

Identity Disharmonies

Identity disharmonies are design flaws that affect single entities such as classes and methods. The particularity of these disharmonies is that their negative effect on the quality of design elements can be noticed by considering these design elements *in isolation*.

5.1 Rules of Identity Harmony

As mentioned at the end of Chapter 4 before presenting the various identity disharmonies, let us first take a closer look on the most important harmony rules related to a single design entity.

We identified three distinct aspects that contribute to the identity (dis)harmony of a single entity: its size, its interface and its implementation. We summarize each of these aspects in the form of a rule, a rationale and a set of practical consequences. The three rules of identity harmony that we defined are:

Operations and classes should have a harmonious size i.e., they should avoid both size extremities

Each class should present its identity (i.e., its interface) by a set of services, which have one single responsibility and which provide a unique behavior

Data and operations should harmoniously collaborate within the class to which they semantically belong

Proportion Rule

Operations and classes should have a harmonious size, i.e., they should avoid both size extremities

Rationale

When considering quality and harmony the first aspect we think about is proportion. The same applies to object-oriented software design. While this first rule is simple to understand it is crucial to follow it. Most of the maintenance and reuse problems come from an unbalanced distribution of a system's complexity (responsibilities) among classes [Rie96, WBM03] or among operations [FBB⁺99]. This does not mean that all classes or operations must have the same size; rather, it warns us about the danger of going to extremes. Both extremes can be dangerous: too large classes or operations are a maintenance nightmare, while many tiny classes are in most cases a sign of *class proliferation* and hinder understanding. In the same manner, while it is desirable to have slim operations, sometimes this is abused and we end up with an excessive number of methods, that again hampers maintenance.

Presentation Rule

Each class should present its identity (i.e., its interface) by a set of services, which have one single responsibility and which provide a unique behavior

Rationale

This rule encourages a balanced distribution of a system's intelligence among classes [Rie96, WBM03] and was the underlying idea behind CRC cards (class, responsibility, collaborator) [BS97]. The aim of the rule is to focus each class on a single task (responsibility), expressed in terms as a set of services (i.e., a set of public methods). The rule makes sure that each concrete piece of functionality is implemented once and only once in the system to avoid code duplication.

Practical Consequences

- **Provide services and hide data** – *A class should present itself to others only in terms of a set of provided services (i.e., public methods). Never let a class present itself in terms of its data, as this breaks encapsulation and consequently spoils the maintenance benefits of object-oriented design.*
- **Take responsibility** – *Most non-abstract services of a class should be responsible for implementing a piece of the class's functionality. A class might have some delegator (i.e., methods that just forward the call to another method) and some accessor methods, but the number of such methods should be limited in each class.*
- **Keep services cohesive** – *Services provided by a class should be focused on one single responsibility. The set of services of a class should have limited size and a high usage cohesion.*

This is a restatement of the *Interface Segregation Principle* [Mar02b] which states that the clients of a class should not be forced to

depend on interfaces that they do not use. Although this consequence also concerns the size of the interface of a class, the main aspect here is not the proportion, but the avoidance of an eclectic interface.

- **Be unique** – *When classes have a harmonious identity, then each piece of concrete functionality has a unique place, i.e., it is implemented once and only once [Bec00]. Consequently, code duplication is avoided.*

Implementation Rule

Data and operations should collaborate harmoniously within the class to which they semantically belong

Rationale

One of the cornerstones of the object-oriented paradigm is encapsulation that makes sure that the data and the operations are kept together. An abstraction (e.g., a class) is harmonious if its operations use most of data most of the time [Rie96], i.e., if most attributes of a class are used together in most methods of that class. This rule does not allow every piece of the system to be visible and accessible by any other part of the system. Applying the law of Demeter [LH89] stresses that locality of data access is important to avoid to have ripple effects when code changes. Hence keeping data and behavior together is a good practice and an important refactoring [DDN02].

Practical Consequences

- **Operations belong to classes** – *Every operation should belong to a class. Thus, avoid as much as possible global operations.*
- **Keep data close to operations** – *Data and the operations that use it most should be placed as close as possible to one another. In other words, data (e.g., attributes, local variables, etc.) should stay in the class or method where they are used the most.*
- **Distribute complexity** – *The functionality provided by a class should be distributed among its operations in a balanced manner.*
- **Operations use most attributes** – *Within the same class, most operations should collaborate and use most of the data most of the time [Rie96]. Thus, avoid abstractions with disjunct sets of behavior and data.*

5.2 Overview of Identity Disharmonies

The most frequent and easily recognizable sign of an identity disharmony is excessive size and complexity of a class and its methods (*Proportion Rule*). Any investigation that intends to assess and improve the identity harmony of a system usually starts with those classes and methods that stand out due to their size. This is very important, because as we will see also in Sect. 5.9 the process of recovering from design problems uses these outlying design fragments as a starting point.

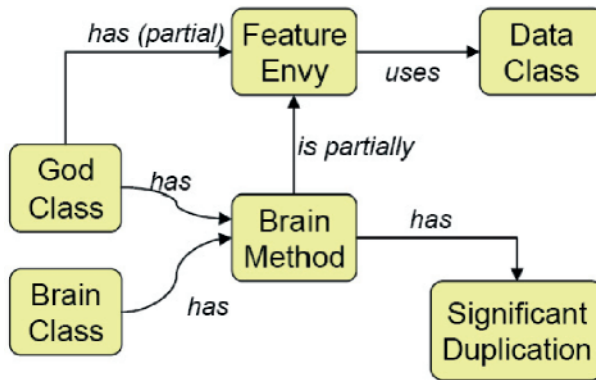


Fig. 5.1. Correlation web between identity disharmonies.

In the remainder of this chapter we present *detection strategies* that capture oversized and overcomplex methods (Brain Method(92)) and the classes that host them (Brain Class(97)). In many cases these outliers are caused by the presence of code duplication; consequently we check for code duplication within classes (Duplication(102)) with excessive size and complexity (see Fig. 5.1).

Another sign of disharmonious identity is the non-cohesiveness of behavior (*Presentation Rule* and *Implementation Rule*) and the tendency to attract more and more features, to gather more and more services (Riel calls such a disharmony a God Class(80) [Rie96]). We defined a *detection strategy* to detect such classes. The more a class tends to become a God Class(80), the more the other classes communicating with it tend to become simple data providers. A data provider does not offer much functionality; instead it merely provides raw data and tends to become a Data Class(88) [Rie96, FBB⁺99]. As an imme-

diate consequence, the methods of the (God) classes, which use the foreign data, smell of Feature Envy(84) [FBB⁺99], being more interested in the attributes of other classes than those of their own class.

5.3 God Class

Description

In a good object-oriented design the intelligence of a system is uniformly distributed among the top-level classes [Rie96]. The God Class design flaw refers to classes that tend to centralize the intelligence of the system. A God Class performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes. This has a negative impact on the reusability and the understandability of that part of the system. This design problem is comparable to Fowler's *Large Class* bad smell [FBB⁺99].

Applies To

Classes.

Impact

God Class is potentially harmful to a system's design because it is an aggregation of different abstractions and (mis)use other classes (often mere data holder) to perform its functionality (see *Proportion* and *Implementation Rules*). Most of the time they are against the basic principles of object-oriented design which is that one class should have one responsibility. At this point it is important to mention that a God Class is a *real* problem if it hampers the evolution of the software system. Thus a class that has the structural characteristics of a God Class but that resides in a stable and untouched part of the system does *not* pose a problem!

Detection

The detection of a God Class is based on three main characteristics (Fig. 5.2):

1. They heavily access data of other simpler classes, either directly or using accessor methods.
2. They are large and complex
3. They have a lot of non-communicative behavior i.e., there is a low cohesion between the methods belonging to that class.

We first detect the classes that strongly depend on the data of other classes, as this is the most significant symptom of a God Class. After that, we filter the first list of suspects by eliminating all the small and cohesive classes. Small classes are eliminated because they are less relevant, while cohesive classes are excused because a high cohesion is a sign of internal harmony between the parts of the class. The *detection strategy* is composed of the following heuristics:

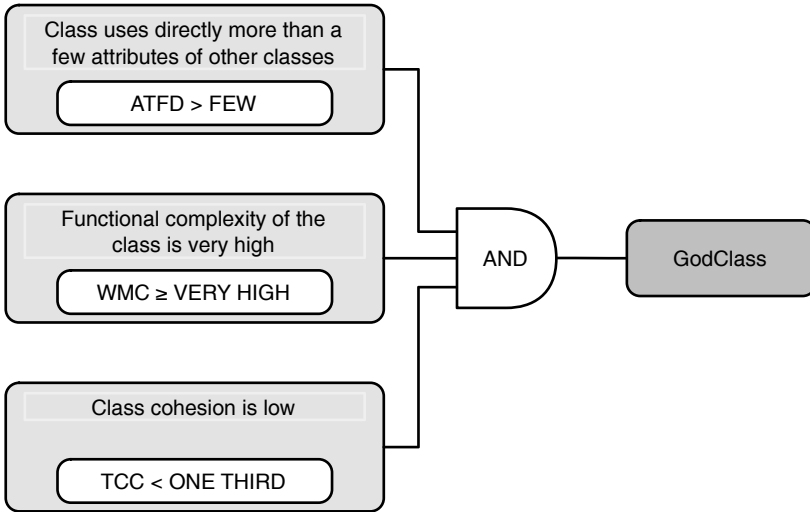


Fig. 5.2. The God Class detection strategy

1. **Class uses directly more than a few attributes of other classes.** Since ATFD measures how many foreign attributes are used by the class, it is clear that the higher the ATFD value for a class, the higher is the probability that a class is (or is about to become) a God Class.
2. **Functional complexity of the class is very high.** This is expressed using the WMC (Weighted Method Count) metric.
3. **Class cohesion is low.** As a God Class performs several distinct functionalities involving disjunct sets of attributes, this has a negative impact on the class's cohesion. The threshold indicates that in the detected classes less than *one-third* of the method pairs have in common the usage of the same attribute.

The general design of *ArgoUML* is good enough so that we could not identify a pure God Class i.e., a class controlling the flow of the application and concentrating all the crucial behavior, which would indicate a clear lack of object-oriented design. However, certain classes in *ArgoUML* acts as a black hole attracting orphan functionalities. Such classes are also detected by the metrics presented above and are still a design problem. A class of *ArgoUML* which clearly stands out is the huge class *ModelFacade* (see Fig. 3.12). This class implements 453

Example

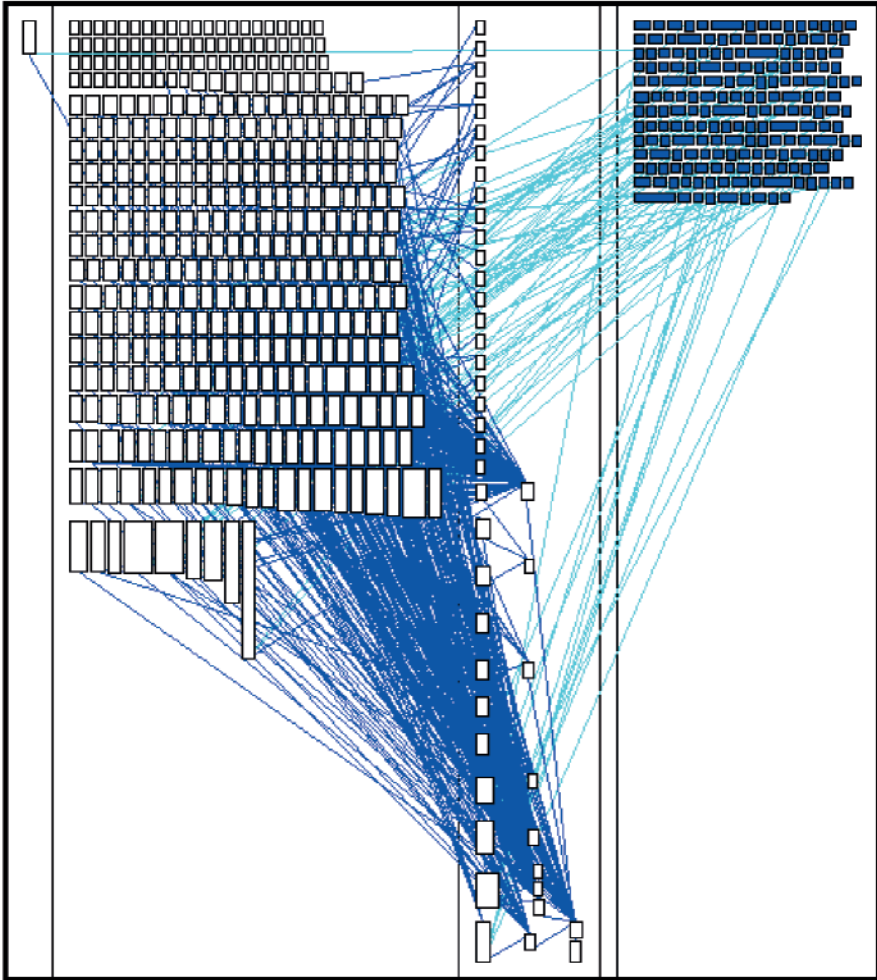


Fig. 5.3. The *Class Blueprint* of ModelFacade

methods, defines 114 attributes, and is more than 3500 lines long. Moreover, all methods and all attributes are static. Its name hints at being an implementation of the *Facade* Design Pattern [GHJV95], but it has become a sort of black hole of functionality. In Fig. 5.3 we see its *Class Blueprint* with a modified layout for the methods and attributes to make this *Class Blueprint* fit on one screen. Looking at the *Class Blueprint* for this class it seems that the developers use it for everything that does not fit into other classes, but the downside is that this class is like a tumor within this system and can only

be removed if somebody makes a great effort to break away pieces of functionality and separate them into other classes. We see from the visualization that many invocations are directed towards distinct methods, pointing to subsets of connected methods that can be extracted.

Refactoring a God Class is a complex task, as this disharmony is often a cumulative effect of other disharmonies that occur at the method level. Therefore, performing such a refactoring requires additional and more fine-grained information about the methods of the class, and sometimes even about its inheritance context. A first approach is to identify clusters of methods and attributes that are tied together and to extract these islands into separate classes. Split Up God Class [DDN02] proposes to incrementally redistribute the responsibilities of the God Class either to its collaborating classes or to new classes that are pulled out of the God Class. Feathers [Fea05] presents some techniques such as Sprout Method, Sprout Class, Wrap Method to be able to test legacy system that can be used to support the refactoring of God Classes.

5.4 Feature Envy

Description Objects are a mechanism for keeping together data and the operations that process that data. The Feature Envy design disharmony [FBB⁺99] refers to methods that seem more interested in the data of other classes than that of their own class. These methods access directly or via accessor methods a lot of data of other classes. This might be a sign that the method was misplaced and that it should be moved to another class.

Applies To Methods.

Impact Data and the operations that modify and use it should stay as close together as possible. This data-operation proximity can help minimize ripple effects (a change in a method triggers changes in other methods and so on; the same applies for bugs, i.e., in case of a poor data-operation proximity bugs will also be propagated) and help maximize cohesion (see Implementation Rule).

Detection The detection is based on counting the number of data members that are accessed (directly or via accessor methods) by a method from outside the class where the method under investigation is defined. Feature Envy happens when the envied data comes from a very few classes or only one class. The *detection strategy* (Fig. 5.4) in detail is:

1. **Method uses directly more than a few attributes of other classes.** We use the ATFD¹ (Access To Foreign Data) metric for this.
2. **Method uses far more attributes from other classes than its own.** We use the LAA (Locality of Attribute Accesses) metric for this.
3. **The used “foreign” attributes belong to very few other classes.** We use the FDP (Foreign Data Providers) metric for this. The rea-

¹ In defining the God Class(80) *detection strategy* we also used a metric called ATFD, which counts how many distinct attributes from other classes are accessed by the measured design entity. The only difference is that in the God Class(80) the metric is defined for a class entity, while here it is defined for a method entity.

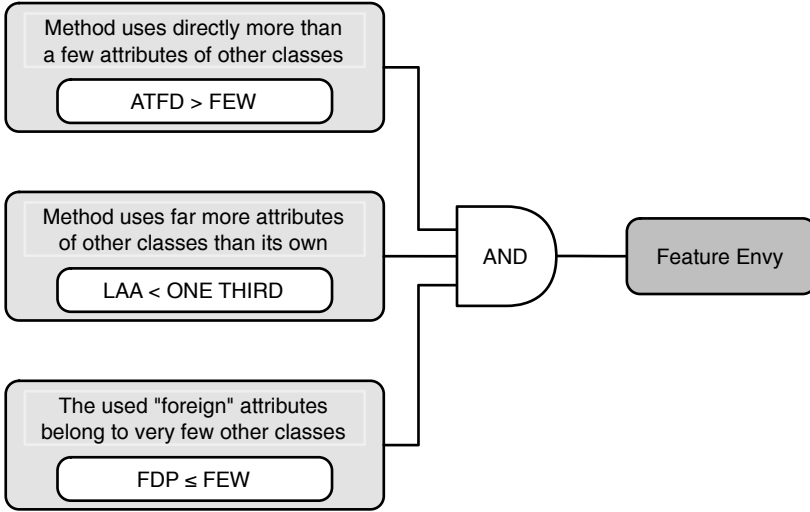


Fig. 5.4. Detection strategy for Feature Envy.

son for introducing this condition is that we want to make a distinction between a method who uses directly data from many different classes, and the case where the method envies especially 1-2 classes. In the first case, it might be that the method acts like a controller [Rie96] and/or that it is a Brain Method(92). But in detecting Feature Envy we are more interested in the second case, as the essence of this disharmony is that the affected method is simply misplaced, and this is reflected by a well targeted dependency on the data from another class.

In analyzing this design disharmony two alternative detection approaches could be used:

1. **Count all dependencies.** Another way to detect Feature Envy would be to consider *all* the dependencies of the measured method, instead of considering only the *data members* accessed by a particular method. In this case we would count both the dependencies on the class where the method is defined, and those on the other classes defined in the system.
2. **Ignore dispersion.** We used the FDP metric in the *detection strategy* because we were focused on detecting those methods that can be easily moved to another class and this involves a reduced

dispersion of the classes on which the methods rely. We might want sometimes to eliminate this restriction and in this case we will again find methods that rely on data taken from *many* other classes. Although in this case moving the method is not obvious, such methods might still require refactoring.

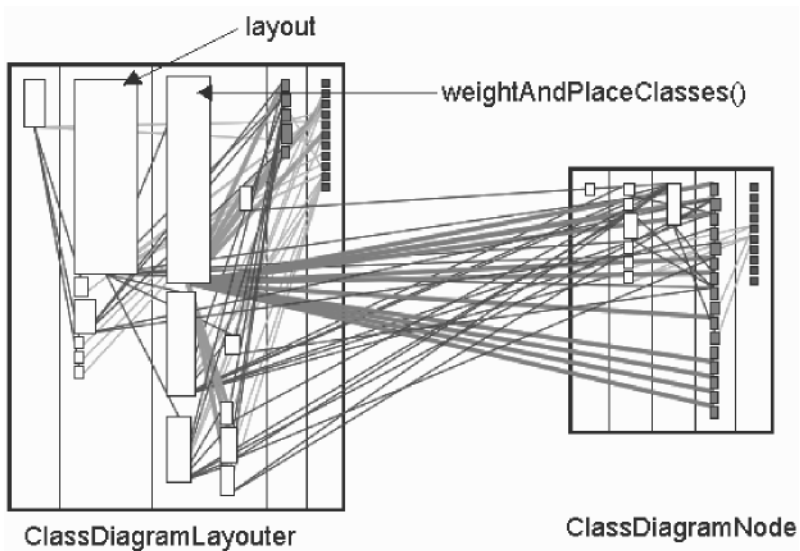


Fig. 5.5. `ClassDiagramLayouter` is envying the features of `ClassDiagramNode`. In red we colored the invocations that `weightAndPlaceClasses` performs towards `ClassDiagramNode`, while in green we see its class-internal invocations and accesses.

Example

Looking again at the *ArgoUML* system we find a good example of Feature Envy, namely the `weightAndPlaceClasses` method in the class `ClassDiagramLayouter`. Although the method uses data from its own class it envies the data encapsulated in the class `ClassDiagramNode` (i.e., by accessing the data heavily via a large number of accessor methods), as depicted in Fig. 5.5. Looking closer at the figure we notice three interesting aspects:

1. The `weightAndPlaceClasses` method is excessively large.
2. The envied class, i.e., `ClassDiagramNode`, contains almost no functionality, but just data which is made accessible via the accessor methods (marked in red). A problem is that the envied class

does not provide a clean interface to clients to offer them functionality, but it exposes its attributes, which is questionable.

3. Looking at `ClassDiagramLayouter` we notice that the method layout is using several attributes from `ClassDiagramNode`.

These three observations illustrate the most significant aspect about Feature Envy, namely that it is a sign of an improper distribution of a system's intelligence. While the `ClassDiagramLayouter` is an excessively complex class (i.e., a Brain Class(97)) with a very high average complexity of methods (AMW = 5.25) the `ClassDiagramNode` class contains very little functionality, being a Data Class(88). For comparison let us mention that the average complexity of methods in this class, namely the values of the AMW metric, is as low as 1.33.

Finally, as we see in this example, often a Feature Envy method also has some dependencies on its own class, and not only on the envied class. This tells us that in order to recover from this problem, it is very rare that we can move the whole method to the other class. Rather, it is more often that a part of the method can be extracted and moved to the envied class (for a more detailed discussion see also Sect. 5.9).

This problem can be solved if the method is moved into the class to which it is coupled the most. If only a part of the method suffers from a Feature Envy it might be necessary to extract that part into a new method and after that move the newly created method into the envied class. If the method envies two different classes, you should move it to the one that it uses most.

Refactoring

Oftentimes, the class that a method affected by Feature Envy is depending on is a class with not much functionality, sometimes even a Data Class(88). If this is a case then moving the Feature Envy method to that class is even more a desirable refactoring, as it re-balances the distribution of functionality among class and improves the data-behavior locality.

The concrete refactoring technique for Feature Envy is based on the Move Method and Extract Method refactorings [FBB⁺99]. Furthermore, the Move Behavior Close to the Data reengineering pattern [DDN02] discusses the steps to follow to move behavior close to the data it uses and the potential difficulties.

5.5 Data Class

Description Data Classes [FBB⁺99] [Rie96] are “dumb” data holders without complex functionality but other classes strongly rely on them. The lack of functionally relevant methods may indicate that related data and behavior are not kept in one place; this is a sign of a non-object-oriented design. Data Classes are the manifestation of a lacking encapsulation of data, and of a poor data-functionality proximity.

Applies To Classes.

Impact The principles of encapsulation and data hiding are paramount to obtain a good object-oriented design. Data Classes break design principles because they let other classes see and possibly manipulate their data, leading to a brittle design (Presentation Rule). Such classes reduce the maintainability, testability and understandability of a system.

Detection We detect Data Classes based on their characteristics (see Fig. 5.6): we search for “lightweight” classes, i.e., classes which provide almost no functionality through their interfaces. Next, we look for the classes that define many *accessors* (get/set methods) and for those who declare data fields in their interfaces. Finally, we confront the lists and manually inspect the lightweight classes that declare many public attributes and those that provide many accessor methods. The *detection strategy* in detail is:

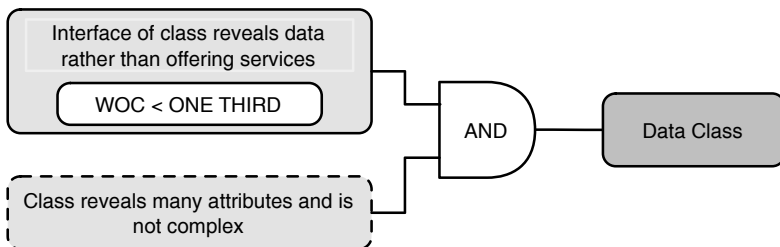


Fig. 5.6. The Data Class detection strategy.

1. Interface of class reveals data rather than offering services.

The large majority of the class's interface is exposing data rather than providing services. We use the WOC (Weight Of Class) metric for this.

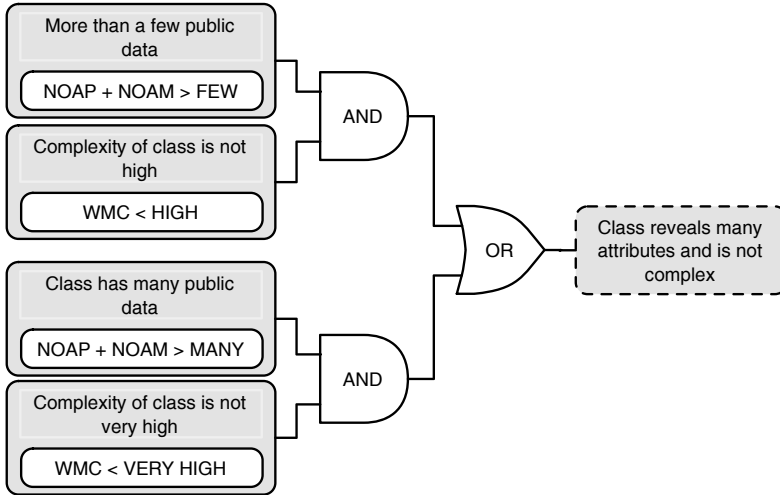


Fig. 5.7. Data Class reveals many attributes and is not complex.

2. **Class reveals many attributes and is not complex.** The WOC metric makes sure that the interface of the class is occupied mainly by data and accessor methods. We also want to be sure that the *absolute* number of these encapsulation breakers is high. We differentiate between two cases (see Fig. 5.7):

- a) The classical Data Class is not very large, has almost no functionality, and only provides some data and data accessors. In this case the class has not a high WMC (Weighted Method Count) value, and we cannot expect to find much public data. Therefore, the only request is that the class has more than a FEW public data holders, expressed using the NOPA (Number Of Public Attributes) and NOAM (Number Of Accessor Methods) metrics.
- b) The other case is that of a rather large class that apparently looks “normal” (i.e., it does also define some functionality), except for the fact that its (large) public interface contains, apart

from the provided services, a significant number of data and data accessors. For this case, in order to consider the class a Data Class, we require that it provides MANY public data. At the same time, we allow the complexity of the class (WMC) to be considerably high, up to the limit of excessively high (because a class with extremely high complexity does not conceptually fit the Data Class term).

Example

In *ArgoUML* we identified several examples of Data Classes, one of which is the class *Property* (see Fig. 5.8).

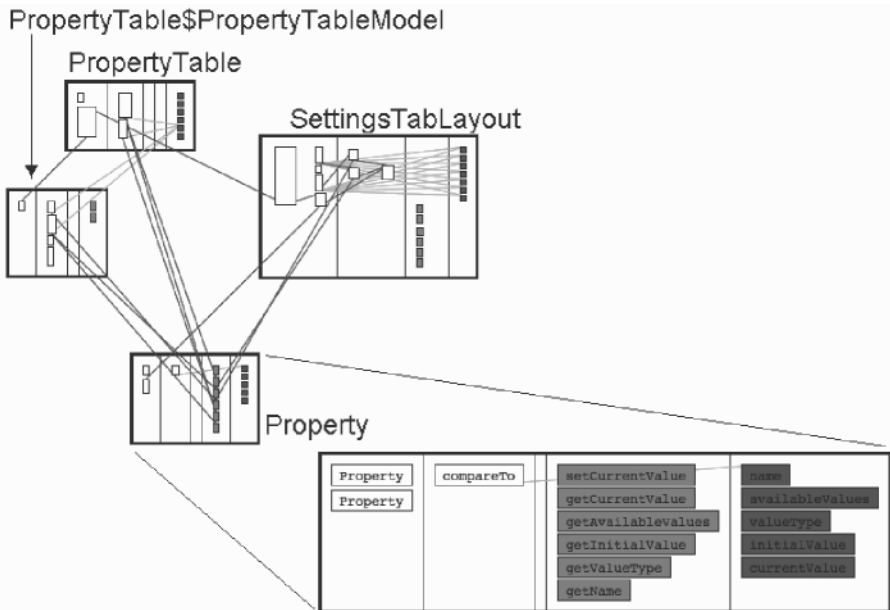


Fig. 5.8. An example of a Data Class: *Property*.

The name itself already suggests that the class is not really modelling an abstraction in the system, but rather keeps together a set of data. Looking closer, we notice that the class has five attributes. In Fig. 5.8 we depict the *Property* class together with the classes that use its data. In spite of the fact that all attributes are declared as private, the class is still a pure data holder, due to the fact that all (but one) of its methods are accessors (see methods in red). Thus, the class has no behavior, it just keeps some data, used by three other classes. Although none of the involved classes are large, the fact that data

and behavior are separated makes that design fragment harder to understand and to maintain. The fact that in this class all attributes are private, is a good example of how accessor methods can obey the principle of data hiding and still let the class be a pure data holder. Speaking about Data Class examples, let us revisit a previous example presented in the context of Feature Envy(84) (see Fig. 5.5 on page 86) in which class `ClassDiagramLayouter` was envying the attributes of `ClassDiagramNode`. The Feature Envy problem is mainly due to the fact the `ClassDiagramNode` is a Data Class(88), and thus its behavior and data are not part of the same class. This reveals an often encountered relation between the two aforementioned disharmonies: a Data Class(88) will make the classes that are using it to envy its data; or, the other way around: when a method is affected by Feature Envy(84), it is rather probable that we will find Data Classes among the classes from which that method accesses data.

The basic idea of any refactoring action on a Data Class is to put together in the same class the data and the operations defined on that data, and to provide proper services to the former clients of the public data, instead of the direct access to this data.

Refactoring

- This *data-operation proximity* (see Implementation Rule) can be achieved in most of the cases by analyzing how clients of the Data Class use this data. In this way we can identify some pieces of functionality (behavior) that could be extracted and moved as services to the Data Class. This refactoring action is very much related to what needs to be done when Feature Envy(84) is encountered. In other words, when refactoring a case of Feature Envy(84), this could lead to a positive effect towards repairing a envied Data Class.
- In some other cases, especially if the Data Class is dumb and has only one or a few clients, we could remove the class completely from the system and put the data it contains in those classes (former clients) where the best *data-operation proximity* is achieved.
- If the Data Class is a rather large class with some functionality, but also with many exposed attributes, it is very possible that only a part of the class needs to be cured. In some cases this could mean extracting the disharmonious parts together to a separate class and applying the classical treatment, i.e., trying to extract pieces of functionality from the data clients as services provided by the new class.

5.6 Brain Method

Description Often a method starts out as a “normal” method but then more and more functionality is added to it until it gets out of control, becoming hard to maintain or understand. Brain Methods tend to centralize the functionality of a class, in the same way as a God Class(80) centralizes the functionality of an entire subsystem, or sometimes even a whole system.

Applies To Operations, i.e., methods or standalone functions.

Impact A method should avoid size extremities (Proportion Rule). In the case of Brain Methods the problem concerns overlong methods, which are harder to understand and debug, and practically impossible to reuse. A well-written method should have an adequate complexity which is concordance with the method’s purpose (*Implementation Rule*).

Detection The strategy for detecting this design flaw (see Fig. 5.9) is based on the presumed convergence of three simple code smells described by Fowler [FBB⁺99]:

- *Long methods* – These are undesirable because they affect the understandability and testability of the code. Long methods tend to do more than one piece of functionality, and they are therefore using many temporary variables and parameters, making them more error-prone.
- *Excessive branching* – The intensive use of switch statements (or if–else–if) is in most cases a clear symptom of a non-object-oriented design, in which polymorphism is ignored.²
- *Many variables used* – The method uses many local variables but also many instance variables.

The *detection strategy* in detail is:

² The excessive use of polymorphism also introduces testability and analyzability problems [Bin99]. Yet, the emphasis in the context of this design flaw is on a very frequent case in which legacy systems migrated from structured to object-oriented programming.

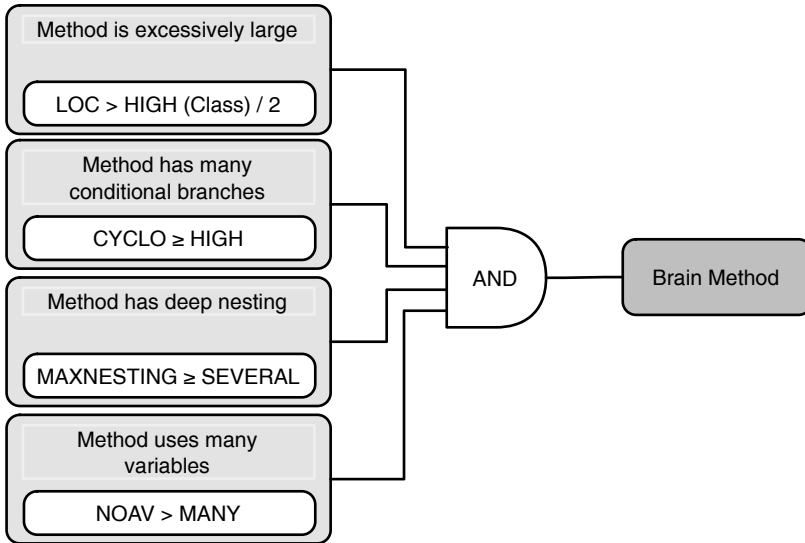


Fig. 5.9. The Brain Method detection strategy.

1. **Method is excessively large.** We are looking for excessively large methods. Based on our practical experience, we used the following heuristic to set the threshold: a method is considered to be excessively large if its LOC count is higher than half of the statistical HIGH threshold for classes (see Table 2.2 for the LOC count of classes)³.
2. **Method has many conditional branches.** This is computed using the CYCLO (McCabe's Cyclomatic Complexity) metric.
3. **Method has deep nesting level.** This is computed using the MAXNESTING (Maximum Nesting Level) metric i.e., the maximum nesting level of control structures within a method or function.
4. **Method uses many variables.** Method uses more variables than a human can keep in short-term memory. Exceeding this limit always raises the risk of introducing bugs. Notice that all types of variables are counted including local variables, parameters, but also attributes and global variables (in programming languages where this is unfortunately possible). We used NOAV (Number Of Accessed Variables) to compute this.

³ Only the lines of code in the methods of the class are counted.

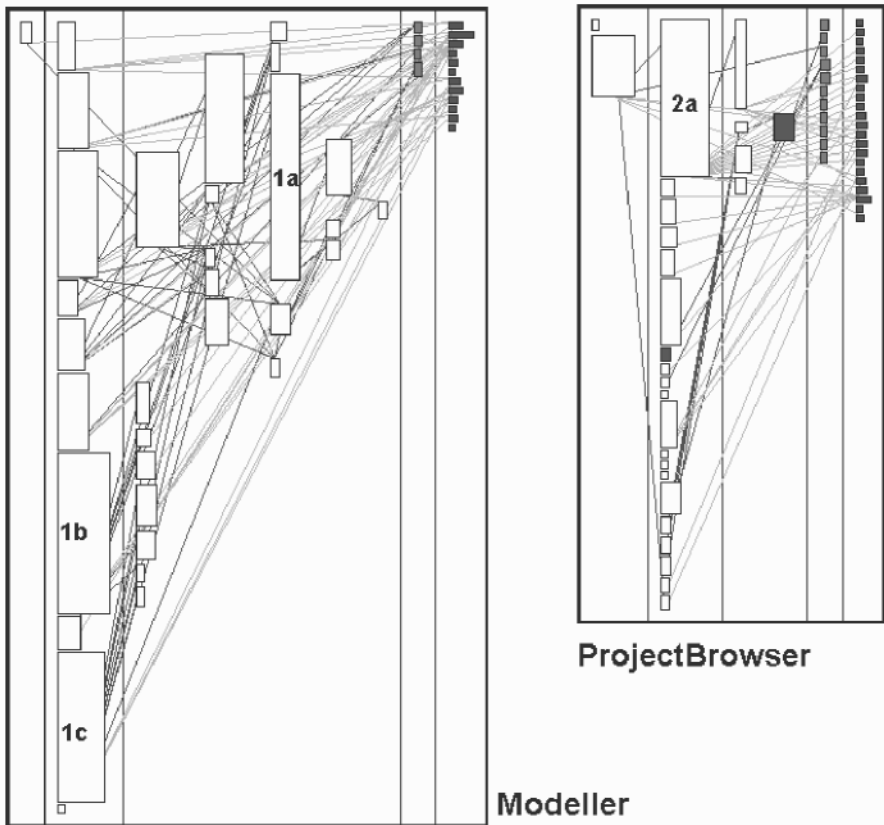


Fig. 5.10. A *Class Blueprint* of Modeller and ProjectBrowser.

Example

Fig. 5.10 shows that Modeller is not a class with an excessive number of methods, but has a certain number of Brain Methods. Some of the methods reach considerable sizes (eight methods are longer than 50 lines of code), the longest one `addDocumentationTag` (annotated as 1a in the figure) is 150 lines of 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 four accessor methods. Moreover, the attributes are accessed both directly and indirectly (using the accessors), denoting a certain inconsistency or lack of access policy. As we will see

in the chapter on collaboration disharmonies, the class `Modeller` is also affected by other problems such as `Dispersed Coupling`(127) and `Intensive Coupling`(120).

The class `ProjectBrowser` has a very high ATFD (Access To Foreign Data) value, as it accesses the data of seven other classes (this cannot be seen in the blueprint, since we only display the class itself). As we look closer at the three disharmonious methods of this class we find out that this situation has two different reasons: the less “harmless” one is encountered in the `createPanels` method (annotated as 2a, 116 LOC) where various UI components are added to an UI panel. There is also a more harmful case, i.e., a violation of Demeter’s Law[Lie96] where the programmers build long chains of method calls, most of which are accessor methods. A relevant example is the following code sequence found in the `setTitle` method:

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

ArgoDiagram activeDiagram =
    ProjectManager.getManager().
        getCurrentProject().
        getActiveDiagram();
```

The problem with such long invocation chains is that only one of the “links” in the middle has to break (because some method has changed) to make the whole chain break down.

Fowler suggests [FBB⁺99] that in almost all cases a Brain Method should be split, i.e., that one or more methods (operations) are to be extracted. He also explains how to find the possible “cutting points”:

Refactoring

[...] whenever we feel the need to comment something, we write a method instead. Such a method contains the code that was commented but is named after the intention of the code rather than how it does it.

In spite of this simple heuristic, refactoring a Brain Method can be a complex task, which needs a global perspective to solve it. Often we find Brain Methods among the suspects of the `Intensive Coupling`(120) and `Dispersed Coupling`(127) design flaws. To properly refactor a Brain

Method we need a complex (interdependent) three-fold analysis, involving all harmony aspects:

1. **Identity harmony.** Implies the already-mentioned aspect of its length which points to a split method refactoring. It may also involve Duplication(102) that implies the extraction of the common part to a method of that class. Additionally, a Brain Method (or a part of it) may exhibit Feature Envy(84). In this case, the refactoring would mean extracting a part of the method or – in some rare cases – moving the method completely to the “data provider”.
2. **Collaboration harmony.** As mentioned before, it is often the case that Brain Methods exhibit also Intensive Coupling(120) classification disharmony. This could imply the following refactoring: replace a “cluster” of calls to lightweight methods and the afferent logic with fewer calls to higher-level (more complex) services (see also explanations on Intensive Coupling(120)). This implies extracting a part of the method and moving it to another class.
3. **Classification harmony.** This aspect of harmony might be involved as well if Duplication(102) is detected in the Brain Method. If this is the case, often among the other methods that are the “duplication partners” we find other Brain Methods. Thus, the method can be restructured by factoring out the commonalities in the hierarchy (e.g., apply the Template Method [GHJV95] design pattern).

5.7 Brain Class

This design disharmony is about complex classes that tend to accumulate an excessive amount of intelligence, usually in the form of several methods affected by Brain Method(92).

Description

This recalls the God Class(80) disharmony, because those classes also have the tendency to centralize the intelligence of the system. It looks like the two disharmonies are quite similar. This is partially true, because both refer to complex classes. Yet the two problems are distinct.

The fingerprint of a God Class is not just its complexity, but the fact that the class relies for part of its behavior on encapsulation breaking, as it directly accesses many attributes from other classes.

On the other hand, the Brain Class *detection strategy* is trying to complement the God Class strategy by catching those excessively complex classes that are not detected as God Classes either because they do not abusively access data of “satellite” classes, or because they are a little more cohesive.

Classes which are not a God Class(80) and contain at least one method affected by Brain Method(92).

Applies To

See impact of the God Class(80) disharmony.

Impact

The detection rule can be assumed as follows (see Fig. 5.11). A class is a Brain Class if it has at least a *few* methods affected by Brain Method(92), if it is very large (in terms of LOC), non-cohesive and very complex. If the class is a “monster” in terms of both size (LOC) and functional complexity (WMC) then the class is considered to be a Brain Class even if it has only one Brain Method(92).⁴ The *detection strategy* in detail is:

Detection

1. **Class contains more than one Brain Method(92) and is very large.** A class is very large if the total number of lines of code from methods of the class is *very high* (see Fig. 5.12).

⁴ Looking carefully at the detection rule and comparing it to the one for God Class(80), you will notice that nothing hinders a God Class from also being detected as a Brain Class. For simplification, we exclude a priori classes classified as God Classes.

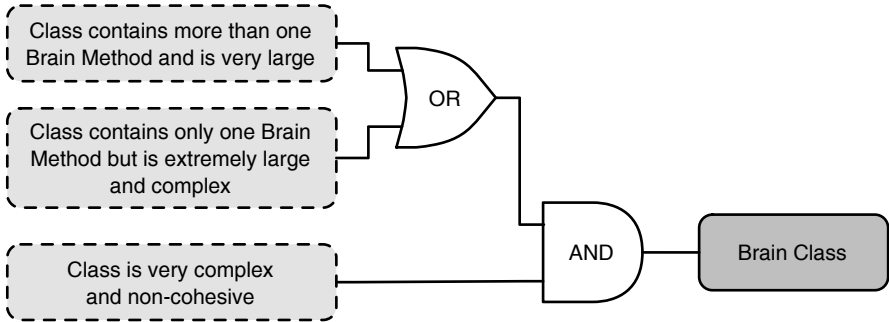


Fig. 5.11. Detection strategy for Brain Class.

2. **Class contains only one Brain Method(92) but is extremely large and complex.** This term covers the above case of a “monster” class in both size and complexity, and which is not captured by the previous term due to the fact that the class has only one *Brain Method*. In other words, this is the case of a class where most methods tend to be excessively large and complex, even if they are not *Brain Methods*. Compared to the previous term, a special condition (WMC) was added on the complexity of the class. This condition overrides the “normal” filtering condition for WMC, as defined in the third term.
3. **Class is very complex and non-cohesive.** This last term sets a requirement on the increased complexity and low cohesion that characterize mainly all classes with identity disharmonies. This pair of filtering conditions is similar to the one found in the detection rule for God Class(80); in fact there is only one difference: the threshold for the cohesion metrics is more permissive than in the other *detection strategy*, as there the very low cohesion is a more significant characteristic than here.

Example

In Fig. 5.13 we see that the class `ParserDisplay` not only is visually deformed, but also plays strange tricks in terms of inheritance. As for the visual deformation, this class implements some very large methods, the largest one (the tallest method box) with 576 lines of code (this method is also the largest in the entire system) and another five methods longer than 100 lines. In total 13 methods are longer than 50 lines. Moreover, there is a large amount of intra-method dupli-

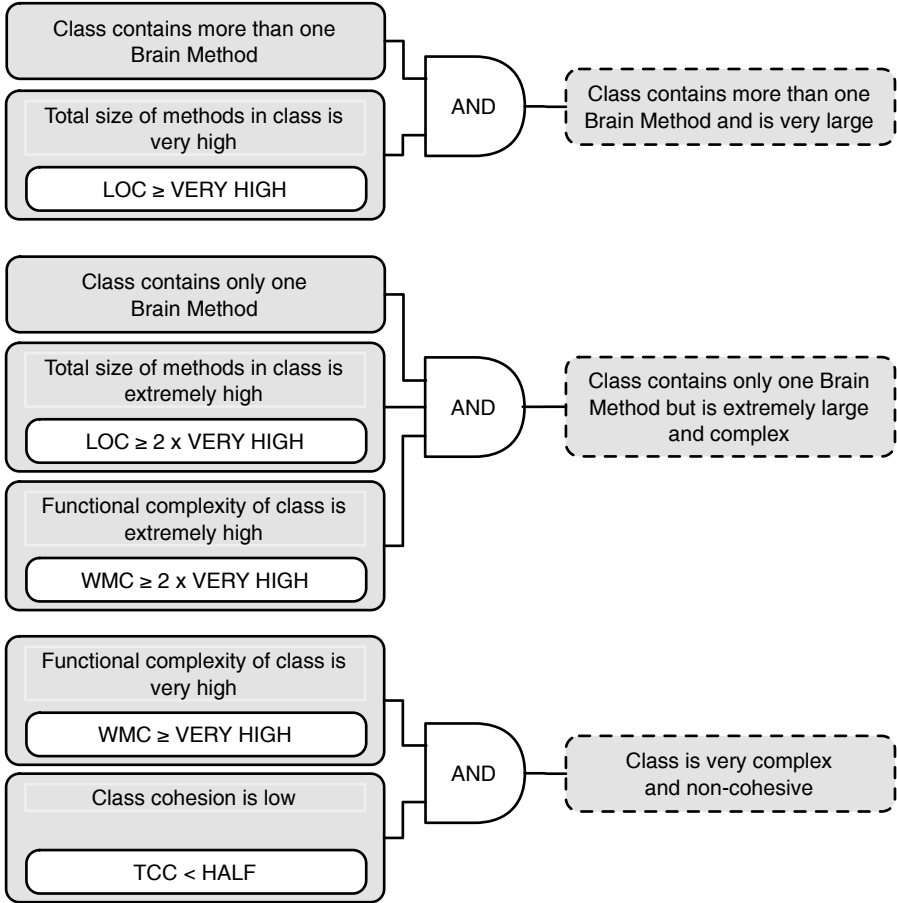


Fig. 5.12. Main components of the Brain Class *detection strategy*.

cation: for example, in the constructor of this class, which contains seven code blocks containing duplication, there are 89 lines of code in total, whereas the constructor has 132 lines in total. Another particular aspect of this class is its inheritance relationship with its superclass, whose discussion we postpone to the section on classification harmony.

The class FigClass is severely affected by the Brain Method(92) disharmony. This class has another problem: code duplication. Eleven of its methods are affected by Duplication(102), nine (!) of which are involved in duplication with three of the sibling classes, especially

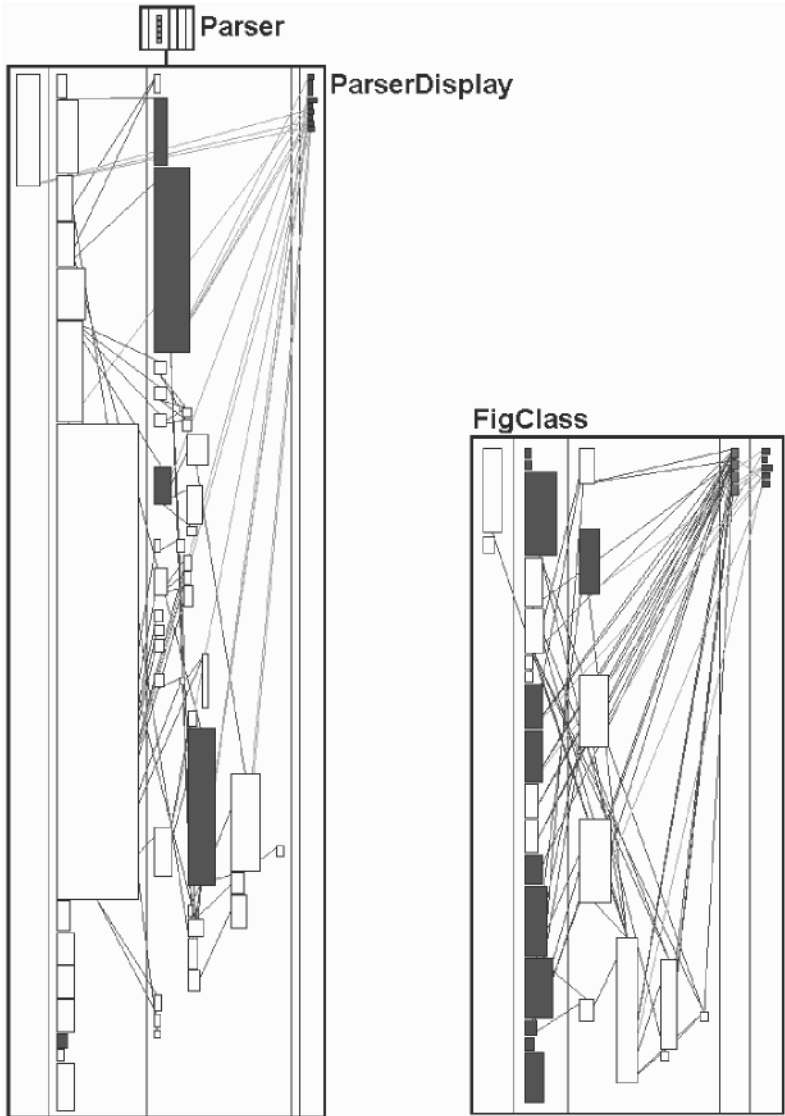


Fig. 5.13. A *Class Blueprint* of `ParserDisplay` with its completely abstract superclass `Parser` and a *Class Blueprint* of `FigClass`.

with `FigInterface` and `FigUseCase`. What is even more interesting is that among the nine methods with `Duplication(102)` we found all three Brain Methods of the class, all of them with significant amounts of duplication. Our conclusion is that if the duplication “plague” were

removed this class would become much lighter, and less problematic. As you can see, in order to restore one aspect of harmony the other aspects must be considered as well. In this concrete example, we would not have found the cause of the Brain Method(92) problem if we had not looked at the duplication within the hierarchy.

The primary characteristic of a Brain Class is the fact that it contains Brain Method. Therefore the main refactoring actions for these classes must be directed towards curing the Brain Method(92) disharmonies. Additionally, in our approach classes affected either by Brain Class(97) or God Class(80) represent the starting point in the detection and correction of identity disharmonies (see Sect. 5.9).

Refactoring

Apart from that, in our experience, there are at least three types of Brain Class, each of them requiring a different treatment:

1. The methods suffering from Brain Method(92) contained in the class are semantically related (oftentimes overloaded methods), and contain a significant amount of duplicated code. Factoring out the commonalities from these methods in form of one or more private or protected methods, while making the initial methods provide only the slight differences would significantly reduce the complexity of the class.
2. A possible type of Brain Method appears when a class is conceived in a procedural programming style. Consequently, the class is mainly used as a grouping mechanism for a collection of somehow related methods that provide some useful algorithms. In this case the class is non-cohesive. Refactoring such a class requires to split it into two or more cohesive classes. Yet, performing such a refactoring requires a substantial amount contextual information (e.g., which class(es) use(s) which parts of the initial class, where is stored the data on which each Brain Method operates on etc.)
3. There are cases where a Brain Class proves to be rather harmless. In several case studies we encountered cases where an excessively complex class was a matured utility class, usually not very much related to the business domain of the application (e.g., a class modelling a Lisp interpreter in a 3-D graphics framework). If, additionally, the maintainers of the system or the analysis of the system's history [RDGM04] show that no maintenance problems have been raised by that class then it makes no sense to start a costly effort of refactoring that class just for the sake of getting better metric values for the system.

5.8 Significant Duplication

Description

The detection of code duplication plays an essential role in the assessment and improvement of a design. But detected clones might not be relevant if they are too small or if they are analyzed in isolation. In this context, the goal of this *detection strategy* is to capture those portions of code that contain a *significant* amount of duplication. What does significant mean? In our view a case of duplication is considered significant if:

- It is the largest possible chain of duplication that can be formed in that portion of code, by uniting all islands of *exact clones* that are close enough to each other.
- It is large enough.

Applies To

Pairs of operations.

Impact

Code duplication harms the uniqueness of entities within a system. For example, a class that offers a certain functionality should be solely responsible for that functionality. If duplication appears, it becomes much harder to locate errors because the assumption “only class X implements this, therefore the error can be found there” does not hold anymore. Thus, the presence of code duplication has (at least) a double negative impact on the quality of a system: (1) the bloating of the system and (2) the co-evolution of clones (the clones do not all evolve the same way) which also implies the cloning of errors.

Detection

In practice, duplications are rarely the result of pure copy-paste actions, but rather of copy-paste-adapt “mutations”. These slight modifications tend to scatter a monolithic copied block into small fragments of duplicated code. The smaller such a fragment is, the lower the refactoring potential, since the analysis becomes harder, and the granted importance is decreased, too. So, for example, imagine we found two operations that have five identical lines, followed by one line that is different, which is followed by another four identical lines. Did we find two clones (of five and four lines) or one single clone

spread over ten lines (5 + 1 + 4 lines)? In such cases, it is almost always better to choose the second option.

Thus, there are two cases of duplication: the *copy-paste* case and the *copy-paste-adapt* case. This *detection strategy* captures both cases. The first term deals with the case of a brute-force duplication which is significantly large. The second term tackles the case of duplication with slight adaptations, assuming that the largest possible chain of duplication is considered. In both case the key element is the size of the duplication.

In order to introduce the *Significant Duplication detection strategy* (see Fig. 5.14), we need first to present three low-level duplication metrics:

- **Size of Exact Clone (SEC).** An *exact clone* is a group of consecutive line-pairs that are detected as duplicated. Consequently, the *Size of Exact Clone* metric measures the size of a clone in terms of lines of code. The size of a clone is relevant, because in most of the cases our interest in a piece of duplicated code is proportional to its size.
- **Line Bias (LB).** When comparing two pieces of code (e.g., two files or two functions) we usually find more than one *exact clone*. In this context, *Line Bias* is the distance between two consecutive *exact clones*, i.e., the number of non-matching lines of code between two *exact clones*. The LB value may allow us to decide if two *exact clones* belong to the same cluster of duplicated lines (e.g., the gap between the two *exact clones* could be a modified portion of code within a duplicated block of code).
- **Size of Duplication Chain (SDC).** To improve the code we need to see more than just a pile of small duplication chunks. We want to see the big picture, i.e., to cluster the chunks of duplication into a more meaningful block of duplication. This is what we call a *duplication chain*. Thus, a *duplication chain* is composed of a number of smaller islands of *exact clones* that are close enough pairwise to be considered as belonging together, i.e., their LB value is less than a given threshold.

Now, with these metrics in mind we can revisit the example mentioned earlier in this section, with two functions having two *exact clones*. In terms of the low-level duplication metrics introduced in this section, we can now say that the first clone has a SEC value of 5, while the second one has a SEC value of 4. Between the two clones

there is a gap of one line; thus, the LB value is 1. Consequently the SDC metric has a value of 10 lines (5 + 1 + 4 lines).

Based on these low-level metrics we can now introduce the heuristics for this *detection strategy*:

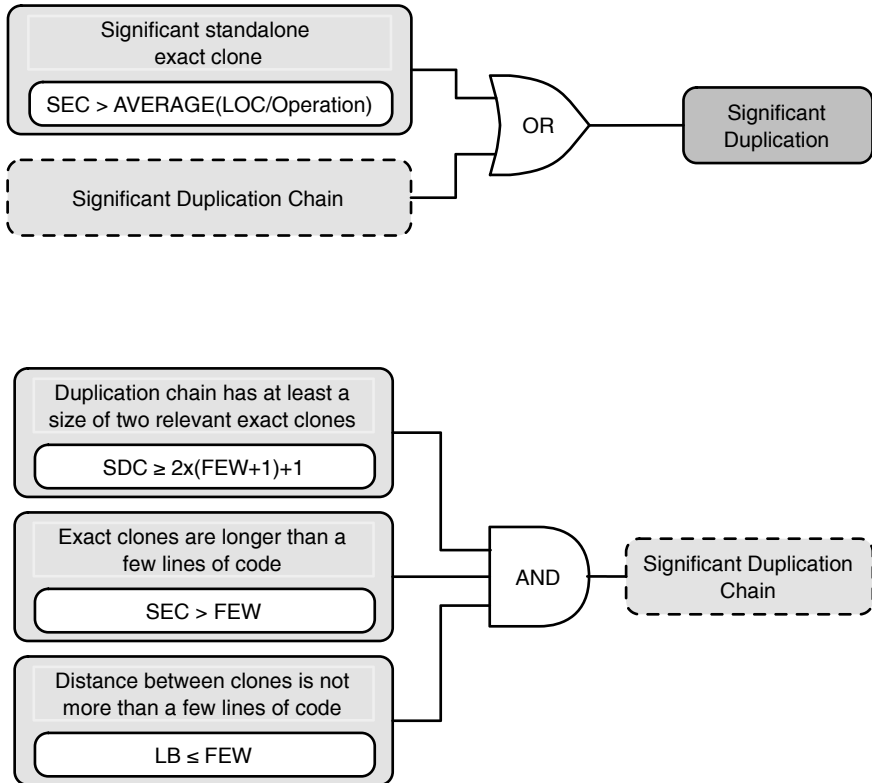


Fig. 5.14. The Significant Duplication detection strategy.

1. **Significant Standalone Exact Clone.** This case captures the case of a contiguous, isolated block of duplication, i.e., a single *exact clone* that has no other clones in its neighborhood. Thus, the only thing that counts is the size of the *exact clone*. We consider a standalone clone to be large enough if it is at least as large as the statistical average size of an operation.
2. **Significant Duplication Chain.** We stated earlier that a block of duplicated code is significant only if it is the largest one that could

be built in a particular area of the two pieces of code that are compared. In other words, we try to build the largest chain of relevant *exact clones* that are not too far from each other. As you might notice, the previous phrase is fuzzy if not associated with a measurement. Based on the low-level metrics defined earlier, we eliminate the “fuzziness” and make the identification of these clusters reproducible. This term is composed of three metrics (see Fig. 5.14), each one carrying out a particular role:

- a) **Duplication chain has at least a size of two relevant exact clones.** This threshold is an indirect one, meaning that the duplication chain has at least the total size of two significant exact clones separated by a gap of minimal distance. The term $2 \times (FEW + 1)$ is based on the condition that each of the (minimum) two fragments involves *more than a few duplicated lines*. Because the minimal distance (i.e., the smallest LB value) is *one* line of code, we add to the first threshold term one more line. It ensures that the total length of the duplication chain is large enough to qualify it as significant.
- b) **Exact clones are longer than a few lines of code.** This makes sure that the chain is not composed only of irrelevant “duplication crumbs”, i.e., that each fragment of the duplication chain is not very small.
- c) **Distance between clones is not more than a few lines of code.** This quantifies the “neighborhood” aspect as it ensures that the pieces of the chains are not too far from each other to be considered as belonging to the same duplication chain. In other words, the threshold for the LB metric is used as a stop condition in the process of looking for further neighbor clones.

Looking at the *ArgoUML* case study just shows that code duplication is one of the plagues that are omnipresent; but this can be now quantified. In the case of *ArgoUML*, we checked for *Significant Duplication* and we found that 239 classes (17% of all the classes) are affected by it. Summing the SDC duplication metric at the system level, we end up with more than 10,000 duplication lines!⁵

Example

Usually duplication is a design disharmony that often appears in conjunction with other disharmonies. Therefore, we believe that it

⁵ Note that one code line may be involved in more than one duplication chain, and thus it is multiply counted; still, the number of lines of code involved in duplication is impressive.

does not make sense to discuss just a single concrete example of duplication. So, the aspect of duplication will occur over and over again, as we discuss in an integrated manner various design problems that we encountered in *ArgoUML*.

Refactoring

The essence of a refactoring that intends to eliminate duplication is based on Beck's *Once and Only Once Rule* [Bec97]:

By eliminating the duplicates, you ensure that the code says everything once and only once, which is the essence of good design.

Thus, it is clear that we have to put all “instances” of a duplicated portion of code into one single location. But what is the proper location? To be able to answer this question we obviously need more information about the *context* of the duplicated entities.

The first problem is that by detecting only exact clones we usually end up with lots of small clones (duplication crumbs) which are irrelevant in themselves; yet, ignoring them would be a mistake as they could be in fact part of a large duplication chain, as a result of an extensive *copy-paste-adapt* process. The *Significant Duplication* strategy helps us in solving this first headache and keeps only significant portions of code affected by duplication.

This brings us to the second headache: How to refactor the code so that duplication is removed? Are all significant duplication blocks the same? Can we apply the same treatment to all? Especially when speaking about object-oriented design, the answer to these questions is definitely (and obviously): No! This is why we identify three different contexts in which duplication appears.⁶

Case I: Duplication Within the Same Class

In this case the two methods involved in a (significant) duplication block belong to the same class. This is probably the easiest refactoring: all that needs to be done is to extract the commonality in the form of a new method and call the new method from both places (see Fig. 5.15).

Case II: Duplication Within the Same Hierarchy

In this second case the two methods that are part of a (significant) duplication block are not part of the same class, but belong to the same

⁶ Note that we speak here exclusively about duplication of functional code, i.e., duplication that appears in the bodies of functions and methods.

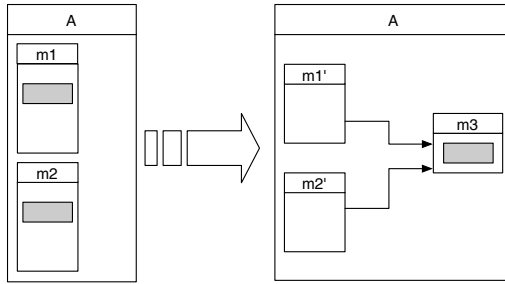


Fig. 5.15. Recovering from Duplication within the same class.

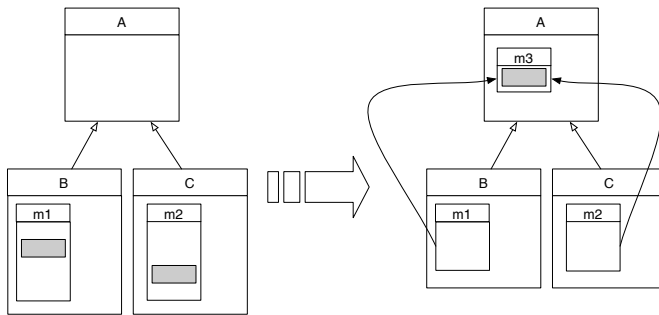


Fig. 5.16. Recovering from Duplication within sibling classes.

class hierarchy, which means that either they are in an ancestor–descendant relation or they share a common base class. This type of duplication can be eliminated in one of the following ways, depending on the inheritance relation between the methods involved in the duplication (see Fig. 5.16):

- *Siblings duplication.* If duplication appears in two methods that have a common ancestor, then the commonality is extracted in the form of a new method placed into the common ancestor.
- *Parent–child duplication.* This case of duplication is a strange one, because the two classes are in a direct relation. Thus, any commonality could have been placed in the base class. The refactoring consists of extracting any common code and placing it in the parent class, where it logically belongs.

A special case is the one where the duplication between two inheritance-related methods is fragmented, i.e., the code is similar but not

identical. In this case you would probably be able to apply the *Template Method* design pattern [GHJV95], as this would help separate the common code (which goes into the closest ancestor class) from the fragments that are different (which will become the hooks from the pattern mentioned above).

Case III: Duplication Within Unrelated Classes

In this third case the two operations that share a duplicated block are neither part of the same class, nor of the same hierarchy; either the two operations are part of two independent classes (in the sense of classification) or they are (one of them or both) global functions.

If you find duplicated code in methods belonging to unrelated classes, there are three major options on where to place the common code, extracted from the two (or more) classes:

- *One hosts, one calls.* In this case we notice that the code belongs to one of the protagonist classes. Thus, it will host the common code, in the form a method, while the other class will invoke that method. This usually applies when the portions of duplication are not very large and especially not encountered in many methods. If the duplication between two classes affects many methods, then we probably miss an abstraction, i.e., a third class. Thus, we define the new class and place the duplicated code there. Now, the question is how to relate the two former classes with this third one? The answer depends on the context, boiling down to two options: *association* and *inheritance*.
- *Third class hosts, both inherit.* If we find that the two classes are conceptually related, then they probably miss a common base class. Consequently the third class becomes the base class of the two.
Good examples for this case are the classes `FigNodeModelElement` and `FigEdgeModelElement` which indeed miss a common base class.
- *Third class hosts, both call.* If the two unrelated classes involved in duplication are not conceptually related we need to introduce an association from the two classes to the third one and call from both classes the method that now hosts the formerly duplicated code.

5.9 Recovering from Identity Disharmonies

Where to Start

In practice we do not have enough time to analyze each suspect class or method reported by the *detection strategies*. Therefore, a pragmatic question pops up: How do we find the most important identity harmony offenders? We used the following criteria in selecting the classes that especially need attention with respect to identity harmony:

- Classes that contain a higher number of disharmonious methods have priority.
- Classes in which more than one identity disharmony appears have priority.
- Classes that are affected by other disharmonies (i.e., collaboration or classification disharmonies) go first in order to reveal relations to other aspects of harmony.

This can be done in two steps (see Fig. 5.17):

1. Start with the “intelligence magnets”, i.e., with those classes that tend to accumulate much more functionality than an abstraction should normally have. In terms of the *detection strategies* presented so far, this means to make a blacklist containing all classes affected by the God Class(80) or by the Brain Class(97) disharmony.
2. For each of the classes in the blacklist built in *Step 1* find the *disharmonious methods*. A method is considered disharmonious if at least one of the following is true:
 - it is a Brain Method(92);
 - it contains duplicated code;
 - it accesses attributes from other classes, either directly or by means of accessor methods.

We mainly use the following quantification means:

Count disharmonious methods. To both assess and cure such a disharmonious class we need first to examine how much the identity problems have spread among the methods of that class. Therefore, we have to count how many *methods* we can identify as having identity problems. The more disharmonious methods a class has, the more critical its identity is. When do we consider a method as being disharmonious? We do so if at least one of the following conditions holds:

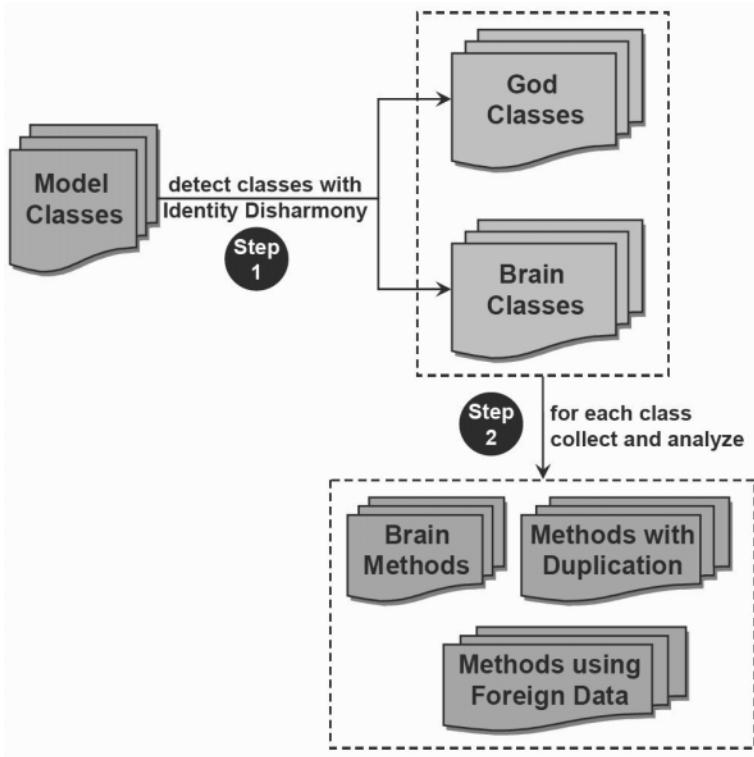


Fig. 5.17. Overview of the assessment process related to Identity Harmony.

1. It contains duplicated code in common with methods of the same class.
2. It is a Brain Method(92).
3. It accesses directly attributes of other classes.

Number of Methods (NOM). This metric gives us information about the functional size of the class. If we correlate the number of disharmonious methods with NOM, we can also determine what percentage of the class is affected by these identity problems.

Number of methods detected as Brain Method(92). We use this number to see how many of the disharmonious methods are detected as being a Brain Method(92).

Duplicated LOC. This metric tells us for each class the amount of intra-class duplication, which refers to source code duplicated within a class, i.e., among the methods defined in the same class.

This information helps us also to get a better understanding of what the problem is with the disharmonious methods.

Access To Foreign Data (ATFD). This metric is included because it quantifies one of the key disharmonies of an identity distortion, i.e., the brute usage of attributes from other classes. As you may notice, this is again one of the reasons that qualify a method as being *disharmonious*.

How to Start

How should you start when you want to improve the identity harmony of your system's classes? Assuming that for a class in the blacklist we have gathered its disharmonious methods, then in order to recover from identity design disharmonies we have to follow the roadmap described in Fig. 5.18, and explained briefly below.

- *Action 1: Remove duplication.* The first thing to be done is to check if a method contains portions of Duplication(102) and remove that duplication in conformity with the indications provided in Sect. 5.8. Because we analyze the class from the perspective of *identity harmony* we concentrate on removing the intra-class duplication first. If a lot of duplication is found, the result of this step can have a significant positive impact on the class, especially on its Brain Methods.
- *Action 2: Remove temporary fields.* Among the bad smells in code described in [FBB⁺99] we find one called *Temporary Field*. This is an attribute of a class used only in a single method; in such cases the attribute should have been defined as a local variable. Obviously, detecting such situations can be done by checking in the class who other than the inspected method uses a particular attribute. If no one else does, then we need to remove the temporary field and replace it with a local variable. Why do we care? Remember that for both Brain Class(97) and God Class(80) one of the “fingerprints” is a low cohesion. One of the causes of low cohesion could also be a bunch of such *temporary fields*, which do not really characterize the abstraction modelled by the class, and thus hamper the understanding of the class.
- *Action 3: Improve data-behavior locality.* If in our inspection process we reach a *foreign data user*, i.e., a method that accesses

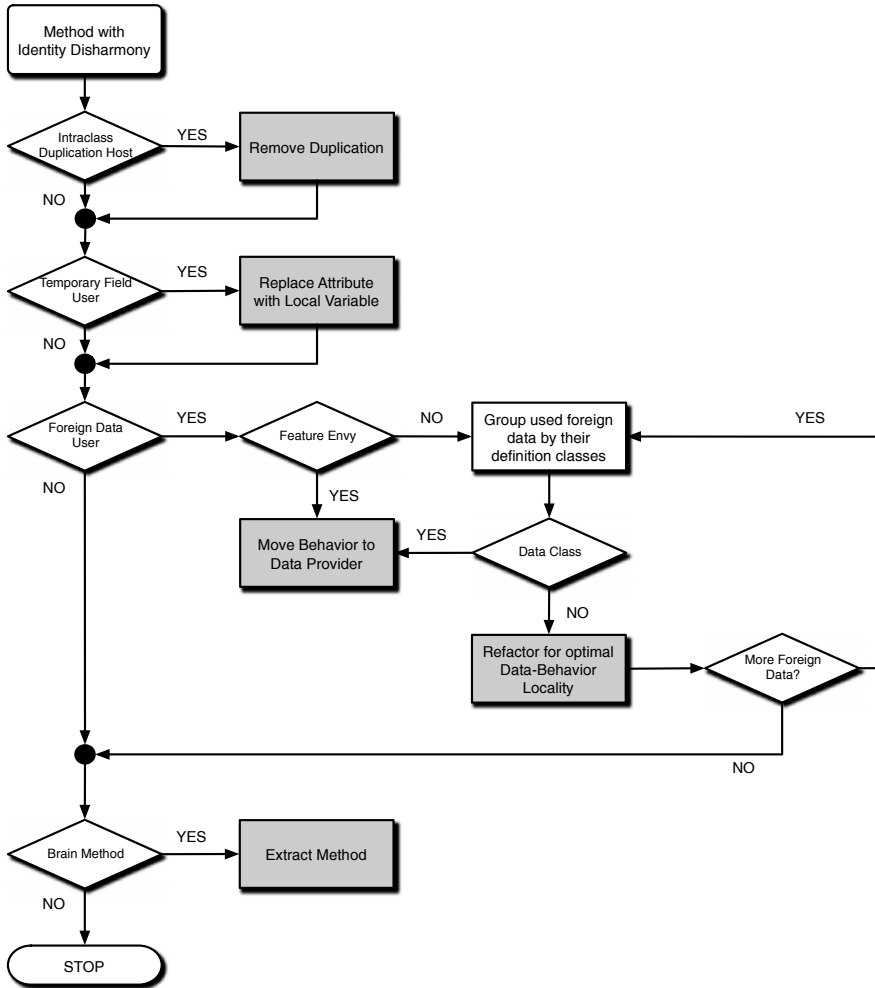


Fig. 5.18. How to address Identity Disharmonies in methods

attributes of other classes, then we have to refactor it so that we reach an optimal *data-behavior locality*. A *foreign data user* has one characteristic: the value for the ATFD metric is at least one. In a simplistic way we can say that refactoring in this case requires one of these two actions:

- *Extract a part of the method and move it to the definition class of the accessed attribute.* The “ideal” case is when the method is affected by Feature Envy(84) and the class that provides the at-

tributes is a Data Class(88). In this case the method was simply misplaced, and needs to be moved to the Data Class(88). But life is rarely that easy, so the situations that you will probably encounter are more “gray” than “black and white”. In most cases only a fragment of the method needs to be extracted and moved to another place. This entire discussion is beyond the scope of this book, but here is a rule of thumb that we often use: if the class that provides the accessed attributes is “lightweight” (i.e., Data Class(88) or close to it) try to extract fragments of functionality from the “heavyweight” class and move them to the “lightweight” one.

- *Move the attribute from its definition class to the class where the user method belongs.* This is very rarely the case, especially in the context of Brain Class(97) and God Class(80). It applies only for cases where the attribute belongs to a small class that has no serious reason to live, and which will be eventually removed from the system.
- *Action 4: Refactor Brain Method.* If you reached this step while inspecting a method that was initially reported as a Brain Method(92), first look if this is still the case after proceeding with *Step 1* and *Step 3*. Sometimes, removing duplication and refactoring a method for better data-behavior locality solves the case of the Brain Method(92). If the problem is not solved, revisit Sect. 5.6 where we discussed the main refactoring cases for a Brain Method(92).