honors-capstones/190

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.
Summer of July 2014

Research and Creative Experiences for Undergraduates Program

Retrospective Case Study and Potential Solutions in Regard to the Object Oriented Design of CharGer through Various Modeling and Analysis Techniques.

Jesse D. Moore
Harry S. Delugach Ph.D.
# Table of Contents

- Introduction ................................. 3
- Background .................................... 4-5
- Procedures .................................... 5-7
- Calculations ................................... 7-8
- Interpreting Results ......................... 8-29
  - Shared Metrics ............................ 9-16
  - SoF Metrics ............................... 16-21
  - Metrics 1.3.8 ............................ 21-29
- Results ....................................... 29-37
- Conclusion .................................... 37-40
- Bibliography .................................. 41
- Appendix I : Metrics ......................... 42
- Appendix II : Importance Factors .......... 43
- Appendix III : SoF Metrics Worst Offenders 44-46
- Appendix IV : Metrics 1.3.8 Worst Offenders 47-50
- Appendix V : Shared Worst Offenders .... 50-52
Introduction

Old source code is similar in many ways to old building blueprints. When the building was originally designed, it may have been a modern marvel. As the years pass on, it is no surprise that the original building design relies heavily on archaic architectural structures and safety protocols (asbestos fire proofing, inefficient structural support, etc.). Parallel to this issue in software design, the now popular approach of object oriented design lays down a new set of standards for software architecture to meet. Code created in the early days of programming in a procedural design may be stable and efficient, but it will be out of date because the market has a new set of standards that it will be impossible for old code to meet. For example, asbestos has been widely rejected as a fire retardant because of its known health risks; thus, the standards for fireproofing were changed, causing a change in modern fireproofing methods. Several typical code patterns that tend to occur in software development (whether through being a relic of procedural style and/or poor object oriented design) have been widely rejected because of their detriment to object oriented design models, indicating that a refactoring of the code would serve to improve the blueprint of the software.

The goal of my work was to assess the conceptual graph editor CharGer as it relates to modern object oriented design standards and to propose viable solutions to poorly designed code if possible. Software analysts have a myriad of tools to help in this task more or less at their fingertips. Modern Integrated Development Environments (IDEs) such as NetBeans or Eclipse have extensive libraries available online. There are several plugins that I have used in this study; I will name them now, but will wait until later to describe them in detail: ObjectAid UML Explorer, State of Flow Metrics, and Eclipse Metrics. ObjectAid is used to provide me with UML diagrams from the source code to determine class relationships. The other plugins for metrics are used to provide detailed output about certain functional aspects of the code.
Background

CharGer

CharGer is a conceptual graph editor developed by Dr. Harry Delugach. In addition to being a tool for conceptual graph researchers, it also contains a repertory grid tool, an in program dictionary/thesaurus, and a tool for analyzing models created by the team model acquisition tool. Essentially, CharGer is used to map concepts, relations, and actors using various linking tools to serve as an intermediary map that takes natural language to a computing oriented design. This tool has been in development for just about twenty years now. Initially, a conceptual graph API called Notio was used for development of the conceptual graphs, but in recent years the Notio has been getting phased out of CharGer. For more information, visit http://charger.sourceforge.net/.

ObjectAid UML Explorer

ObjectAid UML Explorer is an Eclipse plugin for Java code visualization developed by ObjectAid LLC. Essentially, it allows for the reverse engineering of Java Source code into UML diagrams. This is helpful for code analysis since it allows you to see how classes in the project relate to each other. For more information, visit http://objectaid.com/.

State of Flow EclipseMetrics (SoF Metrics)

EclipseMetrics is an Eclipse plugin that calculates various metrics on Java code developed by Lance Walton. These metrics can serve to highlight sections of code that have the potential to be poorly designed. It is important to know that just because a section of code is not highlighted by the metrics does not imply that it is good code. For more information, visit http://www.stateofflow.com/projects/16/eclipsemetrics/.
Eclipse Metrics Plugin 1.3.8

Eclipse Metrics Plugin is a plugin that calculates various metrics on Java code; it has been developed by Frank Sauer and Keith Cassell. It offers a wider range of metrics than those of SoF, but is more limited in its data output features. For more information, visit http://metrics2.sourceforge.net/

Procedure

1. Migrating IDE

At the time of beginning the case study, work on CharGer was done in the NetBeans IDE. Upon researching into plugins for code analysis, Eclipse IDE was found to be better suited for the project. The migration of the code was fairly simple; each package was imported individually from the NetBeans project folder into an Eclipse project folder. Finally, all associated libraries were added to the Eclipse CharGer project build path.

2. Forming UML Diagrams

In an effort to get more familiar with the overall design of CharGer, ObjectAid UML Explorer was used create UML diagrams of the main packages of CharGer. ObjectAid was installed by choosing the “Install New Software” option under help in the Eclipse toolbar and adding the ObjectAid UML Explorer update site. Each package with source code had a UML folder designated for it; from there, a new proprietary ObjectAid file (extension .ucls) was created for that specific package. Once the UML diagram was instantiated, all package associated classes were included (within a reasonable number) so that a bigger picture of interaction between classes could be gleaned. ObjectAid LLC provides a more detailed description on the use of their product at their website listed above.

3. Obtaining Metric Data

This section contains two subsections: State of Flow EclipseMetrics, and Eclipse Metrics Plugin 1.3.8. Metric data sections are cataloged in Appendix I. Interpretations on the metrics used in this study will be covered in the following section, Interpreting Results.
State of Flow EclipseMetrics

SoF Metrics was installed by adding the update site to Eclipse. Once installed, all that was necessary to export the metrics was to context menu select the root CharGer source file and click export. The metrics were then viewable in HTML at a specified file in the computer’s file directory.

Eclipse Metrics Plugin 1.3.8

Metrics 1.3.8 was installed by adding the update site to Eclipse. Unfortunately, there exists no time efficient manner to export the metrics to external files, so all data collection from Metrics 1.3.8 was done through the Eclipse IDE.

4. Identifying “Worst Offenders”

In this process, the top five classes or methods for each metric provided by the metrics plugins were marked. Since the metrics generally serve to highlight poorly designed sections of code, the methods and/or classes that appear in these lists are called the worst offenders. Each metric can be seen as a specific subsection of “crime” done against accepted good design practices. Methods and/or classes that have appeared on the list for the worst offenders in the past versions of CharGer are said to possess prior offenses. Appendices III, IV, and V contain the worst offenders for the SoF Metrics, Metrics 1.3.8, and the shared worst offenders from both SoF and 1.3.8 respectively.

5. Calculating Aggregate Metric Y (SoF only)

Aggregate Metric Y was created to obtain a single value using all the metrics obtained by State of Flow EclipseMetrics to one version of CharGer source code. Theoretically, this value can serve as an anchor point to compare different versions of CharGer. It is important to note however, that the Y metric is an aggregation of all metric values; so as the source code becomes more populated the metric may increase depending on the size of the update, even if many of the worst offenders were culled between versions. See the Calculations section for an in depth explanation of how the Aggregate Metric Y was calculated.
6. Drawing Conclusions

With the worst offenders identified, the hunt for poorly designed code can begin. This step at first seems like a simple enough process, but the metrics being used merely serve as indications of possible infractions. Because of the inherent quantitative nature of the metrics, a high score must be validated as poor design. To name an example, say a graphical component of CharGer exhibits high coupling. This is to be expected to some extent, because it has to communicate with a lot of different classes to accomplish its task. An example of code that would need to be redesigned could be classes that are essentially used for value storage bins. These classes are highly coupled like the graphical components, but are lacking any form of object oriented structure; in this case, it would be a good idea to try and break up this class and distribute its parts uniformly into other preexisting classes or newly created classes. It is also entirely likely that some classes of poor design have successfully evaded detection by the metrics, so it ultimately comes down to the analyst to make judgment calls on code quality. A new design may very well result in one desired metric being improved, but unknowingly result in other metrics being perturbed.

7. Repeating 3-6 with older source code

This particular process is used to gauge CharGer over its developmental history. In particular, if one class or method seems to recur throughout the developmental processes then it indicates that it is likely to be a pressing issue. These components are likely fundamental to CharGer as a whole, so finding an effective refactoring strategy may prove difficult.

Calculations

Aggregate Metric Y

First, every metric in the State of Flow metrics suite was assigned an importance factor. This was derived through subjective reasoning, metrics that tend to qualitatively assess object oriented design were favored over those that assess procedurally designed code; a more extensive list behind why a certain factor was
chosen can be seen in Appendix II. Once an importance factor has been decided for a metric, a total metric \( X_i \) is created that is the sum of all newly weighted individual \( x_i \) values, where \( i \) varies from one to \( n \) (the number of classes and methods). Current \( x \) and Total \( x \) represent the score of the current method or class in the X metric and the total points scored in the metric. Essentially, \( x_i \) is the percent of the method to the total of the metric added to one, times an importance and scaling factor.

\[
\begin{align*}
  x_i &= \left( \frac{\text{Current } x}{\text{Total } x} + 1 \right) \left( \frac{\text{Importance Factor}}{100} \right) \\
  X_{\text{total}} &= \sum_{i=1}^{n} x_i \\
  Y &= \sum_{i=1}^{m} X_i
\end{align*}
\]

Once \( X_i \)'s are created for all metrics, Aggregate Metric \( Y \) can be found by adding the total \( X \)'s together from one to \( m \) (the number of Metrics).

**Interpreting Results**

In this section, each metric will be explained in depth: how it is calculated and why the results are important. First and foremost, there needs to be a section on the importance of cohesion, since a few of the metrics are about testing for the lack of cohesiveness. Metrics shared by both SoF Metrics and Metrics 1.3.8 will be explained second, followed by metrics singular to the SoF Metrics, and lastly the metrics singular to Metrics 1.3.8.

**The Importance of Cohesion**

Class Cohesiveness is important because it refers to the level to which attributes and methods belong together (Bansiya, 1). Examples may be the best route to express the importance of class cohesiveness. If you had a car class that you were designing, surely you would not put the functions to roll the window up and down in the engine class. You would place the functionality in the door that contains the window. For that matter, you would not put the lock switch on the exterior body of the car, because it would render the lock useless. These examples are analogous to lack of cohesion. Good cohesive classes are simple, easy to maintain, and easy to reuse. No one wants to reinvent the wheel because the specific engine in the car has changed. It does not logically make sense in reality or in code.
Shared Metrics

Cyclomatic Complexity

According to McCabe, The cyclomatic number of a graph is defined as the number of edges in the graph, minus the number of vertices, added to the total number of connected components of the graph (McCabe, 308). He goes on to state that if you imagine the exit node of the graph is connected to the entry node of the graph, that the cyclomatic number also represents the number of linearly independent paths in the graph (McCabe, 309). This means that the cyclomatic number can never be below one, and in fact, a one represents the least logically complex graph (i.e. point A to point B); as the number rises, the more logical pathways the code can execute through. Here is an example from CharGer’s utility package:

```java
public static void writeToFile( File f, String s, boolean append ) {
    if ( f != null ) {
        try {
            BufferedWriter out = new BufferedWriter( new FileWriter( f, append ) );
            out.write( s );
            Hub.info( toString() );
            out.close();
        } catch ( Exception e ) {
            Global.error( e.getMessage() );
        }
    }
}
```

The method WriteToFile has a cyclomatic number of three. We know that it must have at least one path because the method has a beginning and an end. By stepping through the method we arrive at the statement “if (f != null)” which branches the program by checking file f for null, this gives us a two. Finally, if f is not null, we try to write to the file; if this doesn’t work we catch the error and return. The try-catch gives us three. With all logical pathways through the function enumerated, our assessment of cyclomatic complexity is concluded.

The results of cyclomatic complexity metrics are important because they serve as an indication how computationally dense a method is. Usually, a high
score with this metric means that the method or class in question can be broken down into other methods or classes; it can also showcase that the underlying logic of the method or class needs to be redesigned to handle various cases with more efficiency. In some cases however, a higher cyclomatic complexity will be unavoidable to solve certain problems. Ultimately, the analyst will have to make the call.

**Efferent Coupling**

Coupling is a measurement of how “connected” a class is to another. Efferent coupling specifically, is the number of other classes that the specific class being tested by the metric depends upon (Chen, 239). Things that are included in efferent coupling are the following: inheritance, interface implementation, parameter types, variable types, and thrown and caught exceptions. Consider the KBException class with an Efferent Coupling of 3.

```java
public class KBException extends Exception {
    private Object source = null;

    public KBException(String message, Object s) {
        super(message);
        source = s;
    }

    public Object getSource() {
        return source;
    }

    public void setSource(Object source) {
        this.source = source;
    }
}
```

KBException inherits from Exception, contains an Object Variable, and receives Strings and Objects on instantiation. Note that the Object class was already counted as a variable type, and was therefore not counted as a parameter type.

Usually, a high value in efferent coupling can indicate that a class is too heavily reliant on outside classes, and is said to be highly dependent on other classes to maintain its functionality. In the case of KBException, it is dependent only on
standard classes of Java, so the likelihood of KBException failing because of its dependencies is very unlikely. If a class were to be reliant on another more volatile class, it could lead to issues in the measured class if the volatile class is changed or if it breaks down. This issue gets particularly interesting when there is a chain of dependencies.

**Lack of Cohesion in Methods (Henderson-Sellers)**

In order to calculate Henderson-Sellers lack of cohesion metric, the following definitions must be made: M is the set of methods defined by the measured class, F is the set of fields defined by the measured class, ρ(f) is the number of methods that access field f, where f is a member of F, and <ρ> is the mean value of ρ(f) over F. With all of these elements defined, it is possible to define Lack of Cohesion as the ratio of <ρ>·|M| over 1 - |M| (Sellers, 147). If the number is close to zero it indicates an overall cohesive structure to the measured class. If the number approaches the opposite end of the boundary (100 in SOF, 1 in Metrics 1.3.8) then the class is said to be lacking in cohesiveness. Theoretically speaking, the most cohesive class uses all of its attributes in all of its methods. In this most ideal case <ρ> - |M| is equal to zero. The metric reaches its upper boundary when <ρ> is equal to one, meaning each attribute is accessed by one method; this indicates no cohesiveness because this class could easily be broken down into M classes. Because the metric increases as the cohesion of the program decreases, this metric is a measure of the Lack of Cohesion and not the measure of Cohesion.

**Number of Parameters**

The number of parameters metric is purely related to the methods defined inside of the class. As is well known in the nature of programming, when a function is defined it provides a list of parameters that the calling method must provide for the function to execute properly. For example, initFontSettings scored a 3 on the Number of Parameters metric, because one String and two int’s must be passed into the function when called.

```
126     public void initFontSettings( String name, int style, int size ) {
127         myFontName = new String( name );
128         myFontStyle = style;
129         myFontSize = size;
```
This metric retains a certain value of importance because if a measured method contains a large number of parameters it may be possible to consolidate some of the data passed into the function into one class and pass an object of that type into the method as opposed to all of the individual pieces. Undeniably, passing a lot of data to a method by arguments results in a complex and long method call; it is also likely that the more parameters are declared with a method the more complex the method itself will be to handle those methods.

**Nested Block Depth/Number of Levels**

When logical statements are placed inside other logical statements, the interior statement is said to be “nested” inside of the exterior one. Because the exterior statement must qualify in a certain way for the interior statements of the logical expression to execute, the syntax of tabbing these interior statements precipitated in order to improve readability. While the syntax is technically only formality, the logical flow of these nested statements do contain levels or depth in a logical sense, because one level must be reached before the one inside it can be reached. The predominant focus of code design should not rest in syntax, because the indentation of the code has marginal effects on the compiler and zero effect on the efficiency of the program. It must be stressed that the semantics of this metric are more important than the text format.

```java
public GOID( String s ) {
    if ( s == null )
        ident = "-1";
    else
        ident = s;
}
```

This code snippet for initializing Graph Object ID’s has a nested block depth of two, the first level is created by the open and closing brackets of the function
and the second level is created by the statement “if (s == null)”. The if statement itself is in the first level, but the statement “ident = "-1";” which follows the if statement is in level two. The else statement resides in level one, but the statement “ident = s” resides in level two.

The more depth each function contains the more logically complex it is. It follows that if the function is logically complex it will be difficult to properly test and understand. It can also serve to highlight sections of code that have been haphazardly designed, where a programmer is programming by exception as opposed to programming by design.

**Weighted Methods per Class**

The weighted methods per class metric consolidates the sum of the cyclomatic complexity scores of all the methods in the class (Chidamber, 483).

```java
public class ShallowIterator extends GraphObjectIterator {

    /**
     * @param g Graph all of whose elements are collected to form the iterator
     * @see DeepIterator
     */
    public ShallowIterator( Graph g ) {
        super( g, null, GraphObject.Kind.ALL, false );
    }

    /**
     * @param g graph whose elements of one kind are collected to form the iterator
     * @param kind one of the GraphObject GNODE, GEDGE or GRAPH
     * @see DeepIterator
     */
    public ShallowIterator( Graph g, GraphObject.Kind kind ) {
        super( g, null, kind, false );
    }

    /**
     * @param g graph whose elements of one class are collected to form the iterator
     * @param go one of the GraphObject subclasses
     * @see DeepIterator
     */
    public ShallowIterator( Graph g, GraphObject go ) {
        super( g, go, GraphObject.Kind.ALL, false );
    }
}
```
The ShallowIterator class has a Weighted Methods per Class Score of three. Each of its methods has a cyclomatic complexity of one because it contains no logical decisions. The sum of three methods with cyclomatic complexity values equal to one is three.

This metric is used in pursuit of the same type of data that is obtained by the cyclomatic complexity metric. The difference between the two metrics is nevertheless important. Cyclomatic complexity pertains only to methods; Weighted Methods per Class pertains to classes as a whole. Just like a method with high cyclomatic complexity, a class with a high metric in Weighted Methods per Class should be inspected for efficiency and perhaps be broken into multiple classes.

**Number of Attributes**

An attribute (sometimes called a field or a member variable) is a variable that is declared inside the scope of an object, for access by all methods encapsulated by the object (Baldwin). Following this definition, it is important to note that instance variables declared inside of class methods are not attributes, since an object does not consistently preserve the state of these variables past the end of the method call. KBException has one attribute:

```java
public class KBException extends Exception {
  private Object source = null;

  public KBException( String message, Object s ) {
    super( message );
    source = s;
  }

  public Object getSource() {
    return source;
  }

  public void setSource( Object source ) {
    this.source = source;
  }
}
```

If a specific class has a large number of attributes, it should almost certainly use all of them; if a class has a large number of attributes and is not using them it is
poor design, because the class is lacking cohesiveness. In this respect a good class with a large number of attributes has a lot of data to manage. A class might just be large out of necessity, but more often than not classes get large out of convenience. Often, classes with a large number of attributes can be refactored into other classes that are strongly coupled, although this may prove difficult if the program has been designed around the large measured class.

**Lines of Code in Method**

Lines of Code in Method has been renowned by many software analysts as a highly disputed area of metric use. Many philosophical problems plague this metric when it comes down to the deciding factors of what constitutes a line of code. Companies in the field of Line of Code metrics have devised several different definitions for what constitutes a line of code from counting every line in source file to specifying only effective and logical lines of code (M Squared Technologies). SoF Metrics approaches the problem by simply counting all of the lines in the method, where Metrics 1.3.8 counts non-blank, non-commented lines of code in the method.

```java
/** Perform any clean-up required by the actor when it is deleted or its graph is de-activated. */
public void stopActor()
{
    qbar.setVisible(false);
    qbar = null;
    charger.Global.info( getClass().getName() + " actor stopped."
);}
```

SoF Metrics scores the stopActor method at a nine since it includes comments and blank lines. Metrics 1.3.8 gives stopActor a three, since the only counts the interior statements of the stopActor function.

In many ways, the Lines of Code metric serves to highlight both the good and bad of metric analysis techniques. Certainly, if a program truly has a great number of lines of code in a method, the more linearly complex the method should be theoretically. But if this logic were used with the SoF Metrics on an empty function with a large amount of blank space, the result would be the same as if the function were filled to the brim with statements. Conversely, enough logic and
statements to fill a thousand lines could be fit into one line, and the program would run identically and be just as complex without the lines of code metric even detecting it.

SoF Metrics

Feature Envy

Feature Envy seems to be a less known metric when it comes to metrics used by software analysts. Most references refer to it as a “bad code smell” meaning that it is indicative of bad code even if it may be the only readily available option to employ in the code. The following declarations were needed by SoF metrics to calculate feature envy: m is the method for which we want to calculate Feature Envy, F_c is the set of features used by m that belong to type c, and c_m is the class in which m is defined. Once these are defined then the feature envy of a function can be calculated by subtracting the set of features used by m that belong to the class in which m is defined from the set of features used by m that belong to type c.

Essentially, the metric counts all of the features used in the method and removes all of those contained in the class, effectively getting a count of all methods used from classes external to the measured class.

```java
/**
 * Constructs a new GEdge with label "-".
 */
public GEdge() {
    textLabel = "-";
    myKind = GraphObject.Kind.GEDGE;
    setColor();
}
```

The Feature Envy score for the GEdge constructor is a one. This is because it is reaching inside of the enumerated type Kind in the GraphObject to set its own kind “myKind” as a GEDGE kind.

Feature Envy is a novel metric, though at first its purpose may not seem clear. When a person is envious they hold a strong drive to possess something that belongs to another. When a feature is “envious”, the class strongly wants to possess the functionality of another class. In reality, envy is usually seen as destructive to relationships between the subjects of the envy (that who is envious and those who inspire envy). In Object Oriented Design, it is more or less the same...
The function that is the target of envy is being used by other classes outside its own jurisdiction, increasing its workload outside of its already pertinent class functionalities; while the envious class is outsourcing its workload, making it dependent on classes external to itself. Often times, the solution to these issues is to move the envied feature into the envious class. If both the envious and envied classes need to use the functionalities, it may be best to move the envied functionalities and associated attributes into their own class completely and allow that class to handle it.

Lack of Cohesion in Methods (Chidamber and Kemerer)

In the paper by Chidamber et al., *A Metrics Suite for Object Oriented Design*, a precise mathematical definition for their method of gauging lack of cohesion is defined on page 488. It must be warned however, that the definitions are heavily rooted in set theory. In the recognition that not everyone can fully understand these concepts, an effort will be made in an attempt to clarify how the metric is obtained. The metric analyzes the number of method pairs in a class that have no common fields and subtracts the number of method pairs that do share at least one field. If the method pairs that share at least one common field outnumber the methods that do not, the metric value is set to zero to avoid negative values.

```java
public class KBException extends Exception {
    private Object source = null;

    public KBException( String message, Object s ) {
        super( message );
        source = s;
    }

    public Object getSource() {
        return source;
    }

    public void setSource( Object source ) {
        this.source = source;
    }
}
```

KBException has the Lack of Cohesion in Methods score of zero using the methods proposed by Chidamber et al. There are a total of three method pairs possible inside of KBException; each of these method pairs shares the same
attribute “source” and none of the metric pairs do not share common attributes. Because of this, the metric is scored at zero since the shared attribute method pairs outnumber the no shared attribute method pairs three to zero.

**Lack of Cohesion in Methods (Total Correlation)**

The method of discerning Lack of Cohesion in Methods through Total Correlation determines if a group of variables exhibit redundancy or structure using a generalization through the probability concept of mutual information. Mathematically, each method in the class has to use some subset of the fields in the class, even if that subset is empty. Knowing this, we want to see if some sort of pattern arises from these attribute subsets and their methods. If a structure can be formed from these patterns we know that these structures can probably be extracted from the class into one or more classes. While at first it may seem antithetical to remove these structures from the class, it is important to note that by the nature of the Total Correlation the ideal scenario for true cohesion would exhibit no structure, because all fields in the class would be used by all the methods.

```java
public class KBException extends Exception {
    private Object source = null;

    public KBException( String message, Object s ) {
        super( message );
        source = s;
    }

    public Object getSource() {
        return source;
    }

    public void setSource( Object source ) {
      this.source = source;
    }
}
```

KBException scores a zero in the Total Correlation metric. This is because each method has the same set of attributes, the one attribute source. Because there are no different subsets among the methods attribute usage, Total Correlation strategy dictates that there is “no structure” to the class.
Lack of Cohesion in Methods (Pairwise Field Irrelation)

Lance Walton makes the following definitions before calculating the lack of cohesion in methods using the pairwise field irrelation method: $M$ is the set of methods defined by the class, $F$ is the set of fields defined by the class, and $M_f$ is the subset of $M$ that access a field $f$ that is a member of set $F$. He then defines the Total Field Irrelation as the average Jaccard Distance between $M_{f_1}$ and $M_{f_2}$ where $f_1$ does not equal $f_2$. The Jaccard Distance $J_D$ is defined as one minus the Jaccard similarity coefficient $J$.

$$J(M_{f_1}, M_{f_2}) = \frac{|M_{f_1} \cap M_{f_2}|}{|M_{f_1} \cup M_{f_2}|}$$

$$J_D(M_{f_1}, M_{f_2}) = 1 - J(M_{f_1}, M_{f_2}) = \frac{|M_{f_1} \cup M_{f_2}| - |M_{f_1} \cap M_{f_2}|}{|M_{f_1} \cup M_{f_2}|}$$

The Jaccard similarity coefficient is the quotient of the intersection of the two sets over the union of the two sets, and therefore serves as an indicator of the similarity between the two sets. The coefficient is bounded by one and zero, so subtracting one by the Jaccard similarity coefficient gives us the Jaccard distance, which can be used to measure dissimilarity. When applied to our method subsets $M_f$ it can tell us the dissimilarity between our various methods and their attribute.
access. If our methods access the same variables, we get a Jaccard distance of zero because the Jaccard similarity coefficient is one. Each possible pair methods in the class go through this calculation and get averaged together to give us the Pairwise Field Irrelation.

```java
public class KBException extends Exception {
    private Object source = null;

    public KBException(String message, Object s) {
        super(message);
        source = s;
    }

    public Object getSource() {
        return source;
    }

    public void setSource(Object source) {
        this.source = source;
    }
}
```

KBException receives a zero from the Pairwise Field Irrelation metric. For the three possible method pairs that can be formed, each shares the same attribute “source”. This means that all possible Jaccard similarity indices are equal to one and therefore all Jaccard distances are zero. This indicates that the mean of the Jaccard distances has to be zero as well.
Number of Statements

The Number of Statements metric is usually seen as an “upgraded” version of the standard Lines of Code in Method metric. It is interesting to note that the Number of Statements metric in the State of Flow suite is more similar to the Lines of Code in Method metric in the suite provided by Metrics 1.3.8 than the Lines of Code in Method metric in its own suite, despite the fact that they are all similar to begin with. The following statements are counted by the metric: break, continue, do, explicit constructor call, explicit super constructor call, for, if, return, switch, throw, try, catch, finally, while, assignments, method calls, pre/post increment/decrement. stopActor has five statements, each indicated uniquely.

```
/**
   Perform any clean-up required by the actor when it is deleted or its graph is de-activated.
*/
public void stopActor()
{
    qbar.setVisible( false );
    qbar = null;
    charger.Global.info( getClass().getName() + " actor stopped. " );
}
```

This metric extrinsically gets a lot of its value in being more precise than the average Lines of Code in Method metric. But, there are several metrics that measure lines of code that are in fact more precise than this particular metric. stopActor scored a three on Metrics 1.3.8 Lines of Code in Method metric because it just measured the lines instead of going in depth and measuring against a list of preset statements. The presence of a large number of statements in a method shows that the method is dense with various logical statements as it would figure. This may be necessary for the program to run as efficiently as possible, but it is likely that certain aspects of the method could be refactored into other methods; potentially even another class if the number of statements is vast.
Metrics 1.3.8

Number of Classes

If the reader is familiar with object oriented design at all, this metric should be relatively straightforward. Metrics 1.3.8 analyzes the project source code and arranges it for viewing by packages. Each package contains classes that are related by some means. Someone designing an airplane in an object oriented language would have packages for the engines, landing gear, cockpit, fuselage, etc.; these packages would be composed of classes that detail the specific inner workings of their respective package. This metric simply counts the number of classes per package. For example, the charger package in the source files of CharGer has 26 classes.

Number of Children

Object oriented languages in general allow for inheritance. Inheritance is a relationship which forms between the class that is being inherited from (the parent) and classes that are inheriting (the children). The child class takes on the same attributes and methods as the parent, but may also have extra methods or attributes. In addition to direct children in the inheritance model, classes that implement an interface into their designs are also classified as children with the interface as the parent. The ActorPlugin interface is listed as having six children, this means that six classes implement the ActorPlugin interface.

Number of Interfaces

This metric scans all of the Java files available in the source package of CharGer and looks for the keyword “interface” to count the number of interfaces in each Java source file. An interface is essentially a collection of methods (and sometimes constant values) that can be tacked onto existing classes for improved functionality. An example of a Java interface would be something like an ice cream decorator. Any ice cream class can implement the ice cream decorator to allow for ice cream objects to take on various toppings like chocolate chips or
sprinkles and keep track of extra cost, extra calories, etc. Metrics 1.3.8 tells us that the Java file ActorPlugin.java contains precisely one interface. It is generally common practice to declare any new interface design in a new source file, but it is possible to declare multiple interfaces in one source file if it is particularly needed for a specific reason.

Depth of Inheritance Tree

Using the concepts of inheritance, it is possible to place superclasses and subclasses into a tree structure, like this:

In this tree structure, the parent classes form the root of the tree while the children form the leaves. Every subclass in the tree must necessarily retain the functionalities and attributes of their parent classes, and since they are a subclass it would be poor object oriented design if the subclass did not have any new
functionalities with it, otherwise the subclass would be pointless. Workers and Software Developers must be Employees, and School students and College Students must be Students; in this same nature Employees and Students must be Persons. With this taken into account, it becomes clear that each new subclass is more complex than its parent class, so it can serve as a useful measurement of potential complexity (Chidamber, 483). In Java specifically, all classes stem from one root class defined in Java as Object, making it the only true class in Java without a superclass. Since it is the root of all objects created in Java, subclasses are measured down the tree from the root class Object, through all of their preceding superclasses, to their own class. KBException scores a depth of inheritance of three. Object → Throwable → Exception → KBException is its path in the inheritance tree.

Number of Overridden Methods

Method overriding takes place when a child class provides an implementation for a method that has already been declared by a parent class. In this way, a list of data containing both parent and child objects can be easily traversed with one method call and still allow for variability between parent and child reactions. In example, a superclass called pet can be used to define cats, dogs, and parrots. In the pet class, it is declared that all pets have to implement their own makeNoise function. This way, when we go through a list of pets and call makeNoise cats, dogs, and parrots will all respond differently.

```java
@Override
public String toString() {
    String out = "";
    out += "Capacity = " + stateList.size();
    out += ". Current state = " + currentIndex + "\n";
    if (currentIndex >= 0) {
        for (int stateNum = 0; stateNum < stateList.size(); stateNum++) {
            out += "state " + stateNum + " = ";
            T state = stateList.get(stateNum);
            if (state == null) {
                out += " null";
            } else {
                out += state.toString();
            }
            out += "\n";
        }
    } else {
        out += state.toString();
    }
    return out;
}
```
This is an example of an overridden method in the UndoRedoList class. It is overriding the toString method declared by the Object class. By default, if you call toString on any object without an overridden toString you get the name of the class and the location of memory, but if you call toString on an instantiation of the UndoRedoList the code listed above will run. Method overriding is important as a metric because it is usually good programming practice to let superclass declarations handle as much as possible and overriding should only take place in special functions. If a lot of method overriding has to take place in a subclass it may be an indicator to reshape the inheritance hierarchy.

**Number of Methods**

This is a straightforward metric. For each method declaration in the source code of a class, this metric is incremented. KBException on page 19 has precisely three methods available to any KBException object. In all of CharGer, there are 1,746 methods. Usually, a class with a large number of methods can be considered more complex than a class with a smaller number of methods. This means that a class with a large number of methods should be vetted to see if all of its methods are cohesive. If this is not the case, one should go about reducing the number of methods by creating new classes or by reassigning methods to classes better suited for holding them.

**Specialization Index**

The Specialization Index metric is a class based metric used to gauge properties of inheritance. Specifically, the specialization index is calculated by taking the product of the class’s Number of Overridden Methods and its Depth of Inheritance Tree over the Total Number of Methods in the class. The specialization index is said to highlight classes worth looking at for design problems; particularly when it comes to their placement in the inheritance hierarchy (Genero, 69). As stated in the section on Number of Overridden Methods, a large number of overridden methods deep in the inheritance hierarchy suggest poor subclassing; this metric gauges the number of overridden methods in the class compared to the total number of methods in class, taking into account its depth of inheritance to attempt and indicate these issues. KBException (In addition to other classes with zero overridden methods) scores a zero Specialization Index, because any time
KBException gets called as an Exception it will use the methods of Exception, and not its own implementation specific versions of those methods.

**Afferent Coupling**

Coupling is a measure of how “connected” classes are. Specifically, Afferent Coupling is the measure of how many other classes are dependent upon the measured class (Jureczko). Efferent Coupling and Afferent Coupling are very similar in concept but they are differentiated when it comes to the matter of informational flow. Efferent Coupling indicates how many classes the measured class relies on, while Afferent Coupling indicates how many classes rely upon the measured class. A class with high Afferent Coupling should be checked for cohesiveness, and should then be broken up into new classes or moved into classes with dependencies on the current measured class. In Metrics 1.3.8, Coupling is measured on a package level. For example, the charger package has an Afferent Coupling value of 83. This means that there are 83 instances of classes outside of the charger package using charger classes.

**Instability**

The Instability metric gauges the ratio of efferent coupling to the sum of both efferent and afferent coupling. When the Instability score of a package is equal to zero, it is maximally stable; when the Instability score of a package is equal to one, it is maximally unstable (Martin, 6). Packages that are not reliant upon other packages (I.E. packages with no afferent couplings) are stable because their operations are independent of other packages in the source code. The inverse of the statement holds true as well; packages that are solely reliant upon other packages (I.E. packages with only efferent couplings) are unstable because their operations are dependent on other packages in the source code. The plugin package in CharGer is completely unstable with a score of one. This is not surprising since plugins are designed to be reliant on a code base for use without any independent functionality.

**Abstractness**

Abstractness is the ratio of the number of abstract classes in a package to the total number of classes in the package (Martin, 6). Abstract classes cannot be
directly instantiated, but serve to lay out a schematic for child classes to modify restrictively. A class called Tree is likely an abstract class; you can create a lot of different types of trees (oak, pine, maple, etc.), but there is no example in reality of a Tree that is not also type specific. In another example, there is a class called Vehicle; one cannot just create Vehicle, one must create some instance of a Vehicle like a car or a plane. Similarly, one cannot create and implement CGDatabase; they can only create objects derived from the abstract class and implement them like TextDatabase.

```java
abstract public interface CGDatabase {
    abstract public String getName();
    abstract public void resetDB();
    abstract public void closeDB();
    abstract public String doLookup( String keytype, String keyvalue, String targettype );
}
```

The nature of abstract classes allow for highly stable classes that can be specified into one or more (usually less stable) implementation of the class. These less stable implementations can still be accessed through the functionality of their abstract parent class via properties of inheritance, so overall stability of the program is increased. At the same time, there must be classes that are not abstract; because without concrete implementation the program cannot run. It would be like designing a car without specifying the components of the car.

**Distance from the Main Sequence**
By plotting Abstraction against Instability, it is possible to see that the Main sequence is more or less a balance between abstraction and instability. It is best to have a (1,0) or a (0,1) ordered pair on this graph, this is not always attainable however. Secondly, it is best to have it resting as close to the main sequence as possible; this way it is neither too abstract for its stability nor too unstable for its abstractness (Martin, 7). The equation |(A+I-1)| can be used to calculate this distance between the range of zero and one. The preferred distance between the main sequence and the package is zero. A one indicates that it is the greatest distance away from the main sequence possible and is therefore the worst score. Since plugin has a one for instability and a zero for abstraction, plugin has a zero distance from the Main Sequence.

**Number of Static Methods**

This metric is the same as the Number of Methods metric except it excludes all methods that are not static. Static methods are identifiable by the keyword “static”, here is an example:

```java
public static EditFrame getCurrentEditFrame() {
    return CurrentEditFrame;
}
```

The static keyword turns the method from an object method to a class method. This means that the method can be accessed without having to first instantiate an object of the class. The method getCurrentEditFrame is a method belonging to the Global class. Because it is static, one only has to access the method through the class name like so “Global.getCurrentEditFrame()” as opposed to creating a Global instance called x and using “x.getCurrentEditFrame()”. Static methods do serve some practical use, particularly with utility classes for math libraries, system operations, and other similar code segments where it does not make practical sense to have to instantiate an object to access a method. In most standard applications, most necessary utilities will already be prepackaged in the development kit or easily attainable. Static methods should be avoided in object oriented languages because they are more or less a workaround for holding onto global methods.
Number of Static Attributes

This metric is the same as the Number of Attributes metric except it excludes all attributes that are not static. Static attributes are identifiable by the keyword “static”, here is an example:

```java
public static boolean craftEnabled = false;
```

The static keyword turns the attribute from an object attribute to a class attribute. The attribute craftEnabled is an attribute belonging to the Global class. Because it is static, one only has to access the attribute through the class name like so “Global.craftEnabled” as opposed to creating a Global instance called x and using “x.craftEnabled”. Static Attributes should be used sparingly in an object oriented languages because they tend to be used as global placeholders for data.

Results

NOTE: Appendices III, IV, and V contain the worst offenders for the SoF Metrics, Metrics 1.3.8, and the shared worst offenders from both SoF and 1.3.8 respectively. These particular sets of raw data were highly integral to the project, but are too large to place in the results section. As such, Appendices III – V will be referenced many times.

UML Diagrams

The UML Diagrams created by ObjectAidUML were distributed with this document in a folder titled “ObjectAidUML”. The diagrams can serve as a roadmap of sorts by allowing the reader to see the overall design of the program; this will allow for a certain amount of familiarity with the layout of the source code even if one is not familiar with the finer points of object oriented design. In Appendices III – V, most of the metrics listed have an associated package with them; these packages can be located in the ObjectAidUML to see the classes in the package itself, as well as some other classes in other packages the diagramed package associates with.
Worst Offenders

On whole, the data obtained from the Metrics Plug-ins speaks for itself once the consideration to the metrics given in the section Interpreting Results is affiliated with a metric listed in Appendices III – V. Here are some examples on the analysis techniques applied to the worst offenders. The format for listed items is usually as follows: method being tested – method Class – class Package, class – class package, or just package. The format varies based on the specification of each metric. The metrics work on a method, class, or package level; methods have a class and a package, classes have a package, and packages are independent.

State of Flow – Cyclomatic Complexity

1. actionPerformed() - EditManager – charger
2. initComponents() - MMAnalysisFrame – mm
3. startElement() - CGXParser – charger.xml
4. initComponents() - HubFrame – charger
5. activateArithmeticPrimitive() - ActorPrimitive – charger.act

By looking at the data provided by the SoF Cyclomatic Complexity metric it is possible to tell that it is working on a method based level. Now it is possible to scan these quantitative measures for poor method design to assess them quantitatively. Here is the method actionPerformed:

```java
public void actionPerformed( ActionEvent e ) {
    // handle all button events here
    Hub.info("at action performed: " + e.toString());
    Object source = e.getSource();

    JMenuItem sourceMenuItem = null;
    if ( source instanceof JMenuItem ) {
        sourceMenuItem = (JMenuItem)source;
        //parentMenuItem = sourceMenuItem.getParent();
    } else if ( source instanceof JButton ) {
        //Hub.info("a button was pressed.");
        // If a button was pressed, simulate the menu selection, and continue processing
        if ( source == ef.makeContextButton ) {
            e = new ActionEvent( source, ActionEvent.ACTION_PERFORMED, Global.struts("MakeContextLabel")
        } );
        if ( source == ef.makeCutButton ) {
            e = new ActionEvent( source, ActionEvent.ACTION_PERFORMED, Global.struts("MakeCutLabel")
        } );
    }
    if ( source == ef.alignHorizontalButton ) {
        e = new ActionEvent( source, ActionEvent.ACTION_PERFORMED, Global.struts( "AlignHLabel" )
    } if ( source == ef.alignVerticalButton ) {
        e = new ActionEvent( source, ActionEvent.ACTION_PERFORMED, Global.struts("AlignVLabel")
    } if ( source == ef.unMakeContextButton ) {
```
e = new ActionEvent( source, ActionEvent.ACTION_PERFORMED, Global.strs("UnMakeContextLabel") );
)
    ef.requestFocus();
    }
    ef.clearStatus();
    //Hub.info("event e "+ e);
    if ( e.getActionCommand().equals( Global.struts("NewWindowLabel")) ) {
      EditFrame ef = new EditFrame();
      if ( Global.enableEditFrameThreads ) {
        new Thread( Global.EditFrameThreadGroup, ef ).start();
      }
      ef = null; // 09-05-05 : maybe will help with memory leaks
      //Hub.setCurrentEditFrame( ef );
    } else if ( e.getActionCommand().equals( Global.struts("CutLabel")) ) {
      performActionClipBoardCut();
    } else if ( e.getActionCommand().equals( Global.struts("CopyLabel")) ) {
      performActionClipBoardCopy();
    } else if ( e.getActionCommand().equals( Global.struts("PasteLabel")) ) {
      performActionClipBoardPaste( ef.lastMouseClickPoint );
    } else if ( e.getActionCommand().equals( Global.struts("Duplicatelabel")) ) {
      performActionDupSelection();
    } else if ( e.getActionCommand().equals( Global.struts("UndoLabel")) ) {
      performActionUndo();
    } else if ( e.getActionCommand().equals( Global.struts("Redolabel")) ) {
      performActionRedo();
    } else if ( e.getActionCommand().equals( Global.struts("Clearlabel")) ) {
      performActionDeleteSelection();
    } else if ( e.getActionCommand().equals( Global.struts("SelectAllLabel")) ) {
      performActionSelectAll();
    } else if ( e.getActionCommand().equals( Global.struts("MakeContextlabel")) ) {
      performActionMakeContext( false );
    } else if ( e.getActionCommand().equals( Global.struts("MakeCutLabel")) ) {
      performActionMakeContext( true );
    } else if ( e.getActionCommand().equals( Global.struts("UnMakeContextlabel")) ) {
    } else if ( e.getActionCommand().equals( Global.struts("AlignVLabel")) ) {
      performActionAlign( false );
    } else if ( e.getActionCommand().equals( Global.struts("AlignHLabel")) ) {
      performActionAlign( false );
    } else if ( e.getActionCommand().equals( Global.struts("MininizeLabel")) ) {
      performActionMininizeSelection();
    } else if ( e.getActionCommand().equals( OperManager.MakeGenericCmdLabel ) ) {
      ef.omgr.performActionMakeGeneric();
    } else if ( e.getActionCommand().equals( OperManager.ValidateCmdLabel ) ) {
      performActionValidate( ef.TheGraph );
    } else if ( e.getActionCommand().equals( "Show Internals" ) ) {
      ef.omgr.performActionShowInternals( sortSelectionObjects() );
      // REMOVE-NOTIO  else if ( e.getActionCommand().equals(OperManager.ModifyMatchingSchemeLabel) ){
      // REMOVE-NOTIO  ef.omgr.performActionCreateMatchingScheme();
    } else if ( e.getActionCommand().equals( charger.Global.struts( "AttachOntologyLabel" ) ) ) {
      ef.omgr.performActionAttachOntologyLabel( sortSelectionObjects() );
      // REMOVE-NOTIO  ef.omgr.performActionAttachOntologyLabel( sortSelectionObjects() );
      // REMOVE-NOTIO  } else if ( e.getActionCommand().equals(OperManager.AttachOntologyLabel) ) {
      // REMOVE-NOTIO  ef.omgr.performActionAttachOntologyLabel( sortSelectionObjects() );
      // REMOVE-NOTIO  } else if ( e.getActionCommand().equals(OperManager.DetachOntologyLabel) ) {
      // REMOVE-NOTIO  ef.omgr.performActionDetachOntologyLabel( sortSelectionObjects() );
      // REMOVE-NOTIO  } else if ( e.getActionCommand().equals(OperManager.MakeGenericCmdLabel ) ) {
      // REMOVE-NOTIO  ef.omgr.performActionMakeGeneric();
      // REMOVE-NOTIO  } else if ( e.getActionCommand().equals(OperManager.ValidateCmdLabel ) ) {
      // REMOVE-NOTIO  performActionValidate( ef.TheGraph );
    } else if ( e.getActionCommand().equals( Global.struts("DisplayAsCGIFLabel") ) ) {
```java
// These are the file export format commands
// Formats of family FileFormat.Family.TEXT
else if (e.getActionCommand().equals(Global.strs("ExportCGIFLabel"))) {
    performActionSaveGraphFormattedAs(FileFormat.CGIF2007);
} else if (e.getActionCommand().equals(Global.strs("SaveAsLabel"))) {
    performActionSaveGraphFormattedAs(FileFormat.CHARGER3);
} else if (e.getActionCommand().equals(Global.strs("ExportTestXMLLabel"))) {
    performActionSaveGraphFormattedAs(FileFormat.CHARGER4);
} else if (e.getActionCommand().startsWith("PDF")) {
    performActionSaveGraphFormattedAs(FileFormat.PDF);
} else if (e.getActionCommand().startsWith("SVG")) {
    performActionSaveGraphFormattedAs(FileFormat.SVG);
} else if (e.getActionCommand().startsWith("EPS")) {
    performActionSaveGraphFormattedAs(FileFormat.EPS);
}
else if (e.getActionCommand().equals(Global.strs("ImportCGIFLabel"))) {
    performActionImportCGIF();
} else if (e.getActionCommand().equals(Global.strs("PageSetupLabel"))) {
    performActionPageSetup();
} else if (e.getActionCommand().equals(Global.strs("Printlabel"))) {
    performActionPrintGraph();
} else if (e.getActionCommand().equals(Global.strs("Openlabel"))) {
    performOpenGraph();
} else if (e.getActionCommand().equals(Global.strs("Closelabel"))) {
    ef.closeOut();
} else if (e.getActionCommand().equals(Global.strs("PreferencesLabel"))) {
    performActionPreferences();
} else if (e.getActionCommand().equals(Global.strs("Quitlabel"))) {
    performActionQuit();
}
```

if ( sourceMenuItem == ef.ChangeFillItem ) // if ( sourceMenuItem.getText().equals("Fill") )
{
  // change the text color of the selection
  c = JColorChooser.showDialog( ef,
                               "Choose fill color for selected objects.", c);
  if ( c != null )
    performActionColorSelection( "fill", c );
} else if ( sourceMenuItem == ef.ChangeColorDefaultItem )
{
  // change color to match color defaults
  performActionColorSelection( "text", Global.userForeground );
  performActionColorSelection( "fill", Global.userBackground );
} else if ( sourceMenuItem == ef.ChangeColorFactoryItem )
{
  // change color to match color defaults
  //Hub.useFactoryDefaultColors( true );
  performActionColorSelection( "text", Global.factoryForeground );
  performActionColorSelection( "fill", Global.factoryBackground );
  //Hub.useFactoryDefaultColors( false );
} else if ( sourceMenuItem == ef.ChangeColorBlackAndWhiteItem )
{
  // change color to match color defaults
  //showBorders.setSelected( false );
  showBorders.doClick(); // force borders to be turned on
  showShadows.setSelected( true );
  showShadows.doClick(); // force shadows to be turned off
  Global.showBorders = true;
  Global.showShadows = false;
  //Hub.useFactoryDefaultColors( true );
  performActionColorSelection( "text", Global.bwForeground );
  performActionColorSelection( "fill", Global.bwBackground );
  //Hub.useFactoryDefaultColors( false );
} else if ( sourceMenuItem == ef.ChangeColorGrayscaleItem )
{
  // change color to match color defaults
  //showBorders.setSelected( false );
  showBorders.doClick(); // force borders to be turned on
  showShadows.setSelected( true );
  showShadows.doClick(); // force shadows to be turned off
  //Hub.showBorders = false;
  //Hub.showShadows = false;
  //Hub.useFactoryDefaultColors( true );
  performActionColorSelection( "text", Global.grayForeground );
  performActionColorSelection( "fill", Global.grayBackground );
  //Hub.useFactoryDefaultColors( false );
} else if ( sourceMenuItem == ef.SpringAlgorithmItem )
{ //
  // else if ( sourceMenuItem == ef.SugiyamaAlgorithmItem )
  //{
    //Hub.info( "look for modality label in " + sourceMenuItem.getText() );
    // else if ( GraphModality.isValidLabel( sourceMenuItem.getText() ) )
    //{
      //ef.setGraphModality( sourceMenuItem.getText() );
      // should re-write using File objects
      // String parent = ef.graphSourceFile.getParentFile().getAbsolutePath();
      // ef.renameGraphInFrame( parent + File.separator + Hub.makeUpFileName( ef.graphName, ef.purpose.getAbbr(), Hub.ChargerFileExtension ) );
    //}
    // here, at the end of action performed, yield to prevent an accident
    Thread.yield();
  }
  // here at the end of the action performed tasks
  if ( ef != null )
  {
    ef.refreshBorders();
    ef.setCursor( new Cursor( Cursor.DEFAULT_CURSOR ) );
    if ( ef.somethingHasChanged )
    { //Hub.info( "something changed.");
    }
// DON'T BE FOOLED!!! This section rarely happens!!! somethingHasChanged obviously toasted.
)
)

One does not have to understand code or the finer points of metric based code analysis to grasp that the actionPerformed method is logically complex. The actionPerformed method scored a 63, using the logic behind cyclomatic complexity it can be deduced that there are 63 unique ways to get from the beginning of the function to the end of the function, making this method a monster to test. Certainly, this method needs to be revised, although it would need a very fine grooming to catch all the possibilities it has been delegated.

Here is another example of qualitatively assessing a function on quantitative means:

State of Flow – Feature Envy

1. getMode() – EditManager – charger
2. getMode() – OperManager – charger
3. paintComponent() – QuantityBar.barPanel – charger.util
4. doInBackground() – MMAnalysisMgr.GrandTotalGenerator – mm
5. paintComponent() – CanvasPanel – charger

Another method called getMode from the EditManager class is at the top of a metrics score, this time for feature envy. Here is getMode:

```java
public String getMode( JRadioButton source ) {
    if ( source == ef.selectionTool ) {
        return EditFrame.SelectToolMode;
    }
    if ( source == ef.conceptTool ) {
        return EditFrame.ConToolMode;
    }
    if ( source == ef.relationTool ) {
        return EditFrame.RelToolMode;
    }
    if ( source == ef.actorTool ) {
        return EditFrame.ActorToolMode;
    }
    if ( source == ef.arrowTool ) {
        return EditFrame.ArrowToolMode;
    }
    if ( source == ef.corefTool ) {
        return EditFrame.CorefToolMode;
    }
```
if ( source == ef.typeTool ) {
    return EditFrame.TypeToolMode;
}
if ( source == ef.relTypeTool ) {
    return EditFrame.RelTypeToolMode;
}
if ( source == ef.genSpecLinkTool ) {
    return EditFrame.GenSpecLinkToolMode;
}
if ( source == ef.editTextTool ) {
    return EditFrameEditTextToolMode;
}
if ( source == ef.deleteTool ) {
    return EditFrame.DeleteToolMode;
}
return "none";

Feature Envy tells us generally that this getMode method is using a lot of features external to its own class. This is undoubtedly true, since a multitude of references to an instantiated EditFrame called ef as well as references to static variables of the EditFrame class following the return statements. There are some matters that must be weighed when deciding if this method needs to be refactored. Based on the concepts of feature envy, it is known that this relationship is placing more responsibility on the EditFrame class than on the EditManager class; not only that, but the EditManager is dependent on the EditFrame to work effectively. This being said, it would make sense for a manager to rely on subclass and delegate work to it, because that is what a manager does. The relationship between the two classes seems to be in order, but their methods for interaction seem to lack in formality; in many ways the issue here seems akin to a manager walking into an employee’s office to rummage through his desk for information as opposed to just asking the employee. A solution to this issue is provided in the Conclusions section.

Closing Statements about Worst Offenders

The work portrayed in the two examples above detail the processes of qualitatively analyzing source code for object oriented validity, and while the two examples above lead to the conclusion that the method could use refactoring it is
important to know that this is not always the case. To prove this is the case, the following method called showInputDialog has seven parameters.

```java
public Object showInputDialog( Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue )
    {
        log.append( tell + message + "\n   (" + initialSelectionValue + ")", suggested) + eol );
        String answer = JOptionPane.showInputDialog( parentComponent, message, title, messageType, icon, selectionValues, initialSelectionValue );
        String loganswer = answer;
        if ( answer != null && answer.equals( initialSelectionValue ) )
            loganswer = "agrees with suggestion."
        log.append( user + loganswer + eol );
        return answer;
    }
```

Just because this method has seven parameters does not mean that it needs to be rewritten, because this method makes practical use of all the parameters that it receives in its implementation.

To be clear, the examples serve as a general outline for this process. The analyst receives quantitative data in the form of metrics for possible leads into where potential design flaws can be located. The moment a decision must be made for whether a method or class or package is truly an artifact of bad design, the analyst must weigh what he or she knows about object oriented design against the source code; this must be done in conjunction with knowing what each metric tends to indicates in order to spot flaws in design.

In addition to being able to discern good code from bad code, the analyst must also be fairly intuitive with his or her choices of analysis. Code analysis is similar to hunting in many ways. Both hunter and analyst have to know where to look when it comes to finding their target. A hunter may use a thermal scope to help him identify prey and an analyst may use metrics to help him identify bad code, but both have to use their intuition and experience to guide them to their target and make the correct decision before executing a shot or a refactoring because the act of doing so may have unintended consequence.
**Aggregate Metric Y**

By the time of this report, only two versions of CharGer were loaded into a spreadsheet to have their Aggregate Metric calculated. Furthermore, Metrics 1.3.8 will not give metric readings on code that cannot be compiled by the IDE, this means that trying to calculate an Aggregate Metric with Metrics 1.3.8 was unfeasible, because each old version of CharGer would need to be placed back in working order. The two versions of the code that were tested using the methods defined in the calculation section were the most modern version of the code, as well as a version of CharGer from June 2011. As suspected, The aggregate metric rose from 208.71 in the June 2011 version to 293.22 in the most modern version. This is not surprising because as more features are added to a code base it will undoubtedly be populated with more code, and thus more total metric values. The two spreadsheets used to calculate the Aggregate Metric Y should be located in a folder distributed with this document called “Aggregate Metric Y”. Because each individual metric total is calculated first, it is also possible to see the rise and fall of total individual metric scores across versions as well if the reader is interested.

**Conclusion**

There is much information that can be gleaned in regard to the metrics, the main issue is simply having enough time to look through it all and find potential solutions. Analysis with the help of metrics is certainly far easier that just looking through source code. Combing through the source code may be inevitable for some analysts depending on how in depth they wish to analyze the code, but nevertheless the metrics are helpful. Metrics are not infallible however; there is no clear line that can be obtained that can tell an analyst that a portion of code is poorly designed. It is also important to know that if the metrics did by some miracle achieve 100% accuracy, they would still only list of particular methods, classes, or packages to their own specification; there would more than likely still be elements of poor design that would slip through the grasp of the metrics. The analyst must also be able to think of different designs for code that has been identified as bad while taking care that changing the code for one metric does not push up metrics in other places; this would invariably lead to the analyst needing to check it again and even possibly refactor again. Another important concept that has been clarified in this experience is the understanding that there is no “perfect” design. Certainly
there are some designs that are better than others, but there is a point in all refactoring jobs where the improvements are marginal at best; the work that must be put into refactoring the code at this point tends to cost too much time and not do much in the way of improving design.

A lot of time for this project was spent in creating UML diagrams and gathering metric data, as a result there was very little time to actually propose solutions to CharGer’s design. However, through looking at the metric data and UML diagrams many problem areas of CharGer were identified. EditManager needs to have a more efficient method for handling action events. In addition to its handling of action events, the getMode feature is highly reliant on features of the EditFrame to diagnose the active mouse mode for CharGer. A solution to this could be the implementation of a CharGer specific button class seen here:

This design would allow for the creation and instantiation of specific CharGer buttons that would keep track of their own modes instead of having to use EditManager and EditFrame in tandem to keep up to date with CharGer’s EditFrame mode without having to gut CharGer’s user interface. There is another
form that could be used, perhaps with more ease and better results is a Modable Interface seen on the next page:

```
<table>
<thead>
<tr>
<th>ModableInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Interface&gt;&gt;</td>
</tr>
<tr>
<td>Modable</td>
</tr>
<tr>
<td>mode : string</td>
</tr>
<tr>
<td>setMode(String)</td>
</tr>
<tr>
<td>getMode()</td>
</tr>
</tbody>
</table>
```

The Modable interface provides the same functionality as the solution proposed with an Abstract CGRadioButton class, and would probably be easier to adapt to CharGer since an interface only has to be implemented.

Global is a form of design pattern known as a singleton; the singleton is dubious in the programming world, some say it is a design pattern and others say it is an antipattern. It is a very useful design because it is predominantly a static class, meaning that much of the class functions at the class level and not the object level, something that almost defeats the purpose of object oriented design. At any rate, Global could certainly have some of its attributes placed into other more relevant classes.

Much of the user interface could benefit from being more precisely designed; The PreferencesWindow class has many different tabs to keep track of, ranging from compatibility settings to actor settings. It is suggested that these tabs be isolated from the window and moved into their own classes, letting the preferences window keep track of the individual tabs and then letting the tabs do the work assigned to them. The class in charge of displaying the repertory grid called RGDisplayWindow keeps track of several components that could be broken into other classes theoretically.
To give an overall review of CharGer conclusively, it is a strongly designed program for its size with 28,526 lines of code; this is especially the case when one considers that CharGer has been under rather consistent development since December of 1998 when Java was only version 1.2 while Java’s current version at the time of writing this document is at version 8. Someone who is familiar enough with Java as a language can tell that CharGer has been in development for a while, because there are many graphical components in CharGer that could now be replaced by some of Java’s modern libraries. The fundamental logic contained in CharGer works well, and most classes function well; the problems that CharGer seems to have (classes being too broad, graphical components being jammed together, high coupling between similar objects) are certainly not deal breakers. CharGer certainly does not need to be redesigned on any fundamental level at this point, though it may need one in the coming decade or so.
Bibliography


<http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm>


Appendix I - Metrics
<table>
<thead>
<tr>
<th>SoF Metrics</th>
<th>Metrics 1.3.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Complexity</td>
<td>Number of Classes</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>Number of Children</td>
</tr>
<tr>
<td>Efferent Coupling</td>
<td>Number of Interfaces</td>
</tr>
<tr>
<td>Lack of Cohesion</td>
<td>Depth of Inheritance Tree</td>
</tr>
<tr>
<td>(Chidamber &amp; Kemerer)</td>
<td></td>
</tr>
<tr>
<td>Lack of Cohesion (Henderson-Sellers)</td>
<td>Number of Overridden Methods</td>
</tr>
<tr>
<td>Lack of Cohesion (Total Correlation)</td>
<td>Number of Methods</td>
</tr>
<tr>
<td>Lack of Cohesion</td>
<td>Number of Fields</td>
</tr>
<tr>
<td>(Pairwise Field Irrelation)</td>
<td></td>
</tr>
<tr>
<td>Lines of Code in Method</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>Number of Levels</td>
<td>Specialization Index</td>
</tr>
<tr>
<td>Number of Parameters</td>
<td>Cyclomatic Complexity</td>
</tr>
<tr>
<td>Number of Statements</td>
<td>Weighted Methods Per Class</td>
</tr>
<tr>
<td>Weighted Methods Per Class</td>
<td>Lack of Cohesion in Methods</td>
</tr>
<tr>
<td></td>
<td>Afferent Coupling</td>
</tr>
<tr>
<td></td>
<td>Efferent Coupling</td>
</tr>
<tr>
<td></td>
<td>Instability</td>
</tr>
<tr>
<td></td>
<td>Abstractness</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normalized Distance from Main Sequence</td>
</tr>
<tr>
<td></td>
<td>Method Lines of Code</td>
</tr>
<tr>
<td></td>
<td>Number of Static Methods</td>
</tr>
<tr>
<td></td>
<td>Number of Attributes</td>
</tr>
<tr>
<td></td>
<td>Number of Static Attributes</td>
</tr>
<tr>
<td></td>
<td>Nested Block Statements</td>
</tr>
</tbody>
</table>
Efferent Coupling (Ce) and Lack of Cohesion (LCOM) received the most importance. A class with low cohesiveness can generally be repurposed into at least two different classes. A class with high efferent coupling to other classes could also use a refactoring, because this would indicate that it is heavily dependent on other classes.

Feature envy (FE) is the next highest importance, this one scored higher than most other metrics because a high score in feature envy indicates that a method external to the scored class is being used. A solution would be to either integrate the dependency into the scored class or create a more efficient method to collect the data from the external class to the scored class.

Cyclomatic Complexity (CC) and Weighted Methods per Class (WMC) are next because they serve to show how computationally complex a given class is. This is not necessarily a fundamental issue; however it can indicate that perhaps another class or set of classes could be created to reduce the scored class’s work load.

Number of Fields (NOF) follows CC and WMC because it is a measure of just how much data the scored class contains. Again, this is not necessarily an indicator of an issue, some classes may require a large amount of attributes, it may be possible however to extract another class from the scored class fields, and take any associated methods along with them.

Number of Locals in Scope (NLS), Number of Statements (NOS), and Number of Levels (NOL) are just below Number of Fields because they indicate code complexity to a lesser extent than CC and WMC to such a minute amount that much of the points that get scored in these categories are more or less unavoidable. Classes that score in this area have the possibility of being refactored into other classes, but it is entirely possible that doing so may not be good for design.

Number of Parameters (NOP) is second to last in the list because it only serves to highlight methods where large amounts of parameters are being passed into the scored class. A lot of times with graphical elements this is unavoidable, but it may still be useful to check if consolidation of the elements being passed in could be integrated with the scored class.

Last in the list of metrics is the good old metric Lines of Code (LOCm). Literally lines of code in the method. This metrics serves more to syntax than to actual code design.
Appendix III – SoF Metrics: Worst Offenders

Cyclomatic Complexity
6. actionPerformed() - EditManager – charger
7. initComponents() - MMAnalysisFrame – mm
8. startElement() - CGXParser – charger.xml
9. initComponents() - HubFrame – charger
10. activateArithmeticPrimitive() - ActorPrimitive – charger.act

Feature Envy
6. getMode() – EditManager – charger
7. getMode() – OperManager – charger
8. paintComponent() – QuantityBar.barPanel – charger.util
9. doInBackground() – MMAnalysisMgr.GrandTotalGenerator - mm
10. paintComponent() – CanvasPanel – charger

Lines of Code in Method
1. initComponents() – MMAnalysisFrame – mm
2. initComponents() – HubFrame – charger
3. initAppearanceTab() – PreferencesWindow – charger
4. setupComponents() – EditFrame – charger
5. initComponents() – SynonymEditor – mm

# Locals in Scope
1. parseOneObject() – IOManager – charger
2. fireActor() – GraphUpdater – charger.act
3. init() – JarResources – charger.util
4. parseOneObject2() – IOManager – charger
5. OpenGraphInNewFrame() – Global – charger

# Levels
1. actionPerformed() – EditManager – charger
2. performActorOperation() – ActorPrimative – charger.act
3. setMenuItems() – EditManager – charger
4. startElement() – CGXParser – charger.xml
5. activateArithmeticPrimitive() – ActorPrimative – charger.act

# Parameters
1. `showInputDialog()` – Transcript – charger.util
2. `drawDashedLine()` – CGUtil – charger.util
3. `FontChooser()` – FontChooser – charger.util
4. `SenseQueryDialog()` – SenseQueryDialog – craft
5. `getTableCellRendererComponent()` – RGDisplayWindow.CustomTableCellRenderer – repgrid

# Statements
1. `initComponents()` – MMAppearanceTab – mm
2. `actionPerformed()` – EditManager – charger
3. `initComponents()` – HubFrame – charger
4. `initAppearanceTab()` – PreferencesWindow – charger
5. `initComponents()` – DatabaseFrame – charger.db

Efferent Coupling
1. RGDisplayWindow – repgrid
2. PreferencesWindow – charger
3. EditManager - charger
4. MMAppearanceTab – mm
5. EditFrame – charger

Lack of Cohesion in Methods (Chidamber & Kemerer)
1. EditFrame – charger
2. PreferencesWindow – charger
3. MTeamMetrics – mm
4. RGDisplayWindow – repgrid
5. RepertoryGrid – repgrid

Lack of Cohesion in Methods (Henderson-Sellers)
1. EditingChangeState - charger
2. FileFormat – charger
3. AbstractMatcher – kb.matching
4. MMetrics – mm
5. MMAppearanceTab – mm

Lack of Cohesion in Methods (Pairwise Field Irrelation)
1. EditingChangeState – charger
2. EditorState – charger
3. FileFormat – charger
4. IOManager$TransferableImage – charger  
5. GNode – charger

Lack of Cohesion in Methods (Total Correlation)  
1. HubFrame – charger  
2. MMAAnalysisFrame – mm  
3. PreferencesWindow – charger  
4. RGDisplayWindow – repgrid  
5. EditFrame – charger

# Fields  
1. MMAAnalysisFrame – mm  
2. EditFrame – charger  
3. PreferencesWindow – charger  
4. RGDisplayWindow – repgrid  
5. SynonymEditor – mm

Weighted Methods per Class  
1. EditManager – charger  
2. EditFrame – charger  
3. MMAAnalysisMgr – mm  
4. Graph – charger.obj  
5. RGDisplayWindow – repgrid
Appendix IV - Metrics 1.3.8: Worst Offenders

McCabe’s Cyclomatic Complexity
1. EditManager - charger
2. CGXParser - charger.xml
3. EditFrame - charger
4. ActorPrimative - charger.act
5. RGXMLParser - repgrid.xml

Number of Parameters
1. Transcript - charger.util
2. CGUtil - charger.util
3. FontChooser - charger.util
4. SenseQueryDialog - craft
5. RGDisplayWindow - repgrid

Nested Block Statements
1. ConceptManager - kb
2. MCongruenceMetris - mm
3. EditManager - charger
4. SynonymGroup - mm
5. EditFrame - charger

Afferent Coupling
1. Charger
2. Charger.obj
3. Charger.util
4. Kb
5. Charger.exception

Efferent Coupling
1. Mm
2. Charger.obj
3. Charger
4. Charger.util
5. Charger.exception

Instability
1. Plugin
2. Mm
3. Cgif.generate.compare
4. Charger.layout
5. Repgrid.xml

Abstractness
1. Charger.layout
2. Charger.undo
3. Charger.xml
4. Charger.act
5. Charger.db

Normalized Distance from main sequence
1. Charger
2. Repgrid.tracks
3. Craft
4. Charger.util
5. Charger.wn

Depth of Inheritance Tree
1. EditFrame - charger
2. PreferencesWindow - charger
3. HubFrame - charger
4. CGSplashFrame - charger
5. DatabaseFrame - charger.db

Weighted Methods per Class
1. EditManager - charger
2. EditFrame - charger
3. Graph - charger.obj
4. MMAnalysisMgr - mm
5. GraphUpdater - charger.act

Number of Children
1. ManagedWindow - charger.util
2. ActorPlugin - charger.act
3. GNode - charger.obj
4. CGException - charger.exception
5. GEdge - charger.obj
Number of overridden Methods
1. Graph - charger.obj
2. CGXParser - charger.xml
3. RGXMLParser - repgrid.xml
4. TableMap - charger.util
5. Concept - charger.obj

Lack of Cohesion of Methods (Henderson-Sellers)
1. MFileType - mm
2. FileFormat - charger
3. MatchedBinaryTouple - kb.matching
4. IOManager - charger
5. MMAnalysisFrame - mm

Number of Attributes
1. MMAnalysisFrame - mm
2. EditFrame - charger
3. PreferencesWindow - charger
4. RGDisplayWindow - repgrid
5. SynonymEditor - mm

Number of Static Attributes
1. Global - charger
2. EditFrame - charger
3. GraphModality - charger
4. Tag - charger.util
5. ConceptManager - kb

Number of Methods
1. EditManager - charger
2. EditFrame - charger
3. PreferencesWindow - charger
4. MMAnalysisFrame - mm
5. RGDisplayWindow - repgrid

Number of Static Methods
1. Global - charger
2. IOManager - charger
3. ConceptManager - kb
4. Util - charger.util
5. CGUtil - charger.util

Specialization Index
1. MMAnalysisMgr - mm
2. MModelNameException - mm
3. NodeOrderException - kb.hierarchy
4. RGIIntegerValueRange - repgrid
5. RGTTableModel - repgrid

Number of Classes
1. Mm
2. Charger
3. Repgrid
4. Charger.obj
5. Charger.util

Number of Interfaces
1. ActorPlugin - charger.act
2. CGDatabase - charger.db
3. Undoable - charger.undo
4. ManagedWindow - charger.util
5. KnowledgeSource - kb

Lines of Code
1. EditFrame - charger
2. PreferencesWindow - charger
3. EditManager - charger
4. RGDisplayWindow - repgrid
5. MMAnalysisFrame - mm

Method Lines of Code
1. MMAnalysisFrame - mm
2. HubFrame - charger
3. PreferencesWindow - charger
4. SynonymEditor - mm
5. EditFrame - charger
Appendix V – Shared Metrics

Cyclomatic Complexity
Shared:
1. EditManager
2. CGXParser
3. ActorPrimitive

Number of Parameters
Shared:
1. Transcript
2. CGUtil
3. FontChooser
4. SenseQueryDialog
5. RGDpyDisplayWindow

Nested Block Depth/ # levels
No Shared metrics

Efferent Coupling
Shared:
1. MM
2. Charger

Weighted Methods per Class
Shared:
1. EditManager
2. EditFrame
3. Graph
4. MMAnalysisMgr

Lack of Cohesion in Methods (Henderson Sellers)
Shared:
1. FileFormat
2. MMAnalysisFrame

# attributes/#fields
Shared:
1. MMAnalysisFrame
2. EditFrame
3. PreferencesWindow
4. RGDisplayWindow
5. SynonymEditor

Lines of Code in Methods
Shared:
1. MMAnalysisFrame
2. HubFrame
3. PreferencesWindow
4. EditFrame
5. SynonymEditor