

# Diagnosing Design Problems in Object Oriented Systems

Adrian Trifu  
FZI Forschungszentrum Informatik  
Haid-u-Neu Str. 10-14  
76131 Karlsruhe, Germany  
trifu@fzi.de

Radu Marinescu  
LOOSE Research Group  
“Politehnica” University, B-dul V. Pârvan 2  
300223 Timișoara, Romania  
radum@cs.utt.ro

## Abstract

*Software decay is a phenomenon that plagues aging software systems. While in recent years, there has been significant progress in the area of automatic detection of “code smells” on one hand, and code refactorings on the other hand, we claim that existing restructuring practices are seriously hampered by their symptomatic and informal (non-repeatable) nature. This paper makes a clear distinction between structural problems and structural symptoms (also known as code smells), and presents a novel, causal approach to restructuring object oriented systems. Our approach is based on two innovations: the encapsulation of correlations of symptoms and additional contextual information into higher-level design problems, and the univocal, explicit mapping of problems to unique refactoring solutions. Due to its explicit, repeatable nature, the approach shows high potential for increased levels of automation in the restructuring process, and consequently a decrease in maintenance costs.*

## 1. Introduction

According to a number of studies [12, 7, 8], maintenance activities account for more than 70% of the total costs associated with the life cycle of software systems. A significant part of maintenance efforts is concerned with fighting the phenomenon called software decay [9] (also known as design drift [1]).

The last decade has brought important progress in the area of object oriented restructuring, especially with the advent of design patterns [10], refactorings [9], as well as technology for automatic detection of “code smells” [4, 17]. In spite of this progress, restructuring large object oriented systems still remains a predominantly manual, time-consuming, risky process, that involves extensive (and costly) human expertise (both domain-specific and technical). In other words, restructuring is still an “art”, rather

than engineering, an aspect which is underlined by studies showing that approximately 50% of maintainer time is spent exclusively on understanding the code [22].

Apart from the obvious overhead due to the need of picking up the semantics buried in the design, a significant amount of time is dedicated to deciding where and how to refactor. The reason behind this is the symptomatic nature of current “code smell”-based approaches, which we claim, seriously hampers the restructuring process. In other words, design flaws as understood today, capture nothing more than purely structural properties which characterize the “shape” in which a given design entity currently is (e.g. *data class*, *god class*, *feature envy*, *duplicated code* [9, 19] etc).

While such information can certainly convey a suggestive picture about the state of a system (quality assessment), it is insufficient for determining a meaningful course of action regarding the affected entities. Each case must be analyzed and treated individually. During this process, the engineer looks at the environment (context) of the affected entity and tries to identify the real cause that induced the code smell.

In the case of “god class” for example, this can be excessive intra-class code duplication (copy-paste-adapt programming), or the fact that the class encapsulates more than one cohesive concept, or even the fact that the class simulates a non-existent hierarchy of classes using switch or if-else statements. All of these situations can potentially lead to what we associate with the informal notion of a god class: a large, overly complex, non-cohesive class, with ties to many parts of the system.

In order to convey this idea clearly, we will make an analogy with the medical world. In the medical world, a disease is diagnosed based on the presence of a specific constellation of symptoms. Therapies are reusable procedures describing how to cure a disease. They are described in medical textbooks in a pattern-like form. Most importantly, they are causal, and not symptomatic: you will find a cure for smallpox, rather than skin rash, fever or headache.

Coming back to software systems, we believe it is ex-

tremely hard (if not impossible) to distinguish problems and symptoms in absolute terms. As it is in medicine too, any problem can be considered a symptom at a higher level of abstraction (i.e. a disease can eventually turn out to be the symptom of a yet undiscovered or not well enough understood illness). The point is, we need to define our design “problems” on the level that best suits our needs, and that is the level which allows us to formulate a unique strategy for treatment. A possible explanation why existing approaches fail to do this lies in their originally intended purpose: assessing the structure of a system, and not improving the structure of a system.

The main goals of the present paper are the following:

- to clearly separate relevant terminology, such as *problem*, *symptom* and *treatment* to the particular field of object-oriented restructuring;
- to propose a *causal* approach to object oriented reengineering by defining a *diagnosis* process that can lead us beyond structural symptoms, to the underlying cause. This will potentially increase the usefulness of currently available quality models, by encapsulating particular correlations between structural symptoms (code smells) and other contextual information into *named* entities (the problems), which are able to shed light onto the reasons behind a bad structure. In addition, for each design problem, a precise, causal and contextual treatment can be recommended;
- to make the transition from symptoms to problems and from problems to treatments *explicit*, by capturing them in the reusable form of patterns. This goes along the same line as the pattern movement: as long as these mappings are not made explicit, the diagnosis and treatment are not repeatable... making them an art, rather than engineering activities. Additionally, this type of formalization creates the necessary premises for increased automation in the decision making process, leading to a potentially important decrease in maintenance costs.

The paper is structured as follows: section 2 defines terminology and discusses the approach in detail, followed by a few examples of design problems and corresponding treatments in section 3. Sections 4 and 5 are dedicated to presenting a case study on a well known open source system, while section 6 gives a short overview of related research. Finally, we conclude in section 7.

## 2. Approach

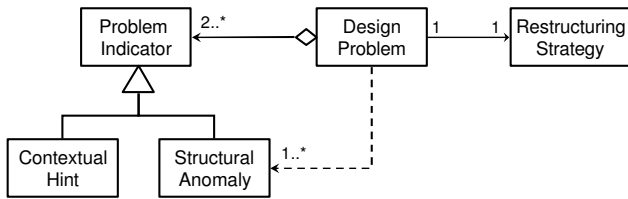
The main idea of our approach is based on the observation that different types of code smells often occur together

in certain configurations. Based on this observation we formulate the hypothesis that the reason for these correlations is the common underlying cause, the design problem, and the code smells are the structural symptoms of the problem.

For example, the design problem that we call “*inheritance for implementation reuse*” (see section 3.3) refers to the situation in which a programmer abusively uses the inheritance mechanism of the programming language to reuse parts of the implementation of the superclass, without any intention of specializing it. In fact, the subclass might model something completely different, with no relation to the semantics of the superclass. This is what we refer to as a concrete design problem, for which the treatment consists of replacing the inheritance with composition. Structural properties (code smells) such as: *refused parent bequest* in the subclass [9], or subclass is a *tradition breaker* (see section 3.3), are symptoms that are likely to appear together, whenever we have an instance of the problem.

For the purposes of this paper, we define the following terminology:

- *Structural anomaly, or symptom*: is a measurable, abnormally large deviation of a certain structural aspect of the source-code from a set of largely accepted norms, characteristic for a given design paradigm. The equivalent of Fowler’s code smells, anomalies can be automatically detected using detection strategies [17];
- *Contextual hint*: is a directly identifiable property of a design entity or fragment, with no negative connotation, and having either a structural or semantic nature, that is used in diagnosing a given design problem. For example, a semantic contextual hint for the “*inheritance for implementation reuse*” problem mentioned above could be that the subclass uses multiple inheritance. Contextual hints and structural anomalies are collectively referred to as *problem indicators*;
- *Design problem*: is a concrete, “bad” design decision that contradicts some commonly accepted design practices, with respect to well determined, reoccurring situations. An instance of a design problem is diagnosed based on the presence of a unique constellation of correlated *problem indicators* (anomalies and contextual hints), and admits a unique treatment.
- *Diagnosis*: is the process of uniquely identifying a design problem;
- *Restructuring strategy, or treatment*: is a precisely defined procedure that describes a precise sequence of operations that need to be carried out in order to eliminate an instance of a given design problem. We distinguish between “*symptomatic treatment*” (one that tries to identify a refactoring or sequence of refactorings based on an isolated anomaly, and without regard



**Figure 1. Relationships between terminology**

for the presence of further symptoms and contextual information), and “causal treatment” (one that is specific for a certain design problem, and relies on the a-priori identification of its instances, using a specific diagnosis procedure).

The relationships that exist between the concepts defined above is illustrated in figure 1.

For specifying design problems we currently use a very simple pattern structure, which has the following three sections:

- *Description*: this is a textual description of the design problem;
- *Indicators*: this section contains a list of indicators (anomalies and contextual hints) which are used to diagnose an instance of the design problem. Indicators are marked with either “M” (*mandatory*) or “O” (*optional*). The difference is that all mandatory indicators are required to fire in order to confirm a problem instance, while optional indicators provide increased certainty in the presence of a given problem instance;
- *Restructuring strategy*: describes a procedure for eliminating an instance of the corresponding design problem. For the time being, an informal description suits our needs perfectly. A formalization of the restructuring strategy, based on the abstract analysis model used in our tools is feasible.

In order to diagnose design problems, we use a two stage approach. First, we run an automated detection process based on detection strategies ([17]), which are a means to compose various metrics, and proper threshold values for each metric, in a rule that expresses a heuristic, which is used to detect instances of a structural anomaly (see figure 2). Based on the set of detected anomalies, and on the problem indicators specified in the problem patterns, we diagnose problem instance candidates. In the diagnosis process, all indicators (anomalies and contextual hints) marked as mandatory need to fire in order to have a valid problem instance candidate. The more additional (optional) indicators fire, the higher the level of certainty in the presence of a problem instance.

### 3. Examples of design problems

This section is dedicated to presenting three examples of design problems, that we also use in the case study. We believe these three problems are relevant for discussion in this paper, because of the following reasons:

- because they address three key areas of the object oriented paradigm: the definition of concepts (section 3.2), the distribution of intelligence (behavior) between the concepts of the application (section 3.1), and inheritance hierarchies (section 3.3);
- because the problems presented in 3.1 and 3.2 are two problems that have similar symptoms, especially the well known “god class”, which is a problem that is often discussed in papers dealing with detection of code smells and therefore is considered relevant. The ability to distinguish between separate causes, and therefore refactoring solutions, for classes affected by this code smell is one of the arguments pleading for a causal approach such as ours;
- because from our own experience in analyzing object oriented systems, these three problems are often encountered.

The individual problem indicators referred to throughout the following three sections (e.g. tradition breaker, god class, class attributes as temporary variables, etc) are described in section 3.4.

#### 3.1. Abusive centralization of control

##### Description

The abusive centralization of control in a class is a typical design problem for systems that have undergone repeated extensions over a longer period of time. Nevertheless, it can also occur as a result of bad design. Contrary to the spirit of the object oriented paradigm, data and associated behavior are not evenly distributed among the system’s classes, but concentrated in usually massive classes that accumulate behavior that actually belongs somewhere else. As a natural consequence, the class suffering from abusive centralization is also strongly coupled to other satellite classes, which in turn tend to be slim.

##### Indicators

- (M) "feature envy" in methods of the class
- (O) class is a "god class"
- (O) the class has satellite data classes (envied classes)
- (O) intensive coupling between abusive class and satellite classes
- (O) violations of "Demeter’s law"

##### Restructuring strategy

- 1: let A be the set of methods suffering from "feature envy" towards intensively coupled satellite classes
- 2: identify cohesive feature envious kernels in methods of A. Fowler’s heuristic concerning extraction of methods based on commented blocks of code can be

- used
- 3: extract methods corresponding to the identified kernels replacing them with method calls
  - 4: move extracted methods to corresponding satellite class based on coupling intensity. Make methods services of these classes.

	Java			C++		
	AVERAGE	HIGH	VERY-HIGH	AVERAGE	HIGH	VERY-HIGH
WMC	14	31	47	23	72	108
AMW (CYCLO/Operation)	2,0	3,1	4,7	2,5	4,8	7
LOC / Class	70	130	195	90	240	360
NOM (& NAS)	7	10	15	19	35	53

### 3.2. Abusive conceptualization

#### Description

One class should model one abstraction. If a class defines attributes and methods that correspond to two or more unrelated concepts, we have a case of abusive conceptualization. This problem is equally likely to appear during design as in the course of the system's evolution. Characteristic manifestations of this problem include among others, low class cohesion, large class size and high complexity and extensive (dispersed but not necessarily intense) coupling to other classes.

#### Indicators

- (M) class is a "god class"
- (O) no class attributes used as "temporary variables"
- (O) extensive coupling to other classes in the system
- (O) refused parent bequest in subclasses of the affected class
- (O) affected class is a "tradition breaker"

#### Restructuring strategy

- 1: identify A, the set of cohesive kernels of data and associated functionality in the class
- 2: extract classes corresponding to the identified cohesive kernels replacing calls with delegations where necessary
- 3: split subclasses of the original affected class, in accordance with the identified kernels and replace calls with delegation as appropriate
- 4: affiliate newly split subclasses as subclasses of the appropriate classes extracted in step 2

### 3.3. Inheritance for implementation reuse

#### Description

This problem corresponds to the situation in which the inheritance mechanism of the programming language is not used in conformity with the object oriented paradigm, for specializing the behavior of the superclass. Instead, the designer/programmer aims at reusing the implementation of a class by inheriting from it. This problem usually appears as a result of quick-and-dirty hacking rather than careful design. As a result of this problem, symptoms such as "refused parent bequest" or "tradition breaker" are not uncommon.

#### Indicators

- (M) at least two of the following indicators must fire
- (O) "refused parent bequest" in the subclass
- (O) subclass is a "tradition breaker"
- (O) superclass is a "short-circuited abstraction"
- (O) subclass uses multiple inheritance

#### Restructuring strategy

- 1: replace inheritance relation with composition

Table 1. Statistical thresholds for some of the metrics used

Semantic Label	Value
SHALLOW	1
FEW	2
MANY	4
SMemCap (Short-term Memory Capacity)	7

Table 2. Additional threshold values used

### 3.4. Definition of indicators and metrics used

The limited space at our disposal in this paper does not allow us to rigorously define all structural anomalies down to the last code metric used. Therefore we will limit ourselves to a semi-formal description of the meaning of individual indicators and their constituent metrics. Since the technique that we rely on is that of *detection strategies*, we refer the reader to [17] for a detailed and rigorous description.

Detection strategies are a means to compose various metrics (and proper threshold values for each metric) in a rule that expresses a heuristic, which is used to detect instances of a structural anomaly (see figure 2). As with any metrics based approach, selecting good threshold values is difficult, because they depend on various factors, including programming language used and type of application. For these reasons, in the rules that specify detection strategies we use symbolic constants in the place of thresholds (e.g. FEW, HIGH, LOW, etc). The actual values that we used in our case study, are defined based on statistical data, collected from more than 60 Java and 50 C++ projects. These are summarized in table 1. The semantics of other metrics (e.g. NOPA) allows us to reason easily in terms of "too much" and "too little". For these metrics the proper thresholds are chosen from the discrete set of semantically "rich" values show in table 2. For metrics with normalized values (between 0 and 1) we will again use a discrete set of thresholds, with clear semantics: 0.25 (one-quarter); 0.33 (one-third); 0.5 (half); 0.66 (two-thirds). The meaning of the individual metrics used in the rules are explained in table 3.

**Extensive coupling:** is a structural anomaly involving a method that is excessively tied to many other methods in the system, and these methods are also dispersed in many of

Metric	Description
AMW	The average statical complexity of all methods in a class. McCabe's cyclomatic number is used to quantify the method's complexity [17]
ATFD	The number of attributes from unrelated classes accessed directly or by invoking accessor methods [17]
CINT	The number of distinct operations called from the measured operation
CDISP	The number of classes in which the operations called from the measured operation are defined in, divided by CINT
FDP	The number of distinct classes in which the attributes accessed in cf. with the ATFD metric are defined
LAA	The number of attributes from the method's definition class, divided by the total number of variables accessed (including attributes used via accessor methods, see ATFD metric)
NAS	The number of public methods of a class, that are not overridden or specialized from the ancestors
NOAM	Number of accessor methods
NOM	Number of methods of the measured class
NOPA	Number of public attributes of the measured class
PNAS	The number of public methods of a class, that are not overridden or specialized from the ancestors, divided by the total number of public methods
TCC	Tight class cohesion: the relative number of method pairs of a class that access in common at least one attribute of the measured class
WMC	Weighted method count: the sum of the statical complexity of all methods in a class. The CYCLO metric is used to quantify the method's complexity
WOC	The number of "functional" public methods, divided by the total number of public members [17]

**Table 3. Short description of used metrics**

the system's classes. A method is considered to suffer from extensive coupling when it is not a flat function (in order to filter out "configuration" methods), and when (see figure 2)

$$EC = (CINT > SMemCap) \text{ and } (CDISP \geq 0.5)$$

**Intensive coupling:** is a structural anomaly involving a method of a class that is excessively tied to other methods in the system, but which belong to a relatively small number of classes (low dispersion of methods). A method is considered to suffer from intensive coupling when it is not a flat function (in order to filter out "configuration" methods), and when

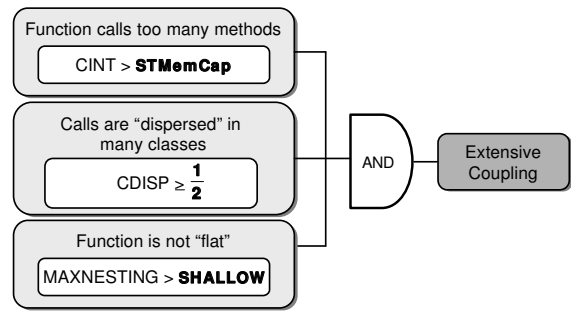
$$IC = (CINT > SMemCap) \text{ and } (CDISP < 0.5) \text{ or } (CINT > FEW) \text{ and } (CDISP < 0.25)$$

**Feature envy:** is a structural anomaly that applies to methods that seem more interested in the data of other classes than that of its own class [9]. The detection is based on counting the number of foreign data members that are accessed (either directly or via accessors):

$$FE = (ATFD > FEW) \text{ and } (LAA < 0.33) \text{ and } (FDP \leq FEW)$$

A "true" feature envy is the case when the envied data comes from a very few classes (usually one).

**Data class:** is a class that acts as a "dumb" data holder for other classes, without complex functionality [9, 19]. In order to detect data classes, we search for "lightweight" classes, classes that define many accessor methods, and classes that declare public data:



**Figure 2. Detection strategy for extensive coupling**

$$DC = (WOC < 0.33) \text{ and } ((NOPA + NOAM > MANY) \text{ and } (WMC < VERY\_HIGH) \text{ or } (NOPA + NOAM > FEW) \text{ and } (WMC < HIGH))$$

**God class:** is a structural anomaly that applies to large, complex "monster"-classes, that tend to do most of the work while delegating very little [9]. As shown in this paper, god classes can be symptomatic for several individual problems (i.e. it has several possible causes). The heuristics we use to detect god classes look for large, non-cohesive classes (non-communicating behavior), which access a lot of foreign data:

$$GC = (ATFD > FEW) \text{ and } (WMC \geq VERY\_HIGH) \text{ and } (TCC < 0.33)$$

**Tradition breaker:** is a heuristic that corresponds to the idea that in an inheritance hierarchy, the interface of a class should increase in an evolutionary fashion, compared to that of its parent. This means that a derived class should not break the inherited "tradition" and provide a large set of new services that are unrelated to those provided by the base class. In order to detect this structural anomaly we look for an excessive increase in the child's interface size, coupled with substantial complexity in size in the child class and a minimal size and complexity in the parent. The detection rule for tradition breaker is represented in figure 3, and has three components. The first component checks for an excessive increase in the child's interface size and is defined as

$$(NAS \geq AVERAGE(NOM)) \text{ and } (PNAS \geq 0.66)$$

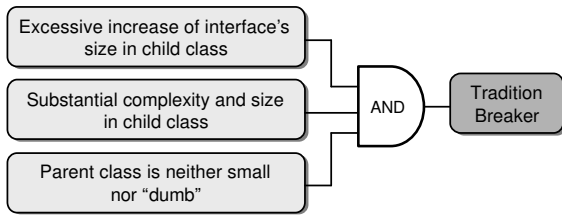
The second component checks for substantial complexity and size in the child class, and is defined as

$$((AMW > AVERAGE) \text{ or } (WMC \geq VERY\_HIGH)) \text{ and } (NOM \geq HIGH)$$

Finally, the last component of the rule filters out small or "dumb" parent classes:

$$(AMW > AVERAGE) \text{ and } (NOM \geq 0.5 * HIGH) \text{ and } (WMC \geq 0.5 * VERY\_HIGH)$$

**Refused parent bequest:** is a structural anomaly that applies to the inheritance relation between a superclass and its subclasses (see [9, 19]). It has two forms: the one that concerns inherited functionality and the more critical case of refused interface, when a subclass privately inherits



**Figure 3. Detection strategy for tradition breaker**

interface methods, or overrides them with empty methods.

**Attributes as "temporary variables"**: This indicator corresponds to the situation in which private attributes of a class are being used as a replacement for temporary variables. In order to detect this situation, we look for private attributes that are accessed from within less than two methods of the class (no methods means that the attribute is a dead attribute).

**Violation of "Demeter's law"**: refers to the well known principle having the same name. For detecting instances of this anomaly, we use the heuristic described under "message chains" in [9]. We currently do not have a detection strategy that can accurately make an automatic detection of the anomaly, thus a manual inspection is always necessary.

**Short-circuited abstraction**: is an indicator that refers to the situation in which a superclass is systematically overlooked throughout the system. In other words, the clients of its subclasses address these subclasses directly, rather than through the abstract interface defined in the superclass. This anomaly can potentially hint towards a problematic inheritance relation.

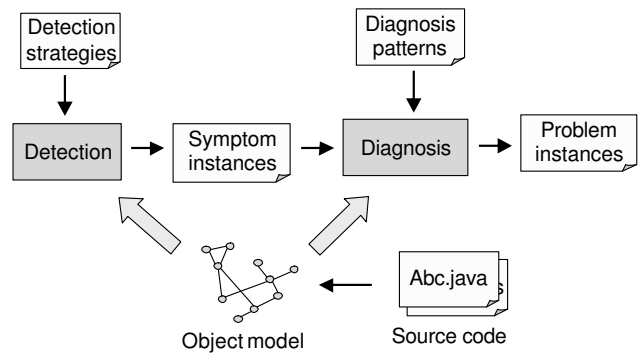
**Class uses multiple inheritance**: This is an example of a contextual hint (the second type of problem indicator). It merely reports classes in the system, that inherit from several other classes.

Table 3 gives a short description of the individual metrics used in defining the problem indicators above.

## 4. Experimental setup

### 4.1. Tool support

The general process that we follow in order to diagnose design problems is illustrated in figure 4. First, the source code of the subject system is parsed and an object model



**Figure 4. Diagnosis infrastructure**

Property	Size	Remarks
Lines of Code	223.068	including comments
Source Files	1.209	
Packages	99	
Classes	1.393	including 140 inner class
Methods	9.561	
Attributes	3.358	

**Table 4. Size properties of the ArgoUML case study**

of the system is built. The object model is persistable in a database (not shown in the figure).

On top of the object model, we run an automated detection process based on a tool called iPlasma [16]. The tool is an integration platform for several low level instruments of investigation developed over time. It allows complex analyses, combining both detection strategies and code duplication to be quickly and easily defined, in a visual manner. Once individual instances of code smells are detected, we aggregate them in order to find problem candidates.

### 4.2. Case study

As our case study, we chose to take ArgoUML<sup>1</sup>, a well known open-source UML modelling tool. The system is implemented in Java and consists of roughly 220.000 lines of code (including comments). For more detailed size and complexity figures, see table 4.

### 4.3. Experimental approach

The fundamental hypothesis for our experiment is that correlations of structural anomalies can hint towards the concrete cause of bad design, and towards the concrete, necessary treatment. The more problem indicators fire during the analysis, and the stronger these indicators are (in

<sup>1</sup><http://argouml.tigris.org>

the sense of abnormal metric values), the more accurate our predictions towards the causing problem are, and the more probable it is, that our suggested treatment is meaningful. Having that in mind, we analyzed the system described above, with the following concrete goals in mind:

- to test whether our fundamental hypothesis is true, that is if the predicted problem/treatment is meaningful for at least the top suspects (i.e. those that show most symptoms);
- starting from the well known structural anomaly called “god class”, and taking into consideration the important role that it plays in two of our described problems (3.1 and 3.2), to see if the process of diagnosis is able to distinguish correctly between cases of one or the other problem type;
- to investigate the possibility of having more than one type of problem affect the same design fragment (e.g. a class) and how to deal with such situations.

We realize that in order to fully validate the approach, further case studies as well as a rigorous analysis of false positives and false negatives are necessary.

## 5. Discussion on experiment results

In the following three sections, we are going to address the three design problems defined in section 3. For each problem, we will insist on a selected few of the top candidate classes, and present some detailed information about their implementations. The criteria for selecting the examples discussed are:

1. the number of coexisting anomalies;
2. the intensity of each anomaly. Because this is not obvious from the tables presented in the paper, for each selected example we will provide the relevant information on the intensity of the occurring anomalies.

The last subsection is dedicated to the most important lessons learned.

### 5.1. Abusive centralization of control

The abusive centralization of control in a class is a typical design problem for systems that have undergone repeated extensions over a longer period of time, but it can also occur as a result of bad design. Contrary to the spirit of the object oriented paradigm, data and associated behavior are not evenly distributed among the system’s classes, but rather concentrated in usually massive classes that accumulate behavior that actually belongs somewhere else. As a

Class	FE	GC	DC	IC	DL
Project	*	*	*		
FigObject	*	*	*		
CoreFactory	*	*	*		
ZargoFilePersister	*		*		
XMIParser	*		*		
DesignIssuesDialog	*		*		
FigEdgeNote	*		*		
ClassDiagramLayouter	*			*	
ProjectBrowser	*	*			*
PGMLParser	*	*			

**Table 5. Top ten candidates for abusive centralization of control**

natural consequence, the class suffering from abusive centralization is also strongly coupled to other satellite classes, which in turn tend to be slim (see section 3.1).

Letting our tools search for instances of this design problem, the top ten candidates that we obtained are shown in table 5, together with the indicators that fired during the analysis. The meaning of the acronyms in the head of each column of the table are: *feature envy* (FE), *god class* (GC), *data class* (DC), *intensive coupling* (IC) and *violation of Demeter’s law* (DL).

Based on the aforementioned criteria, we selected the following three cases for a closer inspection: Project, ProjectBrowser, and CoreFactory.

The class Project is the absolute number one candidate to examine, and a superb example of abusive centralization of control being at the same time a “god class” and showing feature envy towards three other data classes around it (ProjectMemberModel, ProjectMemberDiagram and GenerationPreferences). Figure 5 is a Code Crawler [15] polymetric view which gives a very suggestive picture on the degree of coupling between Project and many other classes of the system. The class defines a number of 85 methods and is coupled to a staggering 131 classes, although this number includes both the incoming and the outgoing. In the figure, those classes represented above Project depend on it, while those represented underneath are depended upon. Since the problem in discussion is mainly characterized by intensive, and not extensive outgoing coupling to neighbors, the relatively low number of dependees is not surprising. Nevertheless, the fact that it depends on relatively many classes, and it serves a very large number of clients, is indicative of the exaggerated amount of “expertise”, accumulated in its 85 methods. In addition the class has cyclic invocation dependencies with ProjectBrowser and CoreFactory.

The third entry in our top ten, CoreFactory, also deserves our attention, being the second largest class in the system. A closer look at it revealed that it defines no less than 116 methods, out of which at least two qualify as so called “brain methods” (i.e. very large and complex methods).

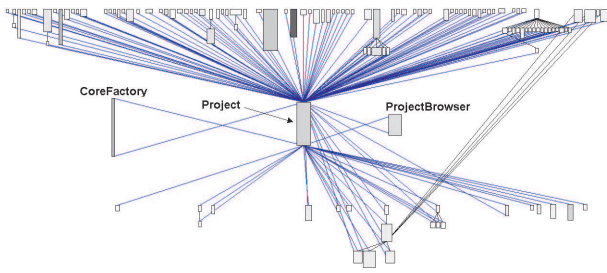


Figure 5. The class Project and its ties

Furthermore, the class accesses 34 attributes in 14 different and unrelated classes (either directly or through accessor methods), it is coupled to a data class (CommentEdge) and interestingly, also possesses a lot of code duplication.

Our next interesting candidate, is the class ProjectBrowser, which is the only class in the top ten that shows violations of “Demeter’s law”. A relevant example is the following code sequence found in the method setTitle(): Furthermore, the class directly accesses data from seven

```
String changeIndicator =
    ProjectManager.getManager().getCurrentProject().
        getSaveRegistry().hasChanged() ? " *" : "";
ArgoDiagram activeDiagram =
    ProjectManager.getManager().getCurrentProject().
        getActiveDiagram();
```

other classes.

## 5.2. Abusive conceptualization

In conformity with the object oriented paradigm, one class should model one abstraction. If a class defines attributes and methods that correspond to two or more unrelated concepts, we have a case of abusive conceptualization. This problem is equally likely to appear during design as in the course of the system’s evolution. Characteristic manifestations of this problem include among others, low class cohesion, large class size and high complexity and extensive (dispersed but not necessarily intense) coupling to other classes (see section 3.2).

As for the previous problem, we have compiled a top ten for this problem type as well. It is presented in table 6. The meaning of the acronyms in the head of each column of the table are: *god class* (GC), *attributes used as temporary variables* (TV), *extensive coupling* (EC), *refused parent bequest* (RPB) and *tradition breaker* (TB).

In the following, we will take a closer look at some of the top classes. Interestingly, some of them (e.g. Project, FigObject) also appeared to suffer from excessive centralization, as described above. This is not a surprise for us, but it poses some interesting questions concerning the order

Class	GC	no TV	EC	RPB	TB
Modeller	*	*	*		
FigObject	*	*			*
FigNodeModelElement	*	*		*	
FigEdgeModelElement	*			*	
Project	*	*			
ModeCreateEdgeAndNode	*	*			
ProjectBrowser	*	*			
FigComment	*	*			
Designer	*	*			
ToDoPane	*	*			

Table 6. Top ten candidates for abusive conceptualization

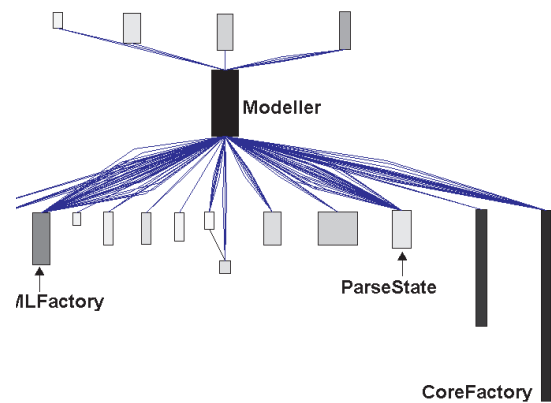


Figure 6. The class Project and its collaborations

in which the two problems should be removed from these classes. We will address this point again in section 5.4.

The class Modeller is a very good example of a class with abusive conceptualization. As can be seen in figure 6, the class consequently shows extensive coupling to a lot of other classes in the system. Modeller is also characterized by a lot of brain methods: 8 methods are longer than 50 LOC, and the largest of them, addDocumentationTag() is more than 150 LOC. Another aspect that makes this class interesting is that it does not seem to be affected by excessive centralization, although it is a god class. This shows the ability of our diagnosis approach to precisely distinguish between situations that, by using a symptomatic approach, would have been considered identical. The power lies exactly in the ability to point out the cause behind the bad shape of a structure fragment, and not only the state itself.

Our final subject is the class FigObject, which is interesting because it is the only tradition breaker in the top ten (being a subclass of FigNodeModelElement). After a closer look we established that part of it does indeed not belong in the class (not even in the hierarchy), and should be extracted

into a separate class.

### 5.3. Inheritance for implementation reuse

As described in section 3, this problem corresponds to the situation in which the inheritance mechanism of the programming language is not used in conformity with the object oriented paradigm, for specializing the behavior of the superclass. Instead, the designer/programmer aims at reusing the implementation of a class by inheriting from it. This problem usually appears as a result of quick-and-dirty hacking rather than careful design. As a result of this problem, symptoms such as "refused parent bequest" or "tradition breaker" are not uncommon.

Searching for instances of this problem throughout the system did lead us to just a few candidates, out of which one deserves some attention. It is the class FigLink, which shows signs of both refused parent bequest and tradition breaker. We confirmed that the diagnosis of inheritance for the purpose of reusing implementation was indeed justified.

### 5.4. Lessons learned

Based on problem-specific constellations of various structural anomalies (most of which are well known from literature) we were able to diagnose concrete problems which express a precise cause and admit precisely defined refactoring solutions. This way, we have achieved causality in the restructuring process for the first time.

Our experiment also showed that it is possible to distinguish between problems having related symptoms, such as abusive centralization and abusive conceptualization, with their dominating symptom "god class". Concretely, in the case of the two classes Modeller and CoreFactory, the different correlations between additional code smells allowed us to accurately distinguish between two separate design problems affecting each class, which would not have been possible using a symptomatic approach (i.e. searching for one type of symptom only).

At the same time, we have identified important areas for future research, for example the question of how to deal with design fragments that are affected by more than one design problem at the same time (e.g. the classes FigObject and Project). Here, a system of priorities between pairs of different problem types could be imagined.

In conclusion, we have been able to show that our diagnosis approach has real potential in improving current restructuring practices, by providing an accurate diagnostic with respect to the cause of bad structure and what needs to be done about it.

## 6. Related work

As foundation for the detection of structural anomalies we use *design principles*, *design rules and heuristics* [18, 5, 19, 9], as well as the concept of *detection strategy* [17], which provides a refined mechanism to express combinations of metrics that formalize various heuristics, intended to highlight structural anomalies in code.

Concerning code refactorings as the low level mechanism to perform safe source code transformations we refer the reader to [20, 9].

Other attempts that target in the direction of linking problem detection with refactorings are addressed in the following. In [1] and more recently in [6], a number of best practices in reengineering large object-oriented systems is identified and formulated in the form of reengineering patterns. Our approach distinguishes itself mainly through its causal nature, which allows us to formulate precise restructuring strategies for the problems we diagnose, rather than vague suggestions. The same applies to a similar effort, although with a somewhat wider scope than just restructuring, the so called anti-patterns described in [2].

In [11, 13, 14], opportunities of inserting design patterns to places in the source code where such patterns are missing, or present in a distorted form, are searched for. The approach focuses exclusively on design patterns. Although widely accepted as good practice, design patterns are not mandatory, therefore their absence cannot be necessarily considered as problematic.

A more recent optimization approach for structure improvement, with good first results, can be found in [21]. The technique is based on applying genetic programming on a model in order to "evolve" the structure of a system. By operating on a simplified model, the process runs without human intervention. The result is a recommended, optimal structure of the model (with respect to the cost function defined), which (for the time being) needs to be implemented manually in the real system.

Significant work concerning tool support for automated introduction of design patterns has been done in [3]. Concerning low-level tool support for generic code transformations, we refer the reader to [23].

## 7. Conclusion

In this paper, we presented a causal approach to support the decision making process in object oriented restructuring. Our approach builds on the explicit encapsulation of correlations between structural anomalies, and other structural and semantic information, into self-sufficient design problems, as well as an explicit pattern-like mapping of problems to univocal, adequate treatments. The main contribution of our method lies in the causal nature of the pre-

scribed treatments and the explicit, reusable documentation of the mappings between symptoms, the causing problem, and measures necessary to defuse a problem instance. The approach has a high potential for automation, which can lead to an important decrease in costs associated with restructuring.

## References

- [1] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A.-M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS object-oriented reengineering handbook, 1999.
- [2] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.
- [3] M. Ó. Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 463–472. IEEE, 1999.
- [4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 18–32, 1999.
- [5] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall, London, second edition, 1991.
- [6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.
- [7] A. Eastwood. Firm fires shots at legacy systems. *Computing Canada*, 19(2):17, 1993.
- [8] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, May/June:17–23, 2000.
- [9] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison-Wesley, 1996.
- [11] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the TOOLS 39*, pages 296–305, 2001.
- [12] S. Huff. Information systems maintenance. *The Business Quarterly*, 55:30–32, 1990.
- [13] J. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. In *ESEC/FSE '97 Workshop on Object-Oriented Reengineering*, 1997.
- [14] S.-U. Jeon, J.-S. Lee, and D.-H. Bae. An automated refactoring approach to design pattern-based program transformations in java programs. In *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*. IEEE, 2002.
- [15] M. Lanza. *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained and Evolutionary Software Visualization*. PhD thesis, University of Berne, 2003.
- [16] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005), Tools Section*, 2005.
- [17] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timișoara, 2002.
- [18] R. C. Martin. Design principles and design patterns. Object Mentor, 2000.
- [19] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, first edition, 1996.
- [20] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [21] O. Seng and G. Pache. Search based structure improvement. In *Proceeding of the First International Workshop on Software Evolution Transformations (SET 2004)*, 2004.
- [22] T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.
- [23] The Inject/J team. The Inject/J website. [injectj.sourceforge.net](http://injectj.sourceforge.net).