4

Evaluating the Design

Object-oriented construction and design are *misleading* words, because they make people think that software can be constructed like a house or designed like a piece of furniture. This is a myth which is hard to kill. The truth is that a software system is at least as complex as any other engineering artifact (such as buildings, if not more, considering the fact that it evolves much faster).

Moreover, a modern software system is written by many people at the same time, leading to (1) communication issues, (2) compatibility issues and above all (3) complexity issues. In addition, a system cannot be written once and for all, put in place and then work forever. It is actually grown like a plant with many interrelated parts that depend on each other, that die, that change, that are bugged and must be fixed (introducing new bugs), etc.

You may well imagine that a plant which is not correctly watered will die. In much the same spirit we can say that a system which is not being cured and maintained will slowly decay and eventually die. But all metaphors, including the one of the plant, do not fit the context of object-oriented software. These systems are much more complex and consist of thousands of artifacts and relationships between the artifacts. A change in one part of the system may break other parts of the system. This is not due to bad programming practice, it is just a matter of complexity: you cannot expect to have a complete picture of a large software system. Moreover, we are speaking about evolving systems which change continuously, leading to more complexity [LB85, DDN02].

Still in software construction finding an appropriate design is important. Indeed it may help people understand the system and ease future changes. For example, it is well-known that using explicit type-

checks goes against the essence of object-oriented programming and creates brittle and hard to change code [DDN02]. However it is important to understand that design decisions such as the impact of applying a given design pattern [GHJV95, ABW98] is difficult to assess — using a design pattern introduces an intrinsic complexity which should be balanced by the benefits of the pattern application. Identifying the exact responsibilities of objects and how they should be distributed among objects is complex. In this book we show you how to use metrics to assess the quality of a design. Metrics measure structural elements and as such they can reveal hidden symptoms. But there will always be a gap between the symptoms and the deep assessment that an expert in object-oriented design can do using these symptoms. Therefore it is important to consider metrics as a tool and as with any tool to know their advantages and disadvantages. This leads us to the crucial questions we answer in this book:

What entities do we measure in object-oriented design?

It depends . . . on the language. In most object-oriented languages we find and can measure *classes*; *operations* (including methods and functions); *variables* (including the whole range from attributes to local variables) etc.

What metrics do we use?

It depends . . . on our measurement goals. We may want to assess the size, the complexity, the quality, etc.

What can we do with the information obtained?

It depends . . . on our objectives. We may want to just assess the status quo to calm down management, we may want to brag with colleagues (“my system is bigger and better than yours”), or we may actually want to ameliorate the quality of parts of the system.

Design Harmony

Simple metrics are not enough to understand and evaluate design, or to put it bluntly: you cannot understand the beauty of a painting by measuring its frame or understand the depth of a poem by counting the lines. Object-oriented systems can be seen as pieces of complex art and the creativity that programming involves backs up this bold statement. Metrics can help to evaluate and improve designs, but

those have to be *meaningful* metrics that are put in a context of *design harmony*.

The reader might be confused to find a word as ambiguous as *harmony* in a book about object-oriented metrics. After all, a major point of this book is that software, objet-oriented and not, can and should be measured. However, metrics have to be put in a context. The aspect of measurability and more specifically about thresholds (such as: When should a class or a method be considered *too large*?) does not make sense if there is no context: A class implementing a parser is never going to be small, the domain is just too complex to be modelled in a concise way. Still, and this is where harmony comes into play, a class can be implemented in several ways, theoretically even in only one huge method. This would however make the class hard to understand.

An application, a class, a method and any other artifact in a software system should be implemented in an harmonious way, e.g., a class has to implement an *appropriate* number of methods of *appropriate* size, complexity, and functionality.

Appropriate to what? This appropriateness is a kind of *harmony* that can indeed be measured and reached. This overall harmony is composed of three distinct harmonies that concern every software artifact:

1. **Identity Harmony** – *"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?

2. **Collaboration Harmony** – *"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?

3. **Classification Harmony** – *"How do I define myself with respect to my ancestors and descendants?"*. This harmony combines elements of both identity and collaboration harmony in the context of inheritance. For example, does a subclass use all the inherited services, or does it ignore some of them?

Boiling it down: Every artifact in a system needs to be in harmony with itself (not too large, not too small, not too complex, not too simple, etc.), in harmony with its collaborators (do not talk to everybody,

do not talk to nobody, etc.), and finally in harmony with its ancestors and descendants. Every artifact must have its *appropriate* place, size, and complexity to fit the system context.

Detection Strategies and Class Blueprints

In the remainder of this chapter we present two techniques to evaluate the design of object-oriented systems and to detect structural disharmonies:

1. A *detection strategy* is a composed logical condition, based on metrics, that identifies those design fragments that are fulfilling the condition.
2. A *class blueprint* is a semantically rich visualization of the internal structure of classes and class hierarchies. We use a *class blueprint* to inspect source code and to detect visual anomalies which in turn point to design disharmonies.

4.1 Detection Strategies

The Principles of Detection Strategies

A metric alone cannot help to answer all the questions about a system and therefore metrics must be used in combination to provide relevant information. Why?

Using a medical metaphor we might say that the interpretation of abnormal measurements can offer an understanding of *symptoms*, but the measurements cannot provide an understanding of the *disease* that caused those symptoms. The *bottom-up approach*, i.e., going from abnormal numbers to the recognition of design diseases is impracticable because the symptoms captured by single metrics, even if perfectly interpreted, may occur in several diseases: The interpretation of individual metrics is too fine grained to indicate the disease.

This leaves us with a major gap between the things that we measure and the things that are in fact important at the design level with respect to a particular investigation goal.

How should we combine then metrics in order to make them serve our purposes? The main goal of the mechanism presented below is to provide engineers with a means to work with metrics at a *more abstract level*. The mechanism defined for this purpose is called a *detection strategy*, defined as follows:

> *A* Detection Strategy *is a composed logical condition, based on metrics, by which design fragments with specific properties are detected in the source code.*

The aim with *detection strategies* is to make design rules (and their violations) quantifiable, and thus to be able to detect *design problems* in an object-oriented software system, i.e., to find those design fragments that are affected by a particular design problem.

The use of metrics in the *detection strategies* is based on the mechanisms of *filtering* and *composition*, described next.

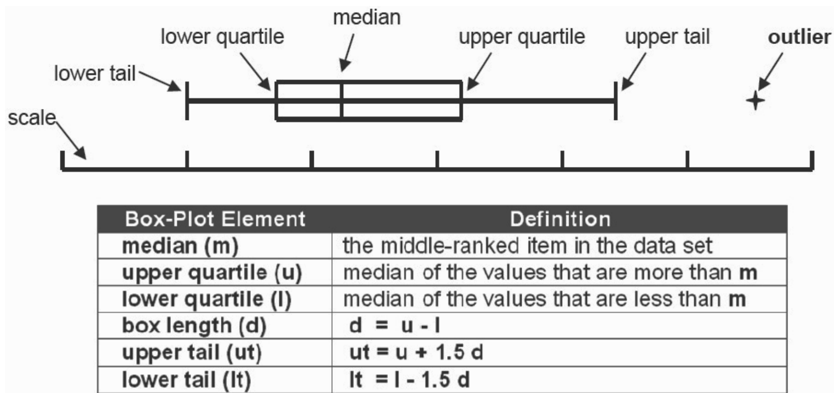
Filtering

The key issue in filtering is to reduce the initial data set so that only those values that present a special characteristic are retained. A *data filter* is a boolean condition by which a *subset* of data is retained from an initial set of measurement results, based on the particular focus of the measurement.

The purpose of filtering is to keep only those design fragments that have special properties captured by the metric. To define a data filter we must define the values for the bottom and upper limits of the filtered subset. Depending on how we specify the limit(s) of the resulting data set, filters can be either *statistical*, based on *absolute thresholds*, or based on *relative thresholds*.

Statistical Filters

A first approach when we seek abnormal values in a data set is to employ statistical means for detecting those values. Thus, the (binary) filtering condition and its semantics are implicitly contained in the statistical rules that we use. The advantage of this approach is that it is not necessary to specify explicitly a threshold value beyond which entities are considered abnormal. One significant example of a statistical filter is the *box-plot technique*, which is a statistical means for detecting the abnormal values (outliers) in a data set [FP96]. In this case, the detection of outliers starts from the *median* value, which can be directly computed from the analyzed data set. Based on this median value, two pairs of thresholds are computed i.e., the *lower/upper quartile* and resp. *lower/upper tail*. These thresholds are again computed *implicitly*, based on the formulas presented in Fig. 4.1. Eventually, in a box-plot an *outlier* is a value from the data set that is either higher than the *upper tail* or lower than the *lower tail* thresholds.



Box-Plot Element	Definition
median (m)	the middle-ranked item in the data set
upper quartile (u)	median of the values that are more than **m**
lower quartile (l)	median of the values that are less than **m**
box length (d)	**d = u - l**
upper tail (ut)	**ut = u + 1.5 d**
lower tail (lt)	**lt = l - 1.5 d**

Fig. 4.1. The box-plot technique [FP96].

Threshold-Based Filters

The alternative way of defining filters is to pick-up a *comparator* (e.g.,*lower than* or *highest values*) and specify explicitly a threshold value (e.g., *lower than 10* or *5 highest values*). But, as already discussed in Chapter 2 (see Sect. 2.1), the selection of proper thresholds is one of the hardest issues in using metrics. There are two ways in which these filters can be specified:

1. *Absolute Comparators.* We use the classical comparators for numbers, i.e., $>$ (*greater than*); $\geq$ (*greater than or equal to*); $<$ (*less than*); $\leq$ (*less than or equal to*).
2. *Relative Comparators*. The operators that can be used are *highest values* and *lowest values*. These filters delimit the filtered data set by a parameter that specifies the *number* of entities to be retrieved, rather than specifying the maximum (or minimum) value allowed in the result set. Thus, the values in the result set will be *relative* to the original set of data. The used parameters may be *absolute* (e.g., retrieve the 20 entities with the highest LOC values) or *percentile* (e.g., retrieve the 10% of all entities having the lowest LOC values). This kind of filter is useful in contexts where we consider the highest or lowest values from a given data set, rather than indicating precise thresholds.

Composition

In contrast to simple metrics and their interpretation models, a *detection strategy* is intended to quantify more complex design rules,

that involve multiple aspects that needed quantification. As a consequence, in addition to the filtering mechanism that supports the interpretation of individual metric results, we need a second mechanism to support a correlated interpretation of *multiple result sets* – this is the *composition* mechanism. It is based on a set of AND and OR operators that compose different metrics together to form a composite rule.

Graphical Notation for Detection Strategies

A *detection strategy* is a composed logical expression by which design entities addressed by the strategy are filtered. Instead of using formulas, we decided to take advantage of a well-known graphical notation used to represent logical circuits. In this representation, the *composition operators* are represented as logical AND and OR gates (see Fig. 4.2). Both the input and the output terms of the gates are *filters*. Inputs can be either *simple* or *composed* filtering conditions.

Representation of Simple Filters

A *simple filter* is represented as a gray rounded rectangle, composed of an *informal description* of the filtering condition and a white compartment (box) where the *filtering formula* is depicted i.e., the metric followed by the filtering operator and the threshold value (see Fig. 4.2).

Representation of Composed Filters

A *composed filter* is represented as a gray rounded rectangle that contains only the *informal description* of the composed condition that it stands for (see Fig. 4.3). Note, that a composed filter is always the result (output) of another gate. Notice, that these intermediary terms are not conceptually necessary. We introduced them, in order to increase increase the understandability of more complex *detection strategies*.

Detection Strategies Exemplified

A *detection strategy* can be used to express in a quantitative manner deviations from a given set of *rules of design harmony*. While it is impossible to establish an objective and general set of such harmony rules that would lead automatically to high-quality design if

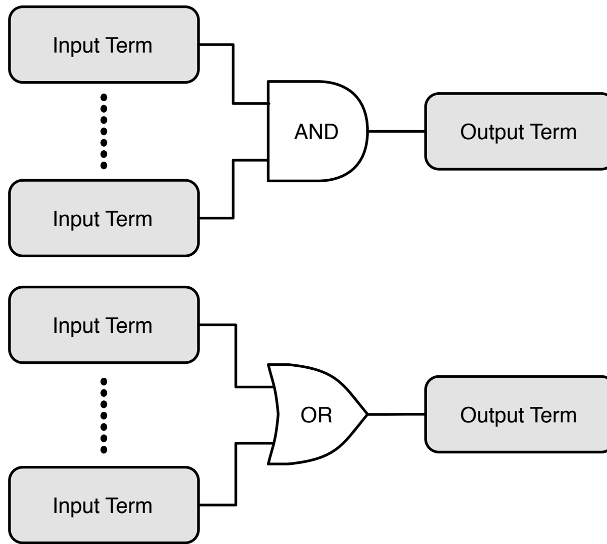


Fig. 4.2. Composition operators used in *detection strategies* represented as logical AND/OR gates.

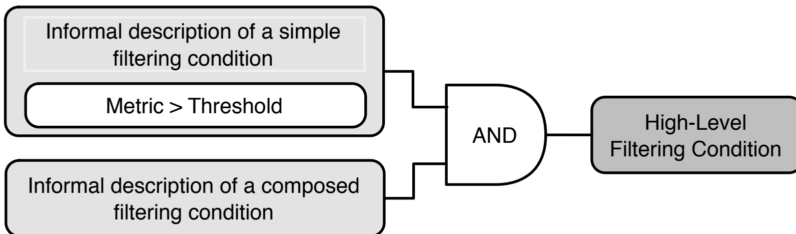


Fig. 4.3. A graphical representation of a *detection strategy*.

they would be applied, yet heuristic knowledge reflects and preserves the experience and quality goals of the developers.

As a consequence, over the last two decades, many authors were concerned with identifying and formulating design principles [Mey88b] [Lis87] [Mar02b], rules [CY91] [Mey88b], and heuristics[Rie96] [JF88] [Lak96] [LR89] that would help developers fulfill those criteria while designing their systems.

An alternative approach to disseminating heuristical knowledge about the quality of the design is to identify and describe the symptoms of bad-design.

This approach is used by Fowler in his book on refactorings [FBB+99] and by the “anti-patterns” community [BMMM98] as they try to identify situations when the design must be structurally improved. Fowler describes around twenty code smells – or “bad smells” as the author calls them – that address symptoms of bad design, often encountered in real software systems.

Let us see now, based on the concrete example of the *God Class* [Rie96] design flaw, how *detection strategies* can be defined for a concrete design flaw. The entire process is summarized in Fig. 4.4.

The starting point in defining such a *detection strategy* is given by one (or more) *informal design rules* — like those stated by Riel [Rie96], Martin [Mar02b] or Fowler [FBB+99] — that comprehensively define the design problem, i.e., the disharmony that we want to capture. In this concrete case we start from the three heuristics related to the *God Class* problem, as described by Riel [Rie96]:

> *Top-level classes in a design should share work uniformly.* [...]
> *Beware of classes with much non-communicative behavior.* [...]
> *Beware of classes that access directly data from other classes.*

Step 1: Identify Symptoms

The first step in constructing a *detection strategy* is to break down the informal rules in a correlated set of *symptoms* (e.g., class inflation, excessive method complexity, high coupling) that can be captured by a single metric. In our case the first rule refers to *high class complexity.* The second rule speaks about the level of intra-class communication between the methods of the class; thus it refers to the *low cohesion of classes.* The third heuristic addresses a special type of coupling, i.e., the direct access to instance variables defined in other classes. In this case the symptom is *access of foreign data.*

Step 2: Select Metrics

The second step is to *select proper metrics* that quantify best each of the identified properties. In this context the crucial question is: from where should we take the proper metrics? There are two alternatives:

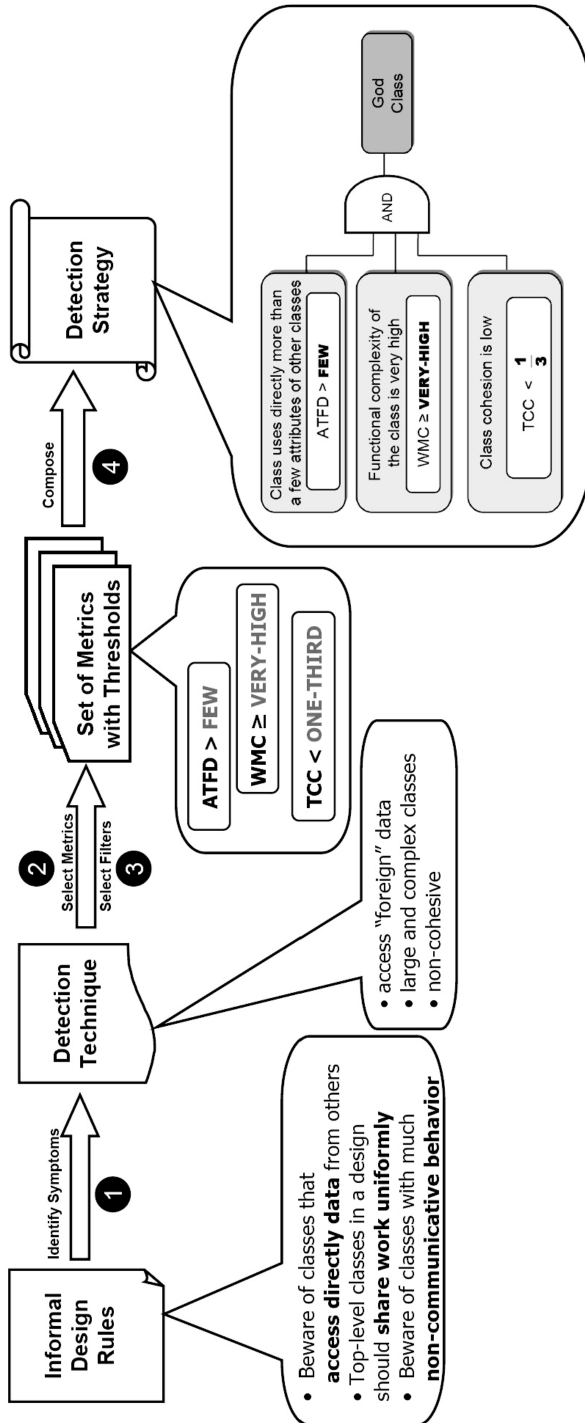


Fig. 4.4. Process of transforming an informal design rule in a *detection strategy*.

1. *Use well-known metrics from the literature.* For example, we could choose a metric from a well-known metrics suite (e.g., the Chidamber&Kemerer [CK94] suite), or from the metrics summarized by various authors (e.g., Lorenz and Kidd [LK94], Henderson-Sellers [HS96], Briand [BDW99, BDW98] etc.)
2. *Define a new metric (or adapt an existing one)*, so that the metric captures exactly one of the symptoms (see previous step) that appears in that design flaw that we intend to quantify.

Our approach is a conservative one, i.e., we try to use as much as possible metrics from the literature, avoiding thus to define new (oftentimes unnecessary) metrics. Yet, in the same time we want to emphasize that, in defining a good *detection strategy*, it is very important not to sacrifice the *exact* quantification of a symptom, just for the sake of using an existing metrics from the literature. In other words, if no adequate metric can be found in the literature, define a new metric that reflects one symptom that needs to be quantified.

For the *God Class* design flaw these properties are class complexity, class cohesion and access of foreign data. Therefore, we choose the following set of metrics[1]:

- *Weighted Method Count* (WMC) is the sum of the statical complexity of all methods in a class [CK94]. We consider McCabe's cyclomatic complexity metric as a complexity measure [McC76, LK94].
- *Tight Class Cohesion* (TCC) is the relative number of methods directly connected via accesses of attributes [BK95, BDW98].
- *Access to Foreign Data* (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods.

Notice that while the first two metrics (i.e., WMC and TCC) are metrics defined in the literature, the last one was defined by us in order to capture a very specific aspect, i.e., the extent to which a class uses attributes of other classes.

Step 3: Select Filters

The next step is to define for each metric the filter that captures best the symptom that the metric is intended to quantify. As mentioned earlier, this implies to (1) pick-up a comparator and (2) to set an

[1] For a precise description of all the metrics used in the book, including the metrics below please refer to Appendix A.

adequate threshold, in conformity with the semantics described in Sect. 2.1.

In our concrete case, the first symptom is referring to *excessively high* class complexity we want to find classes that are complexity outliers. Thus, for the WMC metric we use the $\geq$ (*greater than or equal to*) comparator. How do we find the threshold for extremely high values of the WMC complexity metric? There is no other way than to base it on statistical data related to complexity, as described in Sect. 2.1. Based on the semantic labels described there, we can say now that we will use the *very high* threshold value.

For capturing the aspect of "access to foreign data" we use the $>$ (*greater than*) comparator, whereby the threshold value will be the maximal number of "tolerable" foreign attributes to be used. Thus, the threshold value for ATFD, does not need to be based on statistics, because the metric has a precise semantic: It measures the extent of encapsulation breaking. Based on the rationale presented in Sect. 2.1 "accidental" usage of foreign data, and consequently a *few* such usages are harmless; thus, $ATFD > FEW$.

Eventually, for the *low cohesion* symptom we choose the $<$ (*less than*) comparator. In order to set the proper threshold, we first have to notice that the values of TCC are fractions; thus we can use one of the thresholds with *fraction semantics* summarized in Table 2.3. As this filter must capture non-cohesive classes, we decided to use the *one-third* threshold (see Sect. 2.1), meaning that only one third of the method pairs of the class have in common the usage of the same attribute. If we wanted to capture more extreme cases of non-cohesiveness, we could have used the *one-quarter* threshold.

Step 4: Compose the Detection Strategy

The final step is to correlate these symptoms, using the composition operators described previously. From the context of the informal heuristics as presented by their author in [Rie96], we infer that all these three symptoms should co-exist if a class is to be considered a behavioral *God Class*. Consequently, the final form of the *God Classes detection strategy* is the one depicted in Fig. 4.5.

The Missing Link

Detection strategies are useful to detect problems in object-oriented designs. What they finally produce is a list of *suspects*, i.e., all entities in the system which conform to the applied *detection strategy*.

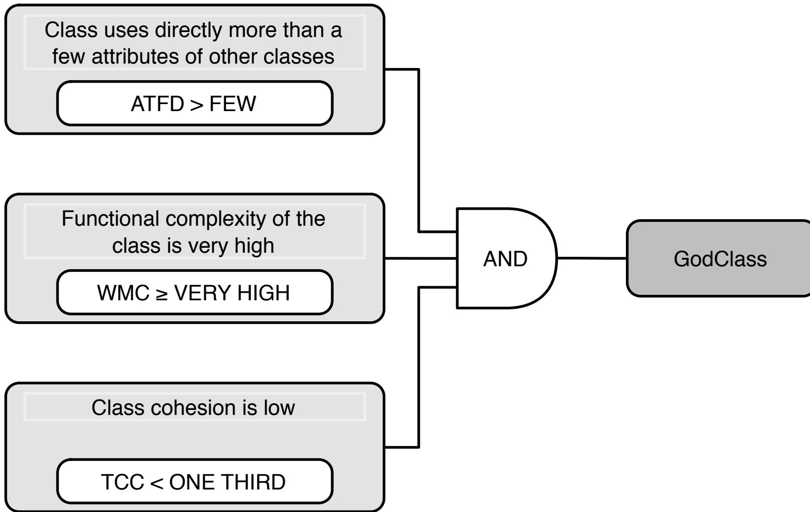


Fig. 4.5. Detection of a *God Class*

These suspects must be manually inspected to find those that cause the most severe problems in the context of the entire system. Applying the numerous *detection strategies* presented in this book (see the next three chapters) would lead to many long code listings that you, the engineer must manually inspect, which is painful and time-consuming process.

Consequently, we need a technique that helps us to (1) *assess quickly* the context of each suspect, (2) decide if the suspect needs to be urgently refactored, and (3) get insights into how this is to be done.

Next, we will introduce a powerful visualization technique called *Class Blueprint* which helps us cover this "missing link" between lists of suspects and the design fragments that need to be improved. *Class Blueprint* is a semantically enriched depiction of the internal structure of classes. It will help us to quickly grasp and discuss the internal design of classes and the way they collaborate with other classes.

Using again a medical metaphor we can say that while *detection strategies* help us to detect abnormal fragments of a system's design, the *Class Blueprint* technique helps us to perform a radiography (or a CAT scan) of suspicious design fragments and decide if and how we need to intervene.

4.2 The Class Blueprint

In this section we present a visualization to assist the understanding of classes by representing a semantically augmented call- and access-graph of the methods and attributes of classes.

We only take into account the internal static structure of a class and focus on the way methods call each other and the way attributes are accessed, and the way the classes use inheritance.

This will help us understand the structure of classes without the need to read all of their code. Classes are difficult to understand because of the following reasons:

1. Contrary to procedural languages, the method definition order in a file is not important [Dek02]. There is no simple and apparent top-down call decomposition, which is necessary to break down the complexity of understanding object-oriented code. This problem is emphasized in the context of integrated development environments (IDEs), which disconnect the class and method definitions from their physical storage medium, e.g., directories and files.
2. Classes are organized in inheritance hierarchies in which at each level behavior can be added, overridden or extended. Understanding how a derived class fits within the context of its base class is complex because late binding provides a powerful instrument to build template and hook methods that allow children's behavior to be called in the context of their ancestors. The presence of late binding leads to yoyo effects: To understand the code by following the call-flow the reader has to browse up and down the hierarchy [WH92, DRW00].

The objective of the *class blueprint* is to help a programmer to understand and develop a mental model of the classes he or she browses and to offer support for reconstructing the logical flow of method calls. In short, a *class blueprint* is a semantically enriched and layered visualization of the control-flow and access structure of classes [LD01, DL05][2].

The Principles of a Class Blueprint

A *class blueprint* is structured according to *layers* that group the methods and attributes. The nodes (varying in size depending on

[2] To locate it in the general context of cognitive models [LPLS96, vMV96], it is intended to support the *implementation plans* at the language level, i.e., working in code chunks, in this case classes and methods.

source code information of the metrics) represent a class's methods and attributes and are colored[3] according to semantic information, e.g., whether a method is abstract, overriding other methods, returning constant values, etc.

The Layered Structure of a Class Blueprint

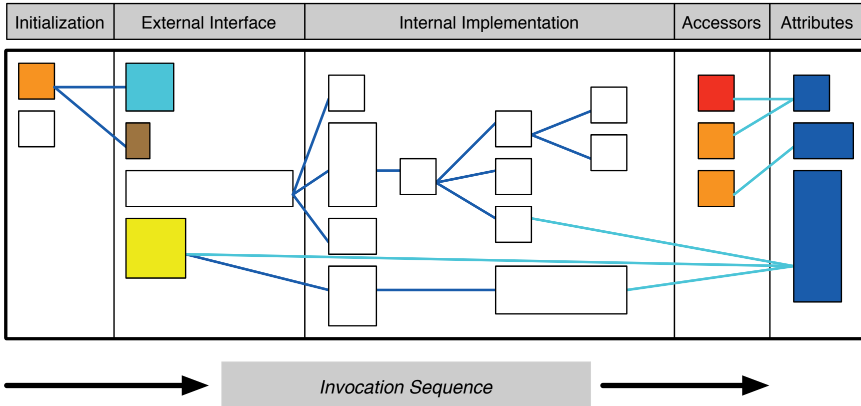


Fig. 4.6. A *class blueprint* decomposes a class into layers.

A *class blueprint* decomposes a class into layers and assigns its attributes and methods to each layer based on the heuristics described below (see Fig. 4.6). The layers support a call-graph notion in the sense that a method node on the left connected to another node on the right is either invoking or accessing the node on the right that represents a method or an attribute.

The layers have been chosen according to a notion of time-flow and encapsulation. The notion of encapsulation is visualized by separating state (to the right) from behaviour (to the left), and distinguishing the public (to the left) from the private part (to the right) of the class' behaviour. Added to this only the actual source code elements are visualized, i.e., we do not represent artificial elements resulting

[3] The colors used in our visualizations follow visual guidelines suggested by Bertin [Ber74], Tufte [Tuf90], Ware [War00], and Pinker [Pin97], e.g., we take into account that the human brain is not capable of simultaneously processing more than a dozen distinct colors.

from a combination/abstraction of source code elements. From left to right we identify the following layers: *initialization layer*, *external interface layer*, *internal implementation layer*, *accessor layer*, and *attribute layer*. The first three layers and the methods contained therein are placed from left to right according to the method invocation sequence, i.e., if method *m1* invokes method *m2*, *m2* is placed to the right of *m1* and connected to an edge.

A *class blueprint* contains the following layers:

1. **Initialization layer.** The methods contained in this first layer are responsible for creating an object and initializing the values of the attributes of the object. A method belongs to this layer if one of the following conditions holds:
   - The method is a constructor.
   - The method name contains the substrings "init(ialize)".
2. **External interface layer.** The methods contained in this layer represent the interface of a class to the outside world. A method belongs to this layer if one of the following conditions holds:
   - It is invoked by methods of the initialization layer .
   - In languages like Java and C++which support modifiers (e.g., public, protected, private) it is declared as *public* or *protected*.
   - It is not invoked by other methods within the same class, i.e., it is a method invoked from *outside* of the class by methods of collaborator classes or subclasses. Should the method be invoked both inside and outside the class, it is placed within the implementation layer .

   We consider the methods of the interface layer to be the *entry points* to the functionality provided by the class. We do not include accessor methods (getters and setters) in this layer, but in a dedicated accessor layer .
3. **Internal implementation layer.** The methods contained in this layer represent the core of a class and are not supposed to be visible to the outside world. A method belongs to this layer if the method is invoked by at least one method of the same class.
4. **Accessor layer.** This layer is composed of accessor methods, i.e., methods whose *sole* task is to get or set the values of attributes.
5. **Attribute layer.** The attribute layer contains the attributes of the class, connected to the method nodes in the other layers by edges representing *access relationships*.

Representing Methods and Attributes

We represent methods and attributes using colored boxes (nodes) of various size and position them within the layers presented previously. We map metric information to the size of the method and attribute nodes, and map semantic information on their colors.

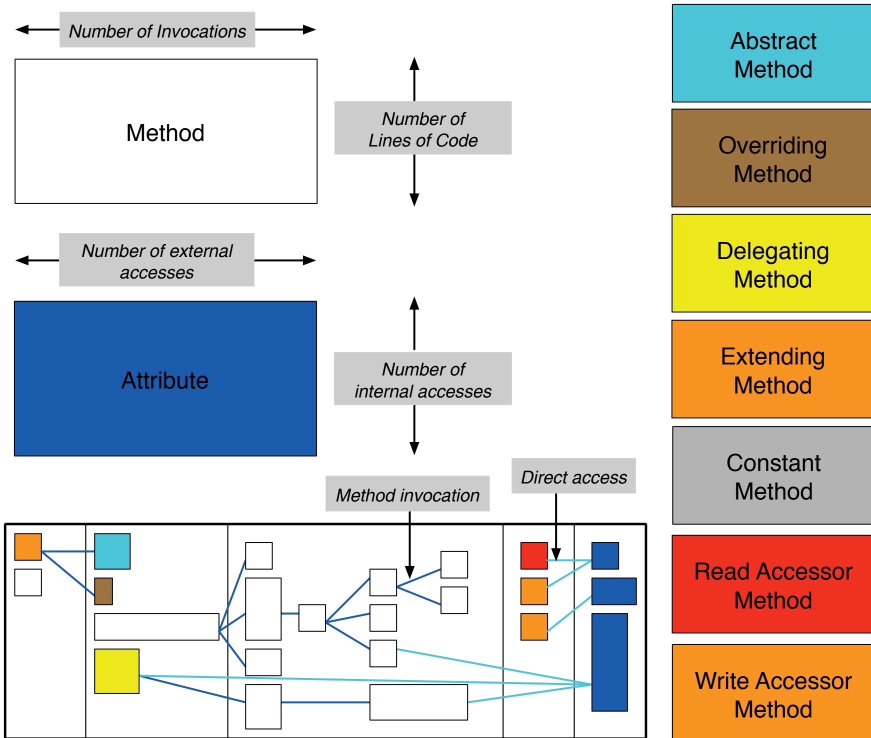


Fig. 4.7. In a *class blueprint* the metrics are mapped on the width and the height of a node. The methods and attributes are positioned according to the layer they have been assigned to.

Mapping metrics information on size. The width and height of the nodes reflect metric measurements of the represented entities, as illustrated in Fig. 4.7. In the context of a *class blueprint*, the metrics used for the method nodes are *lines of code* for the height and *number of invocations* (i.e., number of static invocation going out from the represented node) for the width. The metrics used for the attribute nodes are the number of direct accesses from methods within the

class for the width and the number of direct accesses from methods defined in other classes for the height. This allows one to identify how attributes are accessed.

Description	Color
Attribute	blue node
Abstract method	cyan node
Extending method. A method which performs a *super* invocation.	orange node
Overriding method. A method redefinition *without* hidden method invocation.	brown node
Delegating method, forwards the method call to another object.	yellow node
Constant method. A method which returns a *constant* value.	grey node
Interface and Implementation layer method.	white node
Accessor layer method. Getter.	red node
Accessor layer method. Setter.	orange node
Invocation of a method.	blue edge
Invocation of an accessor. Semantically equivalent to a direct access.	blue edge
Access to an attribute.	cyan edge

Table 4.1. In a *class blueprint* semantic information is mapped on the colors of the nodes and edges.

Mapping semantic information on color. The call-graph is augmented not only by the size of its nodes but also by their color. In a *class blueprint* the colors of nodes and edges represent semantic information extracted from the source code analysis. The colors play an important role in conveying added information [Ber74, Tuf90]. Table 4.1 presents the semantic information we add to a *class blueprint* and the associated colors.

Class Blueprints Exemplified

To show how the *class blueprint* visualization allows one to represent a condensed view of a class's methods, call-flow and attribute accesses, we describe in detail two classes implementing two different domain entities of the Jun framework: The first one defines the concept of a 3D graph for OpenGL mapping and the second is a rendering algorithm. We present the blueprints and some pieces of code to show how the graphical representation is extracted from the source code and how the graphical representation reflects the code it represents, building a trustable model.

To help the reader to understand the first *class blueprint* we also show on the right of the figure a blueprint without metrics in which the method names are shown on the boxes that represent them.

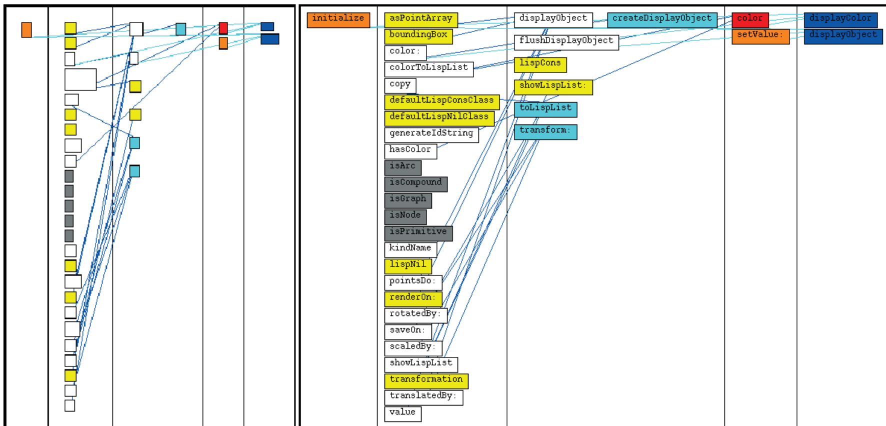


Fig. 4.8. Left: An actual class blueprint visualization of the class JunOpenGL3dGraphAbstract, a class which represents 3D graphs in OpenGL. Right: The same class displayed with method names for illustrating how the methods call each other.

The left part of Fig. 4.8 shows the blueprint of a Smalltalk class named Jun-OpenGL3dGraphAbstract which we describe hereafter. As the named blueprint on the right in Fig. 4.8 shows, this kind of representation does not scale well in practice; additionally, metrics information is not reflected in a named blueprint (i.e., the width and height of nodes is not correlated with metric value). Therefore it is not used in this book.

The code shown is Smalltalk code; however, in order to understand the code sequence being fluent in Smalltalk is not a must as we are only concerned with method invocations and attribute accesses.[4]

Example 1: An Abstract Class

The *class blueprint* shown in Fig. 4.8 has the following structure:

- **One initialization layer method.** This method, called initialize, is positioned on the left. As shown, it extends (invokes) a superclass

[4] In Smalltalk, attributes as local variables are read simply by using the attribute name in an expression. They are written using the := construct. In a first approximation, messages follow the pattern receiver methodName1: arg1 name2: arg2 which is equivalent to the Java/C++syntax receiver.methodName1name2(arg1, arg2). Hence bidiagNorm := self bidiagonalize: superDiag assigns to the variable bidiagNorm the result of the method bidiagonalize.

method with the same name, hence the node color is orange. It directly accesses two attributes, as the cyan line shows. The code of the method initialize is as follows:

```
initialize
   super initialize.
   displayObject := nil.
   displayColor := nil
```

- **Several external interface layer methods.** Note that many of them have a yellow color, i.e., they delegate the functionality. The following method asPointArray is a delegating method:

```
asPointArray
   ^ self displayObject asPointArray
```

The five grey nodes in the interface layer are methods returning constant values as illustrated by the following method isArc. This method illustrates a typical practice to share a default behavior among the hierarchy of classes.

```
isArc
   ^ false
```

- **A small internal implementation layer with two sub-layers.** This layer shows that the blueprint granularity resides at the method level, as the visualization does not specifically represent control flow constructs. The method displayObject performs a lazy initialization, i.e., it initializes the attributes only when the attributes are accessed and acts as an abstract template method by calling the method createDisplayObject which is abstract and thus represented as a cyan node. The method createDisplayObject should then be redefined in the subclasses.

```
displayObject
   displayObject isNil ifTrue:
      [ displayObject := self createDisplayObject ].
   ^ displayObject

createDisplayObject
   ^ self subclassResponsibility
```

- **Two accessors.** There is a read-accessor, color, displayed as the red accessor node and a write-accessor, setValue: displayed as the rightmost orange accessor node.

- **Two attributes.** Note that the read-accessor reads one attribute, while the write-accessor writes the other one. However, no method uses the write-accessor. The attributes are also directly accessed: the initialize method accesses both, while two other methods also directly access the attributes. which is an inconsistent coding practice.

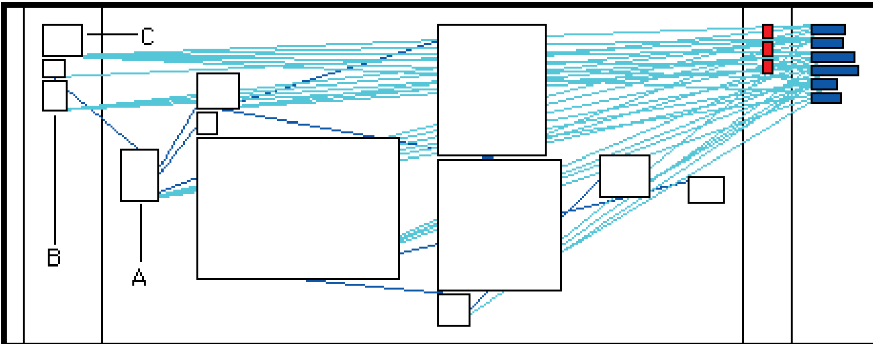


Fig. 4.9. A blueprint of the class JunSVD. This *class blueprint* shows patterns of the type *Single Entry*, *Structured Flow* and *All State*.

Example 2: An Algorithm

The *class blueprint* presented in Fig. 4.9 displays the class JunSVD implementing the algorithm of the same name. Looking at the blueprint we get the following information.

- **No initialization layer method.** The left layer is empty.
- **Three external interface layer methods.** Two of them directly access the attributes of the class. We also see that the second external interface layer method is actually an entry point to all the methods in the internal implementation layer.
- **An internal implementation layer composed of nine methods in five sub-layers.** The class is actually written in a clearly structured way. Therefore the *class blueprint* can also be used to infer a reading order of the methods contained in this class. The blueprint shows that the node *A* which represents the method compute (shown hereafter) invokes the methods bidiagonalize:, epsilon and diagonalize:with:.

```
compute
   | superDiag bidiagNorm eps   |
   m := matrix rowSize.
   n := matrix columnSize.
   u := (matrix species unit: m) asDouble.
   v := (matrix species unit: n) asDouble.
   sig := Array new: n.
   superDiag := Array new: n.
   bidiagNorm := self bidiagonalize: superDiag.
   eps := self epsilon * bidiagNorm.
   self diagonalize: superDiag with: eps.
```

- **Three read-accessor methods.** Although three read-accessors have been defined, they are not used by methods of this class, because they do not have any incoming edges that would exemplify their use.
- **Six attributes.** All the attributes in this class are accessed by several methods, i.e., all the state of the class is accessed by the methods. The blueprint also reveals that the attributes are heavily accessed. The nodes marked as *A*, *B* and *C* consistently access *all* the attributes matrix, n, m, sig, v and u. To understand how this particular behavior is possible we show the code of the method generalizedInverse (*C*). After reading the code we easily understand that this particular behavior for a class is normal for an algorithm and we mentally acknowledge that the other methods are built in a similar fashion.

```
generalizedInverse
   | sp |
   sp := matrix species new: n by: m.
   sp doIJ: [:each :i :j |
     sp row: i column: j put:
       ((i = j and: [(sig at: j) isZero not])
           ifTrue: [(sig at: j) reciprocal]
           ifFalse: [0.0d])].
   ^ (v product: sp) product: u transpose
```

This example shows that the blueprint visualization conveys information which is otherwise hard to notice: all attributes are accessed by the methods. This is an example of how the approach supports opportunistic code reading. First the reader is intrigued by the regularity of the accesses, then reads one method and understands that the methods implement an algorithm. The reader can now extrapolate this knowledge to the other methods of the class.

Example 3: Class Blueprints and Inheritance

Understanding classes in the presence of inheritance is difficult as the flow of the program is not local to a single class but distributed over hierarchies, as mentioned by Wild [WH92] and Lange [LN95]. In the context of inheritance we visualize every *class blueprint* separately and put the subclasses below the superclasses according to a simple tree layout.

In Fig. 4.10 we see a concrete inheritance hierarchy of class blueprints. The superclass defines some behavior that is then specialized by each of the three subclasses named JunColorChoiceHSB, JunColorChoiceSBH and JunColorChoiceHBS. The blueprint of this hierarchy reveals that the subclasses have been developed to satisfy the implementation needs of the superclass: they do not define any extra behavior; it is the superclass that must be analyzed to understand the whole hierarchy.

We see that the root class defines several abstract methods (denoted by the cyan color) that represent color components such as brightness, hue and color and which are overridden (denoted by the brown color) in the three small subclasses. As there is the same number of brown nodes as cyan ones, there is a good chance that the subclasses are concrete classes.

The method named color is a template method that calls three abstract methods as confirmed by the definition of the method color:

```
color
    ^ ColorValue hue: self hue
                 saturation: self saturation
                 brightness: self brightness
```

We see that the methods xy: (B) and xy (C), play a central role in the design of the class as they are both called by several of the methods of each subclass, as confirmed by the following method of the class JunColorChoiceSBH:

```
JunColorChoiceSBH>>brightness: value
    ((value isKindOf: Number) and:
       [0.0 <= value and: [value <= 1.0]])
          ifTrue: [self xy: self xy x @ 1 - value]
```

This example shows again that the blueprint conveys information which is otherwise hard to notice, e.g., the fact that all the subclasses of the root classes implement only methods which override methods

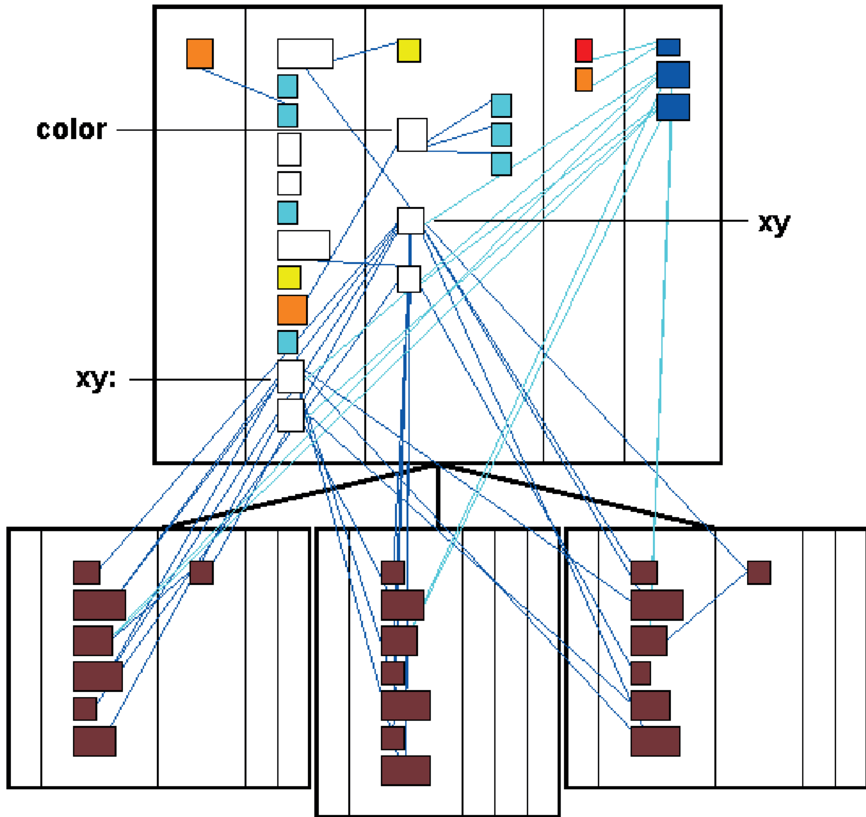


Fig. 4.10. A *class blueprint* visualization of an inheritance hierarchy with the class JunColorChoice as root class. The root class contains an *Interface* visual pattern, while each of the subclasses is a pure *Overrider*. Furthermore, each subclass is a pure *Siamese Twin*.

in the superclass, or it helps to detect the template method design pattern present in the root class.

All these examples illustrate how the blueprints help a software engineer to: (1) build a mental image of the class in terms of method invocations and state access, (2) understand the class/subclass roles, and (3) identify key methods.

Blueprints act as revealers in the sense that they raise questions, support hypotheses, or clearly show important information. When questions are raised, code reading helps confirm the information provided by the visualization. Code reading is not always necessary, but can be used sparingly on identified methods. There is a definitive

synergy between the visual images generated by the blueprint and the code reading. Class blueprints allow one to characterize classes but also represent an important means of communication.

Example 4: Class Blueprints and Design Problems

Class Blueprints provide us with a powerful visual means to inspect the suspects detected by the detection strategies. For example, by applying the *God Class* detection strategy (see page 51) on a case study we found several suspects, one of which is class Modeller.

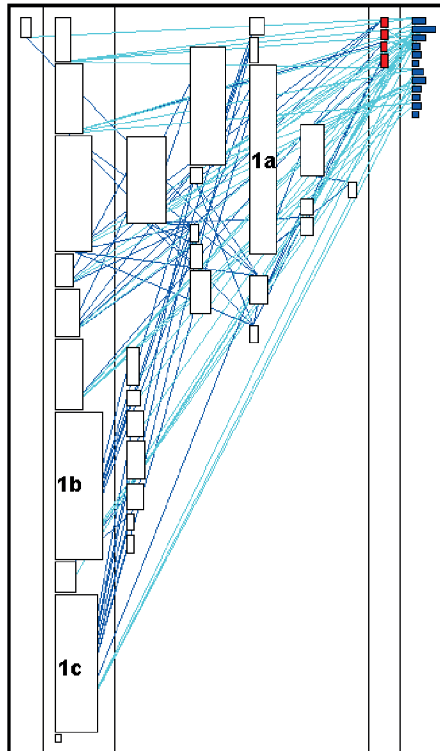


Fig. 4.11. The *class blueprint* of a *God Class* suspect.

By building the *class blueprint* for this class (see Fig. 4.11) we can immediately see that Modeller is not a class with an excessive number of methods, but has a certain number of considerably large and complex methods (3 methods are longer than 100 lines of code, the longest one addDocumentationTag (annotated as 1a in the figure)

is 150 lines code and invoked by three other methods, two of which are the second and third longest methods in this class: addOperation (1b, 116 LOC) and addAttribute (1c, 108 LOC). The *class blueprint* reveals other disharmonies in this class: there are 12 attributes in this class, all of them private (which is good), but there are "only" 4 accessor methods. Moreover, the attributes are accessed both directly and indirectly (using the accessors), denoting a certain inconsistency or lack of access policy.

4.3 Conclusions and Outlook

In this chapter we presented two approaches which will allow us to evaluate the design of object-oriented software systems, the *Detection Strategy* and the *Class Blueprint*:

Detection Strategy: It provides us with a means to detect flawed (from a design point of view) entities in object-oriented systems. The design strategies produce lists of suspects that comply with specific heuristics encoded with metrics.

Class Blueprint: It provides us with a powerful visual means to inspect the suspects detected by the *detection strategies*.

In the beginning of this chapter (see 46) we argued that metrics can help to evaluate designs, but those have to be *meaningful* metrics that are put in the context of rules, best practices and heuristics that express the *harmony of a design*.

Although we partially agree with Fowler stating that "no set of metrics rivals informed human intuition" [FBB+99], there is a big disadvantage: human intuition does not *scale* with the dimensions of today's software systems. Therefore, in order to find and improve disharmonious design fragments in the next three chapters we employ *detection strategies* and the *class blueprint*.

Consequently in the remaining chapters, we present in detail 11 such *design disharmonies*. For each of them we describe the *detection strategy* that helps to detect them automatically using metrics, we look at selected examples using the *class blueprint*, and conclude each disharmony with a discussion of how to *cure* flawed entities using refactorings.

Based on the harmony aspects identified in Sect. 4, we divide the 11 *design disharmonies* in three categories, i.e., *identity collaboration* and *classification* disharmonies:

Identity Disharmonies (Chapter 5): God Class(80), Brain Class(97), Feature Envy(84), Brain Method(92), Data Class(88), Duplication(102)

Collaboration Disharmonies (Chapter 6): Dispersed Coupling(127), Intensive Coupling(120), Shotgun Surgery(133)

Classification Disharmonies (Chapter 7): Refused Parent Bequest(145), Tradition Breaker(152)

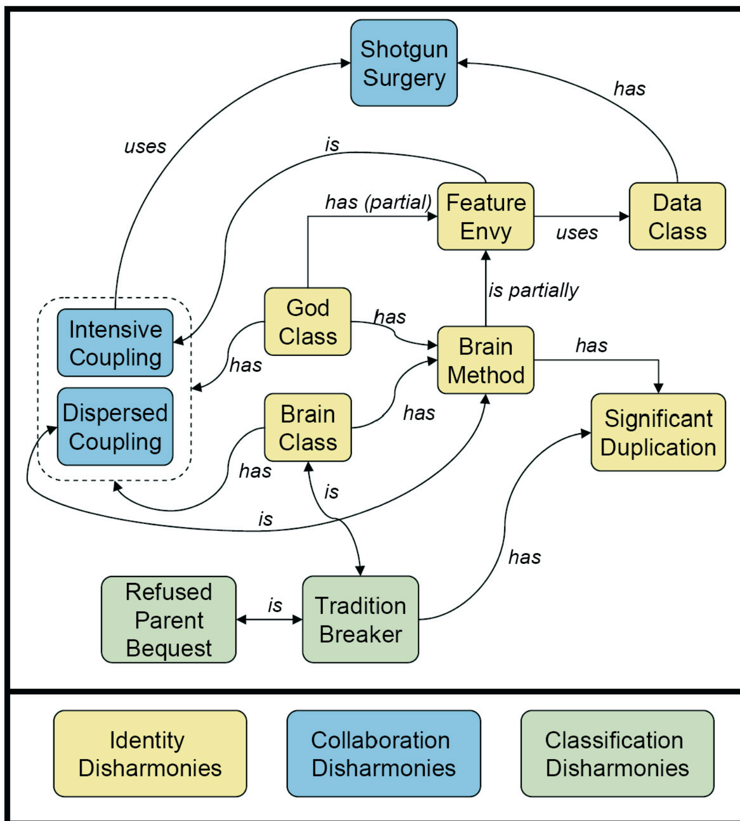


Fig. 4.12. Disharmonies and their correlations.

Each of the following chapters has four major parts:

1. *Harmony Rule(s).* As mentioned before, disharmonies are deviations from a set of principles, rules and heuristics that specify what harmony means. Therefore, before presenting a catalogue of disharmonies, we summarize in the form of one or more *harmony*

rules those aspects of harmony that we took into account when building the catalogue of disharmonies. These *harmony rules* are a concise distillation of various design rules and heuristics found in the literature (e.g., [Rie96, Mar02b, JF88, Lak96, Mey88b, Lis87]).

2. *Overview of Detected Disharmonies*. Most of the times design disharmonies do not appear in isolation. Therefore, before presenting the disharmonies one by one, we provide a brief overview in which we reveal the most common correlations between the various disharmonies. Apart from discussing the correlations, we provide in each case a picture that captures the *web of correlations* involving the disharmonies presented in that chapter. As a sneak preview, in Fig. 4.12 you can see all disharmonies and their most common correlations. complete *web of correlations*
3. *Catalogue of Disharmonies*. The central part of each chapter consists of a catalogue of specific disharmonies that can be detected using a metrics-based approach. Each disharmony is described in a in a pattern-like format.
4. *Summary*. Each chapter ends with a suite of practical guidelines on detecting and recovering from the disharmonies presented in the chapter.

http://www.springer.com/978-3-540-24429-5

Object-Oriented Metrics in Practice
Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems
Lanza, M.; Marinescu, R.
2006, XIV, 206 p. 80 illus., 33 in color., Hardcover
ISBN: 978-3-540-24429-5