Visually Localizing Design Problems with Disharmony Maps

Richard Wettel* and Michele Lanza†
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

Abstract

Assessing the quality of software design is difficult, as "design" is expressed through guidelines and heuristics, not rigorous rules. One successful approach to assess design quality is based on detection strategies, which are metrics-based composed logical conditions, by which design fragments with specific properties are detected in the source code. Such detection strategies, when executed on large software systems usually return large sets of artifacts, which potentially exhibit one or more "design disharmonies", which are then inspected manually, a cumbersome activity.

In this article we present disharmony maps, a visualization-based approach to locate such flawed software artifacts in large systems. We display the whole system using a 3D visualization technique based on a city metaphor. We enrich such visualizations with the results returned by a number of detection strategies, and thus render both the static structure and the design problems that affect a subject system. We evaluate our approach on a number of open-source Java systems and report on our findings.

CR Categories: H.5.1 [Information Interfaces and Presentations]: Multimedia Information Systems—Artificial, augmented, and virtual realities; K.6.3 [Management of Computing and Information Systems]: Software Management—Software Maintenance; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, reengineering

Keywords: software visualization, software design anomalies

1 Introduction

Designing complex software systems is a difficult task, a process which takes a long time to learn and a skill that must be perfected constantly. Over the past two decades a number of design guidelines and recipes have been formulated, usually in the form of patterns [Gamma et al. 1995] or heuristics [Riel 1996]. Nonetheless, due to external factors, namely a changing environment which triggers new requirements on a system, even the best design degrades over time, leading to a phenomenon aptly termed as "architectural drift" [Pinzger 2005], "design erosion" [van Gurp and Bosch 2002], or "code decay" [Eick et al. 2001]. At a fine-grained level such a decline in quality appears in the form of "bad smells" [Fowler et al. 1999]. In the light of this degradation process, it is no wonder that maintenance and evolution claim 90% of the total software costs [Erlikh 2000].

Reengineering [Chikofsky and Cross II 1990] aims at improving the design of parts of the system to make it more capable of embracing future changes [Beck 2000]. It is however not a shotgun with which one should target *all* problematic artifacts, but must be directed towards the artifacts where such an effort is most needed. To do so, we need a means to evaluate the design of a system before taking an informed decision on which of its parts to reengineer.

One approach to assessing the quality of software design is based on detection strategies [Lanza and Marinescu 2006; Marinescu 2004], *i.e.*, metrics-based composed logical conditions, by which design fragments with specific properties are detected in the source code. The approach defines the concept of "design disharmony" by translating a set of design guidelines into detection strategies, with which design disharmonies can be discovered.

Applying detection strategies on large systems usually returns large sets of candidate artifacts (*i.e.*, classes, methods) which potentially exhibit one or more design problems. Manually analyzing the results is a cumbersome activity and it is easy to get lost in the inspection process. Moreover, design problems are not isolated, but interlinked with each other, and a list of results does not provide information about their distribution throughout a system.

The visual approaches aiming at detecting design anomalies use combinations of metrics and let the viewer correlate the outliers, which is prone to both false positives and false negatives, due to the complex nature of high-level design problems. We present an approach to visualize the software entities affected by design disharmonies, in the context of the entire system. We build on top of our previous approach to visualizing software systems, based on a 3D city metaphor [Wettel and Lanza 2007b] and centered around the concepts of software habitability and locality [Wettel and Lanza 2007a]. Such visualizations provide an observation point for a structural characterization of the entire system.

Using an approach inspired by geographical information systems, we enrich the described visualizations with results returned by a number of design anomaly detection strategies. The resulting visualizations, called *disharmony maps*, focus on the design flaws [Marinescu 2004] while maintaining the system's structural context. The main advantage of disharmony maps is that they provide an overview of the system's design and allow the viewer to mentally map the disharmony-affected entities to locations within the city.

We apply our approach on several open-source medium to large Java system, both at a coarse granularity which focuses on classes and on a finer granularity, *i.e.*, at the method level. Some of our case studies showed that correlating a set of metrics is not enough to detect higher-level design problems.

Structure of the paper. We present the design disharmonies in Section 2 and the city metaphor, on which our approach is based, in Section 3. After describing the idea behind the approach in Section 4, we apply it on several systems in Section 5. We briefly introduce our toolset in Section 6, present the related work in Section 7, then discuss our findings and conclude in Section 8.

In this article we make use of color pictures. Please read it on-screen or as a color-printed paper version.

*e-mail: richard.wettel@lu.unisi.ch
†e-mail: michele.lanza@unisi.ch

2 Design Harmony

One aspect of particular interest when analyzing a software system is the quality of its design, which influences both its comprehensibility and the required amount of maintenance over its lifetime. One approach to assessing design is centered around the concept of design harmony and its opposite, design disharmony.

2.1 Detecting Disharmonies

Design disharmonies are formalized design shortcomings to denote pieces of a system that exhibit design problems [Lanza and Marinescu 2006]. Informal design rules and guidelines [Riel 1996; Fowler et al. 1999] are transformed into detection strategies [Marinescu 2004] which are metrics-based logical conditions that detect violations against design guidelines. The antonym of design disharmony is design harmony: a software artifact is found to be harmonious when it is implemented in an "appropriate" way. This "appropriateness" is composed of three distinct harmonies that concern every software artifact:

1. *Identity harmony*, which translates to the question "How do I define myself?". Every entity in a software system must justify its existence: does it implement a specific concept and how does it do that? Is it doing too many things or nothing at all? In the context of this paper we focus on the following identity disharmonies:

   God Class is a class that performs too much on its own and does not collaborate much with other classes, but uses data from other classes.

   Brain Class is a class that accumulates an excessive amount of intelligence, usually in the form of several *Brain Methods*.

   Data Class is a "dumb" data holder class without complex functionality and on which other classes rely on.

   Brain Method is a method that tends to centralize the functionality of a class.

   Feature Envy refers to methods that seem more interested in the data of other classes than in their own data.

2. *Collaboration harmony*, which translates to the question "How do I interact with others?". Every entity collaborates with others to fulfill its tasks. Does it do that all on its own, or does it use other entities? How does it use them? Does it use too many? We focus on the following collaboration disharmonies:

   Intensive Coupling refers to a method that is tied to many other operations located in only a few classes within the system.

   Dispersed Coupling is complementary to the *Intensive Coupling* and it refers to a method which is tied to many operations dispersed among many classes throughout the system.

   Shotgun Surgery refers to the fact that a change in a method implies many changes of different methods and classes [Fowler et al. 1999].

3. *Classification harmony*, which translates to the question "How do I define myself with respect to my ancestors and descendants?". This harmony combines the two other harmonies in the context of inheritance. For example, does a subclass use all the inherited services, or does it ignore some of them? Due to space reasons we omit the presentation of these disharmonies, and refer the interested reader to [Lanza and Marinescu 2006].

Example: The *God Class* Disharmony. This design flaw, first described by Riel [Riel 1996], refers to classes that tend to incorporate an overly large amount of intelligence and whose characteristics are described by the following rules: (1) They heavily access data of simpler classes, either directly or using accessor methods; (2) They are large and complex; (3) They have a lot of non-communicative behavior, *i.e.*, there is a low cohesion between the methods belonging to that class.

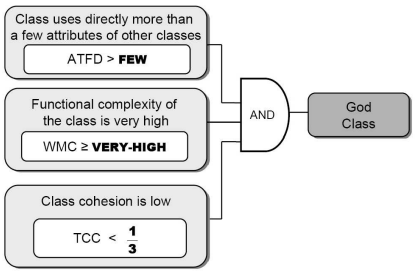

Figure 1: *The **God Class** Detection Strategy*

These informal rules can be transformed into the detection strategy depicted in Figure 1. The filtering conditions are expressed in terms of the following metrics (the left part of the expressions) and related to thresholds (the right part of the expressions):

- *Access To Foreign Data* (ATFD) represents the number of external classes whose any subset of attributes are accessed by the given class.
- *Weighted Method Count* (WMC) is the sum of the statistical complexity in a class [Chidamber and Kemerer 1994], using McCabe's cyclomatic complexity metric [McCabe 1976].
- *Tight Class Cohesion* (TCC) is the relative number of methods connected via attribute accesses [Bieman and Kang 1995; Briand et al. 1998].

3 Code Cities in a Nutshell

In the context of the EvoSpaces[1] project, which aims at exploiting multi-dimensional navigation spaces to visualize evolving software systems, several metaphors were tried [Boccuzzo and Gall 2007] to provide some tangibility to the abstract nature of software, including our *city* metaphor [Wettel and Lanza 2007b]. The main advantages of our metaphor are clear notions of locality and habitability [Wettel and Lanza 2007a], which support the viewer's orientation, as well as a structural complexity which cannot be oversimplified. As a consequence, our city metaphor has been adopted in the project's supporting tool [Alam and Dugerdil 2007].

Since we focus on object-oriented programs, we depict entities such as packages, classes, methods, attributes, and relationships such as inheritance, invocation, and access. We represent classes as buildings located in city districts which in turn represent packages, because a city, with its downtown area and its suburbs, is a familiar notion with a clear concept of orientation. Large cities are intrinsically complex constructs which can be only incrementally explored, in the same way that the understanding of a complex system increases step by step. The city artifacts with their visual properties (*e.g.*, dimensions, position, color) depict a set of properties of the software elements they represent, chosen according to the task at hand. Essentially, our aim is to represent systems as realistic cities that can be navigated and interacted with.

[1] `http://www.inf.unisi.ch/projects/evospaces`

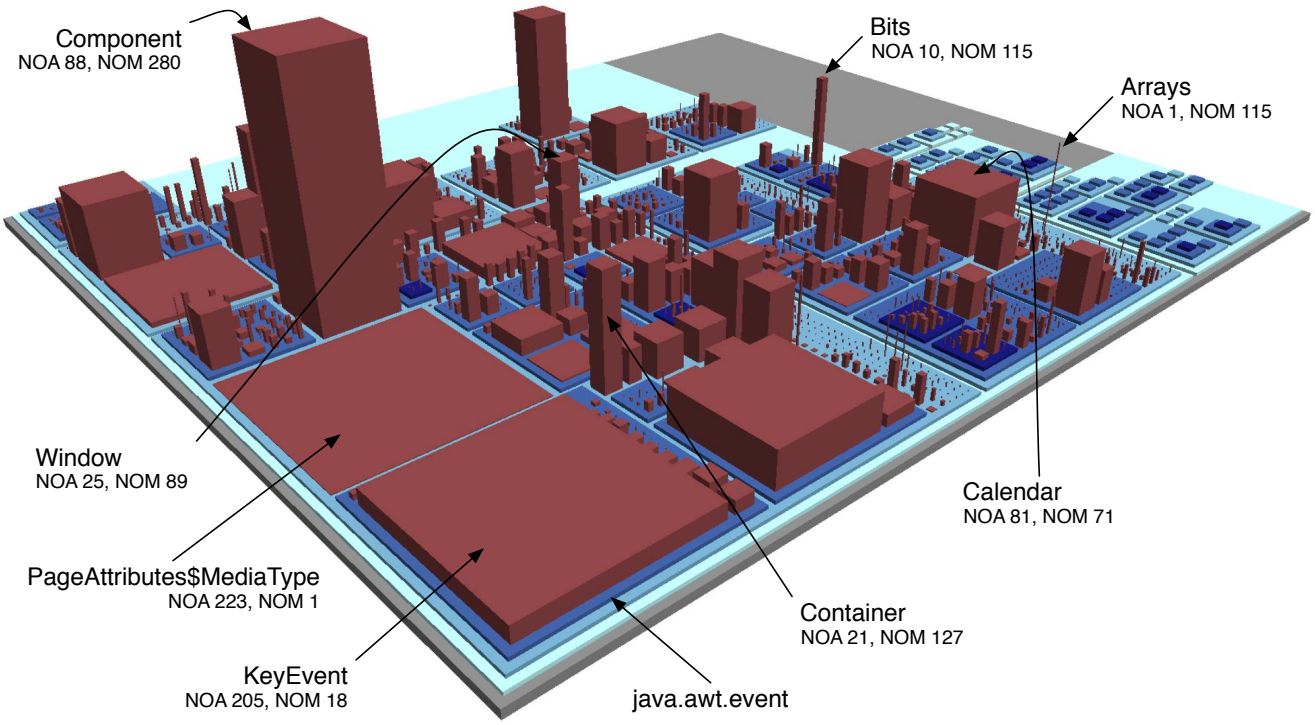

Figure 2: *Code city of JDK1.5 core (160+ kLOC)*

The user can interact with any artifact of the visualization by: (1) hovering over a figure to see information about the figure and about the model behind it, (2) opening a context-sensitive popup menu, or (3) using queries (both predefined and user-defined).

To provide a structural overview of the entire visualized system, we strive for an efficient use of the available space. A widely-used layout for hierarchical structures is the treemap [Shneiderman 1992], which incrementally splits the space into areas proportionally to a particular measure of the elements. Our layout is constrained by the fixed element dimensions (*i.e.*, as the result of the metric mapping), which boils down to solving a rectangle-packing problem. The hierarchical layout we implemented is based on *k*d-trees [Bentley 1975] and aims at minimizing the amount of wasted space.

Example. Figure 2 depicts the core of JDK 1.5 (*i.e.*, the entire java namespace). The buildings represent classes and interfaces, placed on top of tiles representing their containing packages. The height of a building represents the number of methods (NOM) of the class, the width and length represent its number of attributes (NOA). The increasing saturation of the tiles denotes the nesting level of the packages. JDK is a fairly large system with a shallow package nesting level (we count at most 4 stacked district platforms). We can also see outliers in terms of the mapped metrics. The wide and flat buildings are classes with many attributes and few methods, such as PageAttributes$MediaType and KeyEvent. The thin and tall towers, represent classes with NOM $>>$ NOA, such as Bits. There are also classes with high values for both NOM and NOA, such as Component or Calendar. Besides various outliers, we also see how functionality is distributed within packages, for example java.awt.event contains classes with a similar amount of functionality (NOM) and state (NOA), with the exception of KeyEvent, which has many more attributes (the events for each key are saved as constant attributes).

4 Disharmony Maps

To address the complexity of the results returned by the detection strategies, we integrate the information about design disharmonies, by drawing inspiration from the geographical information systems (GIS) domain. The information we are interested in is a form of multivariate data (*i.e.*, there are usually several design disharmonies affecting a system at any moment), which is similar in many ways to a number of theme map types. One such example is the disease map, in which the regions of a world map are colored according to the diseases which affected them. Such a disease map allows one to see which are the dominating diseases in the world for a particular period of time and also how they are distributed around the globe.

Similarly, we assign vivid colors to the design harmony breakers and shades of gray to the unaffected ones. This enables us to focus on the design disharmonies in the presence of a non-distracting context. We incorporate this idea into our city metaphor, which provides the concept of locality to the software elements and suits well the geographical context. The analogy to disease maps is intuitive, since the design anomalies are “diseases” affecting particular elements inside a system’s software artifact “population”. The resulting visualization, called *disharmony map*, provides a quick overview of the problems affecting the software system in terms of proportion, distribution and dominant types of design disharmonies.

In the absence of design anomaly data, conventional approaches allow us to observe outliers in terms of the mapped metrics and classify them as potential design anomalies. For example, a class with a high NOM is a potential *God Class*, or a class with many attributes and few methods is a potential *Data Class*. One advantage of disharmony maps is that they encode the disharmony information in color, allowing us to further map structural information (NOA, NOM) on the rest of the visual properties.

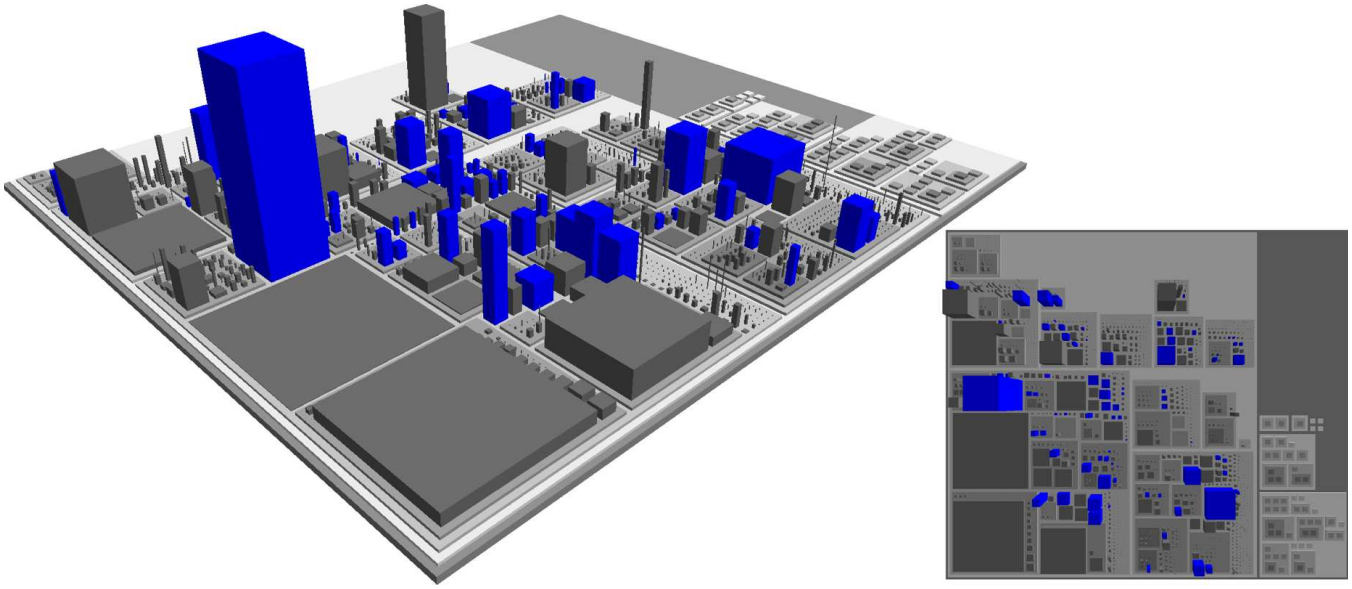

Figure 3: *City of JDK with focus on **God Classes**: isometric view (left) and top view (right)*

4.1 Design Anomalies in Context

By combining the results of design disharmony detection with our visual city metaphor, we obtain the big picture of the system's design problems, which can hardly be imagined using a non-visual, text-based approach. To illustrate this aspect, we further present the same set of results using both a textual representation and our visualization.

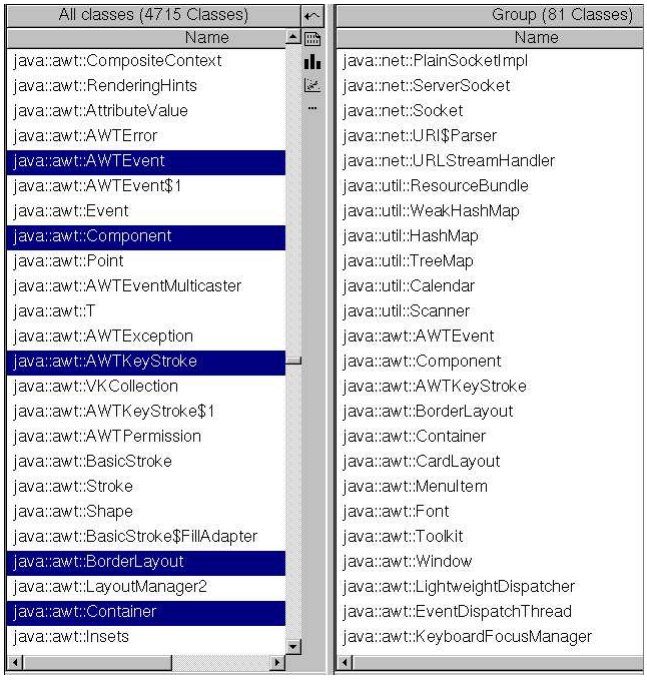

Figure 4: *God Classes in JDK core*

Running the *God Class* detection strategy on the core of JDK (Java Development Kit) returns a list of 81 affected classes out of the system's almost 5,000 classes.

With a non-visual approach, the results can be presented in the form of two lists: the result list (Figure 4, right) and the list of all the classes in the system (Figure 4, left), which serves as context. The shortcomings of such a representation are: (1) it lacks the overview, since it is impossible to look at the results as a whole and scrolling through the list induces context loss, (2) to localize the *God Classes*, one has to process the list by clustering it based on the packages in which the classes are defined and then sort the clusters based on the number of occurrences, and (3) it is completely unfeasible to correlate several disharmony types, even in the case of more effective textual representations (*e.g.*, trees).

Given the complexity of the software systems we analyze, our visual approach aims at localizing the detected disharmonious elements and present them concisely in their context, i.e., the entire system. A disharmony map depicting only the *God Class* problem in JDK is presented in Figure 3, which shows a birds-eye (on the right) and an isometric (on the left) view. Based on it, we see that the suffering classes are dispersed in many of the packages. We can also observe that not all the large classes are *God Classes* and some of the apparently less harmful ones hide their disharmony. Next, we apply our approach on a number of open-source systems and explore observing several design disharmonies in correlation.

5 Application

We applied our approach on 4 open-source Java systems: JDK (Java Development Kit), ArgoUML (UML modeling tool), Jmol (viewer for chemical structures in 3D), and iText (PDF library). In Table 1 we present the version for each system and their magnitudes in terms of lines of code, number of packages, number of classes, and number of methods.

System	version	Lines	Packages	Classes	Methods
JDK	v. 1.5	160'287	137	4'715	19'379
ArgoUML	v. 0.24	144'523	142	2'468	14'692
Jmol	r. 8065	84'984	105	1'032	7'751
iText	r. 2892	80'389	149	1'250	7'182

Table 1: *Systems under study*

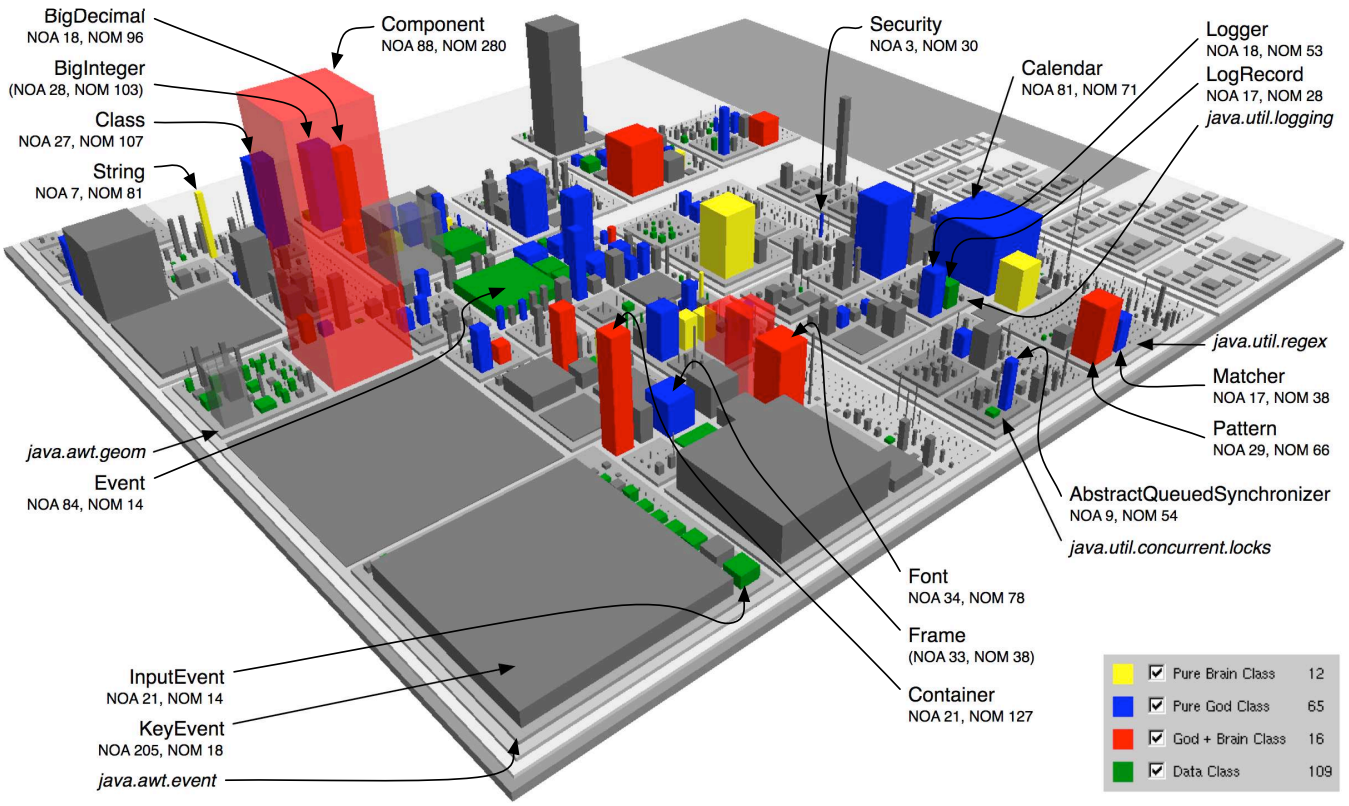

Figure 5: *Class-level disharmonies in JDK*

5.1 Class-Level Disharmonies

We encode each disharmony in a different color: yellow for *Brain Class*, blue for *God Class*, red for *Brain & God Class*, and green for *Data Class*. For better visibility, in the case of large buildings obstructing other buildings relevant to the discussion, we manually set their transparency (user-modifiable) to 40 %. Each visualization shows a legend, presenting the targeted disharmonies, and for each of them the assigned color and the number of affected entities.

JDK. Before diving into details, the first impression we get by looking at the overview of JDK (Figure 5) is that the system looks well-organized, in spite of the numerous disharmonious artifacts: we see green districts, where mostly *Data Classes* are localized and districts of increased complexity, in which several *God Classes* and *Brain Classes* are defined.

An interesting district is java.awt.event, made of one wide and flat building, representing the class KeyEvent and many small classes, all representing other events (*e.g.*,InputEvent). Although by looking at the properties of the classes one would be tempted to categorize KeyEvent as a *Data Class* due to its 205 attributes and only 18 methods, it actually is one of the few classes in this package which is *not* affected by the disharmony. This is due to the fact that not only there are non-accessors among the 18 methods of this class, but some of them are quite complex.

The "green" district (to the right of the large red building) representing package java.awt.geom seems to have grouped a fair number of the 109 *Data Classes* in JDK. The superclass of all these classes is the large *Data Class* Event (84 attributes, 14 methods), defined in the parent package java.awt.

Many of the classes that are both *God* and *Brain Classes* (*i.e.*, depicted by red buildings) are defined in the java.awt package, which handles the core graphics part in Java: Component (the dominating building in the city, due to the class's 88 attributes and 280 methods), Container, or Font. Moreover, some of the core classes belonging to Java's type system are either *Brain Classes*, (*e.g.*, String), *God Classes* (*e.g.*, BigInteger, Class, Calendar), or both (*e.g.*, BigDecimal).

Some of the *God Classes* are easy to overlook in the absence of disharmony data, *e.g.*, class Security with its only 3 attributes and 30 methods, which obviously encodes some complex encryption algorithms. The same holds for class AbstractQueuedSynchronizer residing in package java.util.concurrent.locks, whose complexity is suggested by its very name. Our approach allows us to complement the structural information of the elements with the actual disharmony data, revealing even the subtle design anomalies.

An interesting package is java.util.regex with its share of complexity in the form of *God Class* Matcher and *God & Brain Class* Parser, which practically accumulate the entire intelligence of the package, used for the processing of regular expressions. This is illustrated by a district containing two rather large "corporate" buildings surrounded by small houses.

Package java.util.logging illustrates another pattern, a *God Class* together with the *Data Class* it misuses: Logger (18 attributes, 53 methods), and LogRecord (17 attributes, 28 methods), a *Data Class* in spite of its many methods. To verify this hypothesis, we inspected the relations of the involved classes, which revealed that more than one half (48 out of 86) of the statically-determined invocations of class LogRecord's methods (most of which are getters and setters) are performed by class Logger.

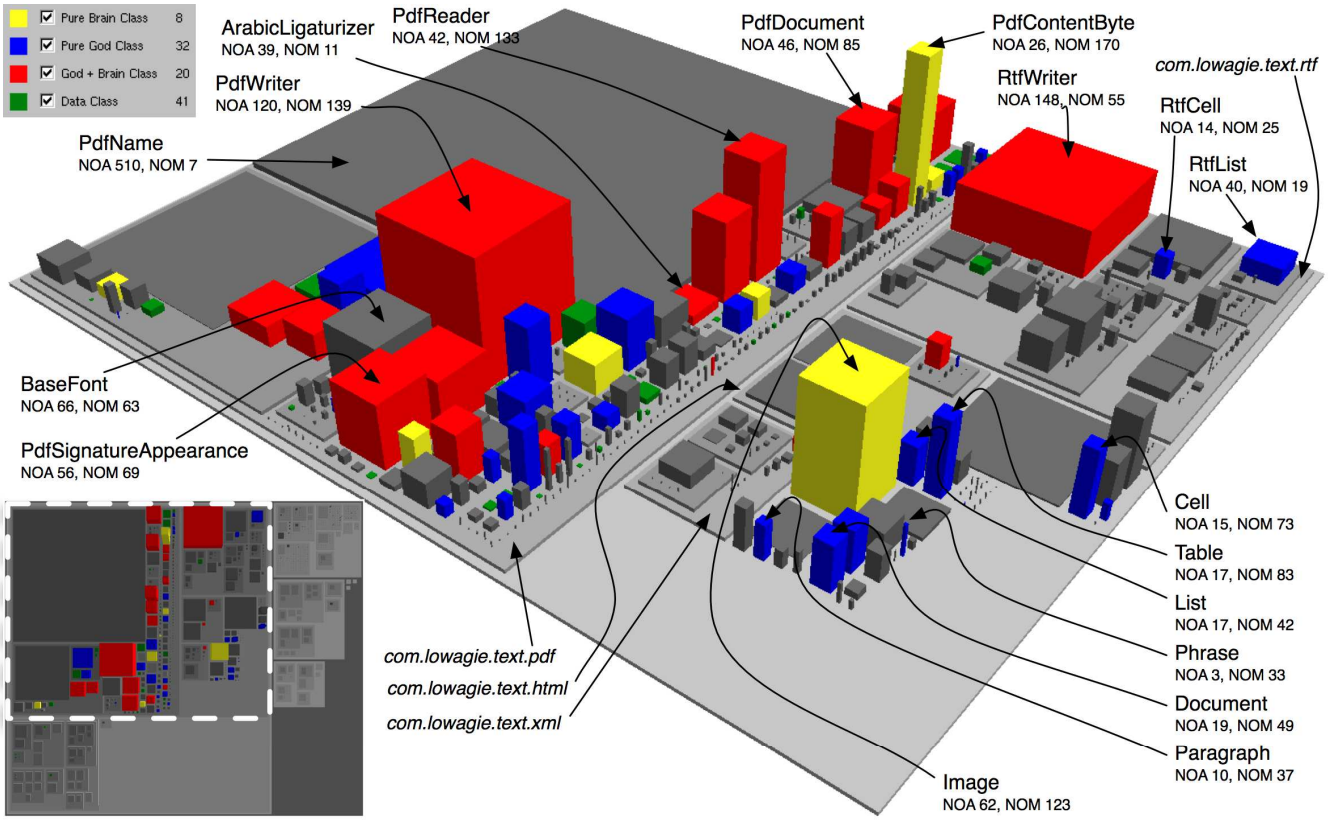

Figure 6: *Class-level disharmonies in iText*

iText. The first impression given by the overview of iText is one of a rather bulky system, with a large number of outlying classes. The system seems to lack organization and the disharmonies are chaotically spread all over it. The dominating colors in the disharmony map reveal many problems: 28 *Brain Classes*, 52 *God Classes* (20 of which affected by both disharmonies), and 41 *Data Classes*.

The lower-left part of Figure 6 shows a birds-eye view of the system, composed of 3 library packages (right part of the view), the examples package com.lowagie.examples (lower part of the view) and the core package com.lowagie.text (the central part, delimited by the perimeter).

This core package also appears as a detailed isometric view in the center of Figure 6 with annotated entities. The package contains several subpackages, one for each file format: com.lowagie.text.xml, com.lowagie.text.html, com.lowagie.text.rtf, and com.lowagie.text.pdf.

The com.lowagie.text.pdf package is vast, with 239 classes (out of which 61 affected by at least one class-level disharmony) and only 8 subpackages, each with just a few defined classes. With that many classes defined in it, this single package has grown into a module which is difficult to understand and manage, a fact reflected by the over one quarter of disharmonious classes.

The system contains hierarchies spreading over the file-format specialized packages (*i.e.*,xml, html, rtf, and pdf), whose base classes are defined in the main package com.lowagie.text. Among these base classes there are many *God Classes* (*e.g.*,Cell, Table, List, Phrase, Document, Paragraph), all annotated on Figure 6 (right part).

In package com.lowagie.text.rtf, we see some examples of "hereditary" disharmony, illustrated by *Brain & God Class* RtfWriter, and *God Classes* RtfCell and RtfList, all disharmonious like their superclasses. The most striking harmony breakers reside in the com.lowagie.text.pdf package, in which the red color dominates, due to the large number of *Brain & God Classes*, such as PdfWriter (120 attributes, 133 methods), PdfReader (42 attributes and 133 methods), or PdfDocument (46 attributes, 85 methods).

Another remarkable phenomenon comes in the form of the apparently tiny buildings affected by design disharmonies that imply an increased complexity (*i.e., God Class* or *Brain Class*). Inspecting one of these classes, called Phrase reveals that its scale is reduced only in the context of the iText system, as NOM=33 is a value considered very large for a Java class [Lanza and Marinescu 2006]. The disharmony map indicates it as a *God Class* and thus does not allow the maintainers of the system to overlook this potentially problematic class.

During our experiments we noticed that there is no evident link between the simple metric values, such as the NOA and NOM metrics for a class and the disharmonies affecting it. The first

One example that confirms this is given by two classes with more or less the same magnitude in terms of the NOM and NOA metrics, yet which are total opposites: While BaseFont (66 attributes and 63 methods) appears as a healthy class with respect to the class-level harmony, PdfSignatureAppearance (56 attributes, 69 methods) is a *God & Brain Class*, due to the complexity of its methods and the way it collaborates with other classes. Another example is class ArabicLigaturizer, (39 attributes, 11 methods) which contains enough complexity in its few methods to qualify as a *God & Brain Class*.

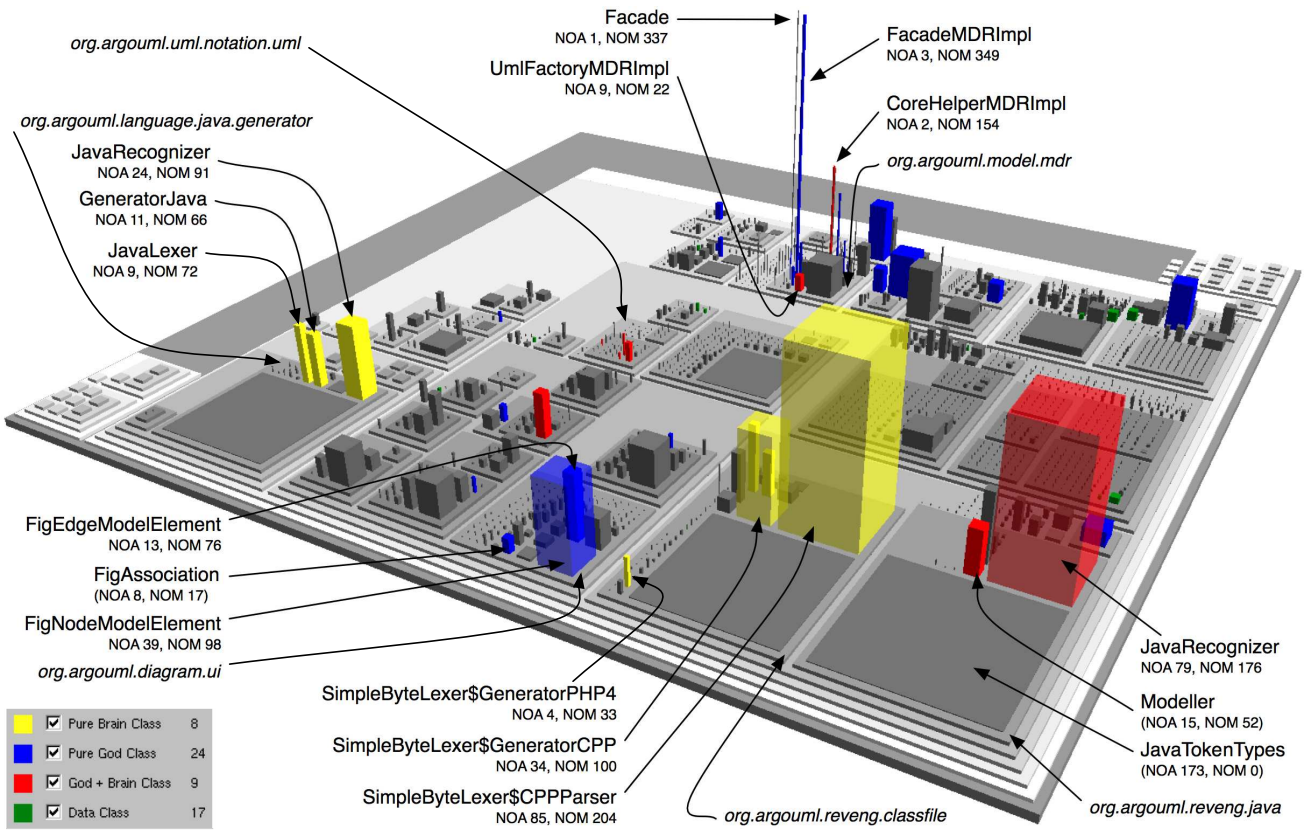

Figure 7: *Class-level disharmonies in ArgoUML*

ArgoUML. ArgoUML has 17 *Brain Classes* and 33 *God Classes* (of which 9 classes affected by both disharmonies), and 17 *Data Classes*, which are not distributed all over the system, but rather sparsely, as Figure 7 shows.

Our attention is drawn to 3 similar formations, each composed of 1 wide, flat building and 2-3 massive neighbor buildings. The first one resides in the org.argouml.reveng.java district, and is made of the huge red building (*i.e.*, *Brain & God Class* JavaRecognizer), a smaller red building (*i.e.*, class Modeller), and a wide and flat building which looks like a parking lot (*i.e.*, class JavaTokenTypes with 173 attributes and no methods). Although we would expect the latter to be a *Data Class* it is *not*, because all its attributes are declared as final public, *i.e.*, they are pure Java constants.

The second similar package is org.argouml.reveng.classfile, with two *Brain Classes*: the city's dominating building, class CPPParser with 85 attributes and 204 methods and the smaller affected one, class GeneratorCPP (34 attributes and 100 methods), which are both inner classes defined in SimpleByteLexer. An inner class of the same class is the "parking lot" representing class STDCTokenTypes with 152 attributes and no methods, which serves as the repository for constants for the C++ parsing. Another example of elusive *Brain Class*, revealed only due to the disharmony data, is GeneratorPHP4 with its only 4 attributes and 33 methods.

The third similar package is org.argouml.language.java.generator, on the left of the picture. It contains three *Brain Classes*: JavaRecognizer (24 attributes, 91 methods), GeneratorJava (11 attributes, 66 methods), and JavaLexer (9 attributes, 72 methods). As in the first package, the constants repository is also called TokenTypes (146 attributes). As reported in [Wettel and Lanza 2007a], having the same code twice (the two JavaTokenTypes have almost 150 identical constants) is questionable, yet less harmful in the case of generated classes, which are not manually maintained.

By contrast, the *God Classes* FigNodeModelElement (39 attributes, 98 methods), FigEdgeModelElement (13 attributes, 76 methods) and FigAssociation (8 attributes, 17 methods), located in org.argouml.uml.diagram.ui, are core classes and thus, very likely to be subject to continuous maintenance and change requirements.

Another disharmonious agglomeration is a district characterized by a "forest" of very thin and extremely tall buildings (few attributes and many methods), representing package org.argouml.model.mdr. Out of its 35 classes, 8 are *God Classes* and 2 are *God & Brain Classes*. The doubly-affected classes are UmlFactoryMDRImpl (9 attributes, 22 methods) and CoreHelperMDRImpl (2 attributes, 154 methods). The largest affected class of this package, depicted by a building that literally touches the sky, is the *God Class* FacadeMDRImpl (3 attributes, 349 methods). All these classes are the only implementations of the interfaces UmlFactory, CoreHelper, and Facade, respectively. In spite of their large number of methods, the interfaces are not affected by disharmonies due to their lack of defined functionality. However, perceived through their implementing classes, these apparently harmless interfaces qualify as mechanisms for building *God Classes* and *Brain Classes*. By analyzing ArgoUML's history (not in the scope of this paper), we learned that the Facade interface (and by analogy the other interfaces in the hierarchy) had two concrete implementations in one of the versions of the system. Some versions later, one of the implementations disappeared, leaving the MDR implementations as the only one until these days. This problematic package is also one of the case studies for the method-level disharmony maps, presented next.

Finally, we observed other hardly visible colored buildings in district org.argouml.uml.notation.uml, representing the *God & Brain Classes* NotationUtilityUML (NOA 6 , NOM 24), MessageNotationUML (NOA 2, NOM 29), AttributeNotationUML (NOA 2, NOM 8), and OperationNotationUML (NOA 0, NOM 9). Since both disharmonies require high complexity, it was unexpected to find these apparently low-functional classes (*i.e.*, reduced height) among the affected. To our surprise, these classes privately held the following amounts of code expressed in LOC: 1240, 1538, 432, and 450, respectively. These classes were not programmed in the object-oriented spirit and should be reviewed by ArgoUML's maintainers.

5.2 Method-Level Disharmonies

To visualize method-level disharmonies, we use a finer-grained representation which extends the previous one by explicitly depicting the methods, combined with a specific layout. Stepping away from the monolithic block representation, we now depict a class as one base platform on top of which we stack up vertically, in layers of 4, the set of "bricks" representing its methods (See close-up detail in Figure 8). Besides the increased level of detail it provides, this representation allows for user interaction down to the method level. The height of the class representation is still proportional to the number of methods. Due to the fact that looking at the entire system using this granularity is impractical (*i.e.*, too many depicted entities), we focus on specific parts of the systems.

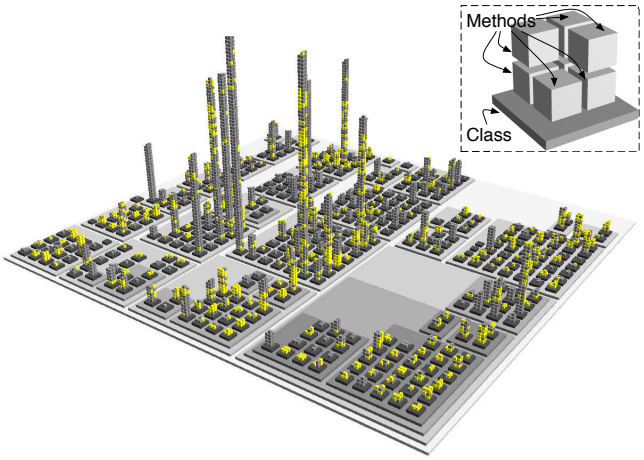

Figure 8: *Yellow-colored* ***Feature Envy*** *in Jmol and a close-up detail of class with 7 methods laid out as bricks (top right)*

Visualizing Jmol using the *bricks view* (see Figure 8) reveals the fact that more than one quarter of the methods in Jmol (*e.g.*, 1'555 out of 5'968) exhibits the *Feature Envy* disharmony. Our visualization depicts the Feature Envy "epidemic" in a suggestive way and the picture says it all: the system needs to be quarantined right away for a serious session of reengineering.

However, classes with an extremely high number of methods (*e.g.*, hundreds) are represented as overly tall buildings, which for an overview pushes the viewer far away in order to comprise every entity and starting with certain distances, the details are too small to be useful. To address this issue, we devised an adaptive vertical layout called *Progressive Bricks*, by considering the *Bricks* layout as a particular case in which the walls of the buildings are 2-bricks wide, and then generalizing it. The width of the wall (in number of bricks) is adapted to the number of bricks of the building in order to obtain reasonable heights. The magnitude of the class (in terms of the NOM metric) is expressed this time by the building's volume.

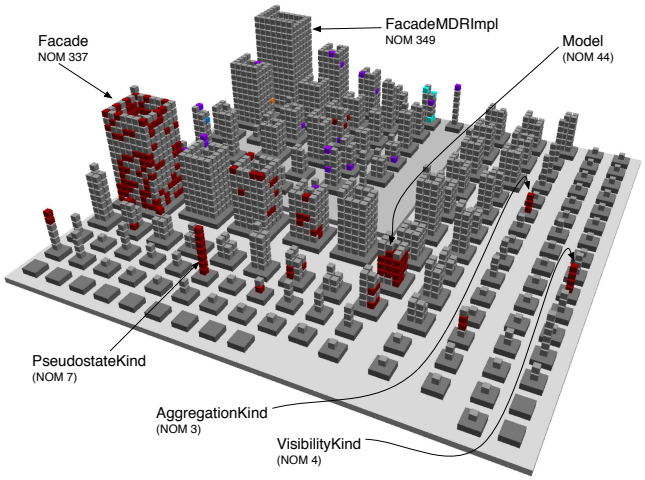

Figure 9: *Red-colored* ***Shotgun Surgery*** *in* org.argouml.model

Figure 9 shows a visualization of package org.argouml.model using the *Progressive Bricks* adaptive layout. This package, which was subject to discussions also during the class-level analysis, contains in this representation the two most massive buildings in the city of ArgoUML (*i.e.*, the highest number of methods), representing the interface Facade and a class that implements this interface, called FacadeMDRImpl. Displaying all the method-level disharmonies of this package reveals the fact that the dominating disharmony characterizing this package is *Shotgun Surgery*, depicted by the many buildings "tainted" with the dark red color. Moreover, we see that a large amount of the dark red "bricks" (representing methods) belong to only a reduced set of classes. The largest building affected by this disharmony is the Facade interface. In contrast to the class-level disharmonies discovered in this package, the method-level disharmonies are detected on the interface and not on the classes implementing it, due to the fact that the calls are done using polymorphism, *i.e.*, they are targeting a reference to the interface. Out of the 337 methods defined in Facade, 140 exhibit the *Shotgun Surgery* disharmony. Apart from this interface, the Model interface has many methods with Shotgun Surgery (28 out of 44) and 3 small classes have only methods with Shotgun Surgery: AggregationKind (3), VisibilityKind (4), and PseudostateKind (7), respectively. This disharmony is somewhat expectable in this package, since it is part of the system's model and all the other modules depend on it. A class with an increased number of methods affected by *Shotgun Surgery* is fairly difficult to change, since any change is likely to require many changes throughout the system.

6 Tool Support

We built our tool called CodeCity on top of the Moose [Ducasse et al. 2005] framework which provides, among others, an implementation of the FAMIX [Demeyer et al. 2001] language-independent meta-model for object-oriented software. For the parsing of Java systems we use iPlasma [Marinescu et al. 2005].

CodeCity is written in Smalltalk and uses OpenGL for rendering the visualizations. Besides configuring the view in terms of glyphs, property mappings, and layouts, the user can spawn secondary views, or interact with the artifacts (*e.g.*, querying, color tagging).

CodeCity can visualize systems written in different languages (*e.g.*, Java, Smalltalk, C++), runs on any major platform (*e.g.*, Windows, Mac OS X, Linux), and is freely available for download at: `http://www.inf.unisi.ch/phd/wettel/codecity.html`.

7 Related Work

Since the early days of software visualization, software has been visualized at various levels of detail, from the module granularity seen in Rigi [Muller and Klashinsky 1988] to the individual lines of code depicted in SeeSoft [Eick et al. 1992].

Besides our metaphor, most software visualizations which target the quality of software depict the software artifacts in terms of a set of software metrics. In the polymetric views [Lanza and Ducasse 2003], software elements are visualized as rectangles with metrics mapped on their position, dimensions and color. Kiviat diagrams [Pinzger et al. 2005] depict evolving software entities over several versions in terms of large sets of software metrics.

Apart from the metrics aspect, our visual representation is related to a number of previous works in 3D, detailed in the following.

The increase in computing power over the last 2 decades enabled the use of 3D metric-based visualizations, which provides the means to explore more realistic metaphors for software representation. One such approach is poly cylinders [Marcus et al. 2003], which makes use of the third dimension to map more metrics. As opposed to this approach in which the representations of the software artifacts can be manipulated (*i.e.*, moved around), our code cities imply a clear sense of locality which helps in viewer orientation. Moreover, our approach provides an overview of the hierarchical (*i.e.*, package) structure of the systems.

The value of a city metaphor for information visualization is proven by papers which proposed the idea, even without having an implementation. [Santos et al. 2000] proposed this idea for visualizing information for network monitoring and later [Panas et al. 2003] proposed a similar idea for software production. Among the researchers who actually implemented the city metaphor, [Knight and Munro 2000; Charters et al. 2002] represented classes are districts and the methods are buildings. Apart from the loss of package information (*i.e.*, the big picture), this approach does not scale to the magnitude of today's software systems, because of its granularity. We owe the scalability of our approach to a more appropriate mapping between the software world and the city environment, to a configurable granularity, and to the computing power available today.

Other 3D geographically-inspired visualizations include the software landscapes [Balzer et al. 2004], which reduces the visual complexity through an incremental level of detail (*i.e.*, the transparency of the container artifacts is adjusted to the distance of the viewpoint). The approach lacks system overview capabilities, because the loose, navigation-targeted layout does not scale for it, even in the presence of complete transparency of the containers. A 3D approach, which produces layouts similar to ours, is found in information pyramids [Andrews et al. 1997], applied to file systems.

As for the visualization of design anomalies, to our best knowledge, all the previous work depicts software artifacts in terms of a reduced set of low-level metrics and does not address the visual representation of such high-level design problems. Although one can correlate the outliers (*i.e.*, extreme values) for a particular metric set to reveal *potential* candidates for a particular design anomaly, these oftentimes include false positives and false negatives, as some of our examples in Section 5 illustrate.

The 3D visual approach closest in focus to ours is [Langelier et al. 2005], which uses boxes to depict classes and maps software metrics on their height, color and twist. The classes' box representations are laid out using either a modified treemap layout or a sunburst layout, which split the space according to the package structure of the system. The authors address the detection of design principles violations or anti-patterns by visually correlating outlying properties of the representations, *e.g.*, a twisted and tall box represents a class for which the two mapped metrics have an extremely high value. Besides false positives and negatives, the drawbacks of this approach is that one needs different sets of metrics for each design anomaly and the number of metrics needed for the detection oftentimes exceeds the mapping limit of the representation (*i.e.*, 3).

The detection strategies [Marinescu 2004] were introduced as a mechanism to formulate complex rules using the composition of metrics-based filters, and extended later [Lanza and Marinescu 2006] by formalizing the detection strategies and providing aid in recovering from detected problems. The 2D polymetric views provided as means to visualize the systems do not explicitly illustrate the disharmonious artifacts, nor do they provide an overview and distribution of the disharmonies within the observed systems. The non-visual approach of [Raţiu et al. 2004] aims at further improving the design flaw detection by taking into account information from the suspect classes' evolution to compute their persistency in exhibiting a particular design flaw during their lifetime.

8 Discussion & Conclusions

We extended our previous work on 3D software visualization [Wettel and Lanza 2007a; Wettel and Lanza 2007b], by enriching it with data on the quality of the system's design. We obtain this information by running detection strategies [Marinescu 2004] to reveal design disharmonies [Lanza and Marinescu 2006]. Drawing inspiration from the field of geographical information systems, we use a gray color scheme for unaffected artifacts and strong colors for affected ones, according to the disharmonies they exhibit.

With our case studies, we showed that using a pure metric-based visualization to assess the design problems of a system is prone to false results, because an outlier is not necessarily a disharmonious entity, as our counterexamples have proved. These false "first impressions" are due to the fact that each design disharmony is defined as a complex expression of a set of metrics. To have a more informed first impression without using the results of the detection strategies would require mapping a large amount of metrics on each visualization, which is not feasible due to the reduced amount of visual properties that the human eye and brain are able to grasp. However, looking only at a list of results of running the detection strategies against a subject system does not provide enough context for the assessment of its design's quality. Neither does visualizing the system using only structural information. An overview of disharmonies and their distribution throughout the system, supported by the locality of our metaphor, helps in building a mental image of the design problems within the system.

The main contribution of this paper is an effective integration of the design anomaly data with our visual approach based on a 3D city metaphor, which provides both the big picture of the system's design problems and the means to further investigate the details.

We applied our approach on 4 Java systems, and were able to learn about false appearances, visualize patterns of disharmonies, observe how a bad organization of the package structure is accompanied by many disharmonies of its classes, or how disharmonies can conquer a system in the absence of reengineering. As future work, we plan to perform an evaluation of our approach's effectiveness.

Acknowledgements

We gratefully acknowledge the financial support of the Hasler Foundation for the project "EvoSpaces" (Hasler Foundation MMI Project No. 1976). We thank the European Smalltalk User Group (`http://esug.org`) and CHOOSE (`http://choose.s-i.ch`) for travel sponsorships.

References

ALAM, S., AND DUGERDIL, P. 2007. Evospaces visualization tool: Exploring software architecture in 3d. In *Proceedings of WCRE 2007*, IEEE CS Press, 269–270.

ANDREWS, K., WOLTE, J., AND PICHLER, M. 1997. Information pyramids: A new approach to visualising large hierarchies. In *Proceedings of VIS 1997*, IEEE CS Press, 49–52.

BALZER, M., NOACK, A., DEUSSEN, O., AND LEWERENTZ, C. 2004. Software landscapes: Visualizing the structure of large software systems. In *Proceedings of VisSym 2004*, Eurographics Association, 261–266.

BECK, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison Wesley.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9, 509–517.

BIEMAN, J., AND KANG, B. 1995. Cohesion and reuse in an object-oriented system. In *Proceedings of the ACM Symposium on Software Reusability*, ACM Press.

BOCCUZZO, S., AND GALL, H. C. 2007. Cocoviz: Towards cognitive software visualizations. In *Proceedings of VISSOFT 2007*, IEEE CS Press, 72–79.

BRIAND, L. C., DALY, J. W., AND WÜST, J. 1998. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal 3*, 1, 65–117.

CHARTERS, S. M., KNIGHT, C., THOMAS, N., AND MUNRO, M. 2002. Visualisation for informed decision making; from code to components. In *Proceedings of SEKE 2002*, ACM Press, 765–772.

CHIDAMBER, S. R., AND KEMERER, C. F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering 20*, 6 (June), 476–493.

CHIKOFSKY, E., AND CROSS II, J. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software 7*, 1, 13–17.

DEMEYER, S., TICHELAAR, S., AND DUCASSE, S. 2001. FAMIX 2.1 — The FAMOOS Information Exchange Model. Tech. rep., University of Bern.

DUCASSE, S., GÎRBA, T., AND NIERSTRASZ, O. 2005. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, 99–102.

EICK, S. G., STEFFEN, J. L., AND ERIC E., JR., S. 1992. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering 18*, 11 (Nov.), 957–968.

EICK, S., GRAVES, T., KARR, A., MARRON, J., AND MOCKUS, A. 2001. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering 27*, 1, 1–12.

ERLIKH, L. 2000. Leveraging legacy system dollars for e-business. *IT Professional 2*, 3, 17–23.

FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., AND ROBERTS, D. 1999. *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

KNIGHT, C., AND MUNRO, M. C. 2000. Virtual but visible software. In *Proceedings of IV 2000*, IEEE CS Press, 198–205.

LANGELIER, G., SAHRAOUI, H. A., AND POULIN, P. 2005. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of ASE 2005*, ACM Press, 214–223.

LANZA, M., AND DUCASSE, S. 2003. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE) 29*, 9 (Sept.), 782–795.

LANZA, M., AND MARINESCU, R. 2006. *Object-Oriented Metrics in Practice*. Springer-Verlag.

MARCUS, A., FENG, L., AND MALETIC, J. I. 2003. 3d representations for software visualization. In *Proceedings of SoftVis 2003*, ACM Press, 27–36.

MARINESCU, C., MARINESCU, R., MIHANCEA, P. F., RATIU, D., AND WETTEL, R. 2005. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of ICSM 2005, Industrial & Tool Volume*, IEEE CS Press, 77–80.

MARINESCU, R. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM 2004*, IEEE CS Press, 350–359.

MCCABE, T. 1976. A measure of complexity. *IEEE Transactions on Software Engineering 2*, 4 (Dec.), 308–320.

MULLER, H., AND KLASHINSKY, K. 1988. Rigi: a system for programming-in-the-large. In *Proceedings of ICSE 1988*, ACM Press, 80–86.

PANAS, T., BERRIGAN, R., AND GRUNDY, J. 2003. A 3d metaphor for software production visualization. In *Proceedings of IV 2003*, IEEE CS Press, 314.

PINZGER, M., GALL, H., FISCHER, M., AND LANZA, M. 2005. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005*, ACM Press, 67–75.

PINZGER, M. 2005. *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology.

RAŢIU, D., DUCASSE, S., GÎRBA, T., AND MARINESCU, R. 2004. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004*, IEEE CS Press, 223–232.

RIEL, A. 1996. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA.

SANTOS, C. R. D., GROS, P., ABEL, P., LOISEL, D., TRICHAUD, N., AND PARIS, J. P. 2000. Mapping information onto 3d virtual worlds. In *Proceedings of IV 2000*, 379–386.

SHNEIDERMAN, B. 1992. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph. 11*, 1, 92–99.

VAN GURP, J., AND BOSCH, J. 2002. Design erosion: problems and causes. *Journal of Systems and Software 61*, 2, 105–119.

WETTEL, R., AND LANZA, M. 2007. Program comprehension through software habitability. In *Proceedings of ICPC 2007*, IEEE CS Press, 231–240.

WETTEL, R., AND LANZA, M. 2007. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007*, IEEE CS Press, 92–99.