7

Classification Disharmonies

7.1 Classification Harmony Rules

The object-oriented programming paradigm captures the *is-a-kind-of* relationship among classes with inheritance. This allows developers to write flexible and reusable code, but it can lead to disastrous designs if misused.

It is not enough for a class to be in harmony with itself; it also needs to be in harmony with its its ancestor and its descendant classes. The major cause of classification disharmonies is the misconception that inheritance is mainly a vehicle of code reuse (i.e., subclassing) rather than a means to assure that more specific objects can substitute more general ones (i.e., subtyping) [LP90, LW93a, Mar02b]. When inheritance is used solely for code reuse purposes maintenance can become painful because abstractions are not derived consistently.

The classification harmony rules are:

Classes should be organized in hierarchies having harmonious shapes

The identity of an abstraction should be harmonious with respect to its ancestors

Harmonious collaborations within a hierarchy are directed only towards ancestors, and serve mainly the refinement of the inherited identity

Proportion Rule

> *Classes should be organized in hierarchies having harmonious shapes*

Rationale

Inheritance is at the same time a curse and a blessing of the object-oriented paradigm. The one extreme is given by applications where basically inheritance is ignored and the application is a flat collection of classes, leading to limited code reuse. The other extreme is overuse of inheritance, where the code is so heavily decomposed in a hierarchy that reading the code is equivalent to browsing excessively up and down the classes and methods in the hierarchy. Inheritance should be used with care and style.

Practical Consequences

- **Avoid wide hierarchies** – *Class hierarchies should not become too* wide, *i.e., avoid inflation of subclasses.*

  Excessively wide hierarchies oftentimes appear because of copy-paste-and-adapt patterns: the developers prefer to copy and modify existing code instead of refactoring the existing code. Since developers do not get to *see* such things they underestimate the effect of copy-paste practices.

- **Avoid tall hierarchies** – *Class hierarchies should not become too* tall. *Avoid very narrow and deep hierarchies.*

Presentation Rule

> *The identity of an abstraction should be harmonious with respect to its ancestors*

Rationale

The concept of inheritance allows for writing compact code and also for the reuse of the code already implemented in one of its ancestor classes. In this sense a descendant should always be *in sync* with what has been defined by its ancestors, and not reinvent the wheel or duplicate the code.

Practical Consequences

- **Extend interface smoothly** – *Keep an harmonious proportion between* tradition *and* novelty. *In other words, keep a balance between the inherited interface and its extension (through addition of new services).*

- **Specialize behavior smoothly** – *Keep an harmonious proportion between* evolution *and* revolution *of behavior. In other words, do not refuse (deny, "cut off") any parts of an ancestor's interface and specialize rather than override the inherited services (i.e., the inherited public methods).*

- **Decrease abstractness smoothly** – *The abstractness level for the set of services of a class (together with their "inheritable" helper methods i.e., the protected ones) should be inversely proportional to the distance to the top of the hierarchy. Thus, root classes should be rather abstract or the other way around: abstract classes should be situated close to the top of a hierarchy and not somewhere in the middle of a hierarchy.*

Implementation Rule

> *Harmonious collaborations within a hierarchy are directed only towards ancestors, and serve mainly the refinement of the inherited identity*

Rationale

This rule could be rephrased as: *The implementation dependency of a class on its ancestor should be unidirectional and serve the refinement of the inherited services.* The rule states that a subclass should not depend on its ancestors just for the sake of code reuse (i.e., by calling methods from its base classes (only) from newly defined methods).

Practical Consequences

- **Dependencies go bottom-up** – *Base classes should not depend on their descendants.*

  Despite the truth behind this, there is a hidden world of horrors where developers who do not have a "complete picture" of the system just reuse pieces of code whenever they see something useful to them.

- **Dependencies serve specialization** – *Inherited operations should be used (i.e., redefined, called, specialized) most of the time in the context of refining (specializing) the inherited services, rather than calling them from newly added services.*

7.2 Overview of Classification Disharmonies

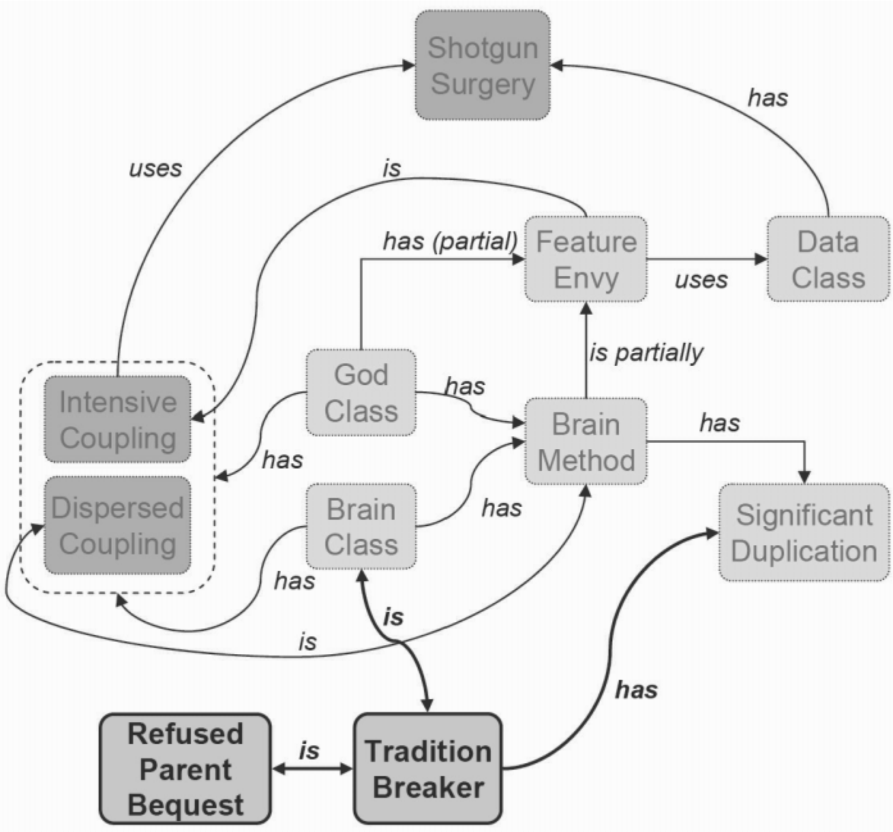

Fig. 7.1. Correlation web of classification disharmonies.

Considering the three harmony rules presented above, and based on our experience with analyzing object-oriented systems, we defined a set of patterns that capture the most disturbing classification disharmonies.

We have to mention Duplication(102) again, this time between inheritance-related classes. So, this is the first disharmony we are going to consider. This is often a symptom that goes together with other disharmonies. But even if there is no duplication in a hierarchy, it still needs to be harmonious with respect to its ancestors, as stated by the Presentation Rule (see Sect. 7.1). Distortions of this harmonious relation to the parent class(es) 7.1 appear as:

1. The derived class denies the inherited *bequest* [FBB+99] (Refused Parent Bequest(145)).
2. The derived class massively extends the interface of the base class with services that do not really characterize that family of abstractions (Tradition Breaker(152))

The shape of the hierarchy itself says a lot about the classification harmony. As we will see, in most cases the Refused Parent Bequest(145) and Tradition Breaker(152) disharmonies appear in an over-bloated hierarchy with an inflation of classes.

In conclusion, while inheritance is (also) a powerful mechanism to reuse code, subtyping is the actual point because it supports a better understanding of a hierarchy than subclassing, since a subclass is a more specialized version of its ancestor and not an unrelated concept that is there because it can reuse some code.

Another difficult issue related to inheritance is when is it useful to introduce a new class in the system. Often developers are afraid of having many small classes and prefer to work instead with fewer but larger classes. Developers often believe that they will have less complexity to manage if they have to deal with fewer classes. It is better to have more classes conveying meaningful abstractions than having a single large one. However, having useless classes or classes without meaningful behavior is not good either because they pollute and complicate the abstraction space: The challenge is to find the right level of abstraction.

7.3 Refused Parent Bequest

Description

Inheritance is a mechanism dedicated to support incremental changes. Consequently, the relation between a parent class and its children is intended to be an intimate one, more special than the collaboration between two unrelated classes. This special collaboration is based on a category of members (methods and data) especially designed by the base class to be used by its descendants, i.e., the protected members. But if a child class refuses to use this special bequest prepared by its parent [FBB+99] then this is a sign that something is wrong within that classification relation.

Applies To

Classes. The Following conditions are assumed:

1. the inspected class has a superclass;
2. the superclass is neither a third-party class (e.g., library class), nor is it an interface.

Impact

The primary goal of inheritance is certainly code reuse. However, extending base classes without looking at what they have to offer introduces duplication and in general class interfaces that become incoherent and non-cohesive. An often overlooked part of the process when adding or extending subclasses is to study the superclasses and determine what can be reused, what must be added and finally what could be pushed into the superclasses to increase generality.

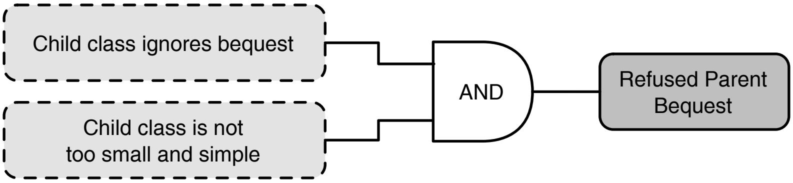
Child class ignores bequest
Child class is not too small and simple
AND
Refused Parent Bequest

Fig. 7.2. Detection strategy for Refused Parent Bequest.

Detection

As illustrated in Fig. 7.2 the detection of such disharmonious classes is based on two main conditions: (1) a low usage of inheritance-specific members from the base class and (2) the detected class must have at least an average size and complexity, otherwise the finding is irrelevant as the bequest refusal might be due to its small size. In other words, the second condition ensures that the bequest refusal is an intentional rather than a circumstantial fact. The *detection strategy* in detail is:

1. **Child class ignores bequest.** What do we mean by "a child class uses the parent's *bequest*"? We mean that it does one of the following:
   - it calls a protected method defined in the parent class
   - it accesses a protected attribute defined in the parent class
   - it overrides or specializes a method defined in the parent class

   To assess how much a child class depends on its parent class in an inheritance-specific way, we used two metrics: (1) The Base-class Usage Ratio (BUR), which quantifies the usage of protected members; and (2) the Base-class Overriding Ratio (BOvR), which quantifies the degree of overriding and specialization of base class methods. The third metric we use, Number of Protected Members (NPrM), just makes sure that there is a specific bequest to use, i.e., that there are at least several protected members. We use these metrics in the following way:

   a) **Parent provides more than a few protected members.** The bequest prepared by the parent class should be significant i.e., the base class has more than a few members declared as protected (in other words, members intended to be used specifically in the context of the inheritance relation).
   b) **Child uses only little of parent's bequest.**
   c) **Overriding methods are rare in child.** Overriding or specializing methods from the base class is a rare case in the derived class. Thus, the fraction of base class methods that are overridden or specialized is very low.
2. **Child class is not too small and simple.** We say about a child class that it intentionally refuses a bequest if it is large and complex enough; otherwise the child class can have the excuse of refusing the bequest because it is too small. Therefore, this term finds those classes that are both significantly large (in terms of methods (NOM)) and complex (Fig. 7.3).

   There are two alternative conditions for considering the complexity of the class significant: either the average CYCLO/method is high

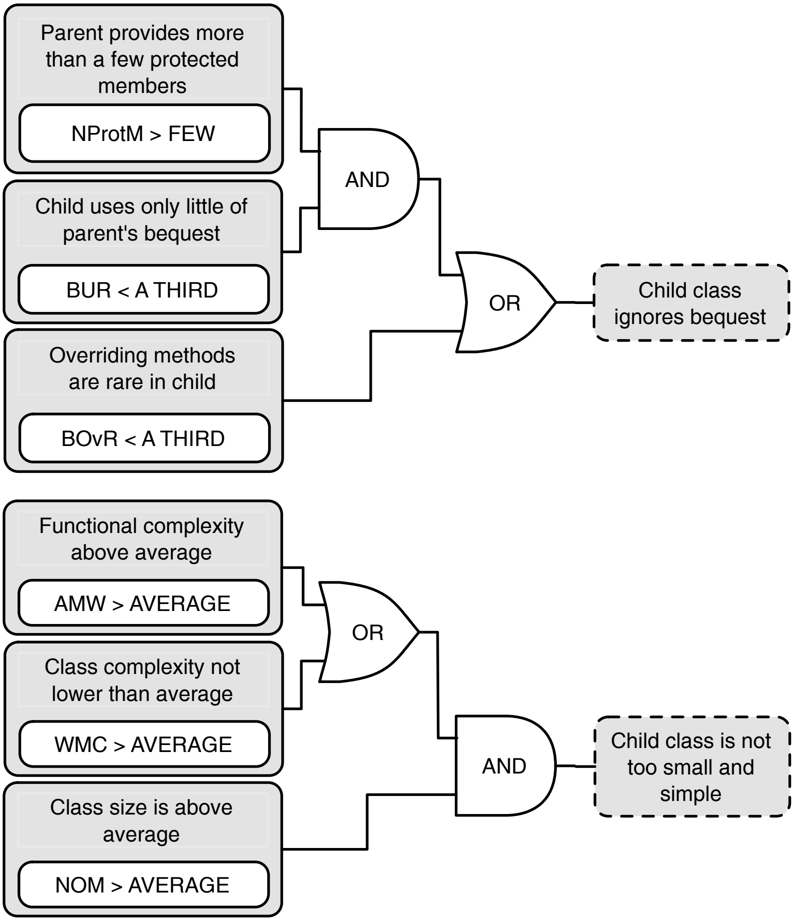
Parent provides more than a few protected members
NProtM > FEW
Child uses only little of parent's bequest
BUR < A THIRD
Overriding methods are rare in child
BOvR < A THIRD
AND
OR
Child class ignores bequest
Functional complexity above average
AMW > AVERAGE
Class complexity not lower than average
WMC > AVERAGE
Class size is above average
NOM > AVERAGE
OR
AND
Child class is not too small and simple

Fig. 7.3. Main components of the Refused Parent Bequest detection strategy.

enough, or the class is large and thus the cumulative complexity (WMC) makes it relevant. The used metrics are:

a) **Functional complexity above average.**
b) **Class complexity not lower than average.**
c) **Class is above average.**

Example

The unusual form of this hierarchy (see Fig. 7.4) already gives us a first hint that its classes are afflicted by some problems. Moreover, the fact that there is an abstract class (called ToDoPerspective) in the

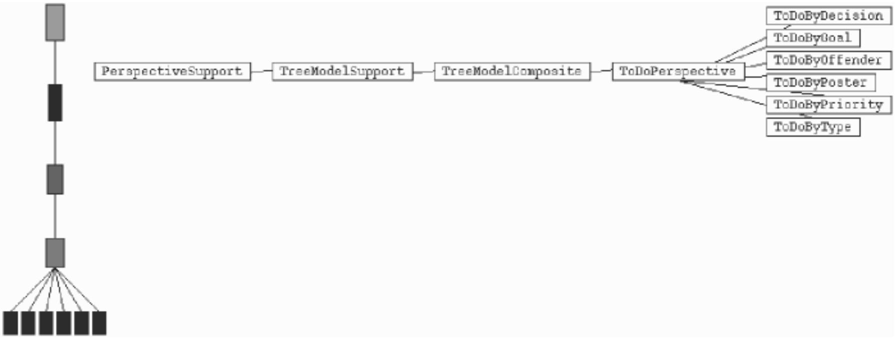

Fig. 7.4. A *System Complexity* view of the PerspectiveSupport hierarchy.

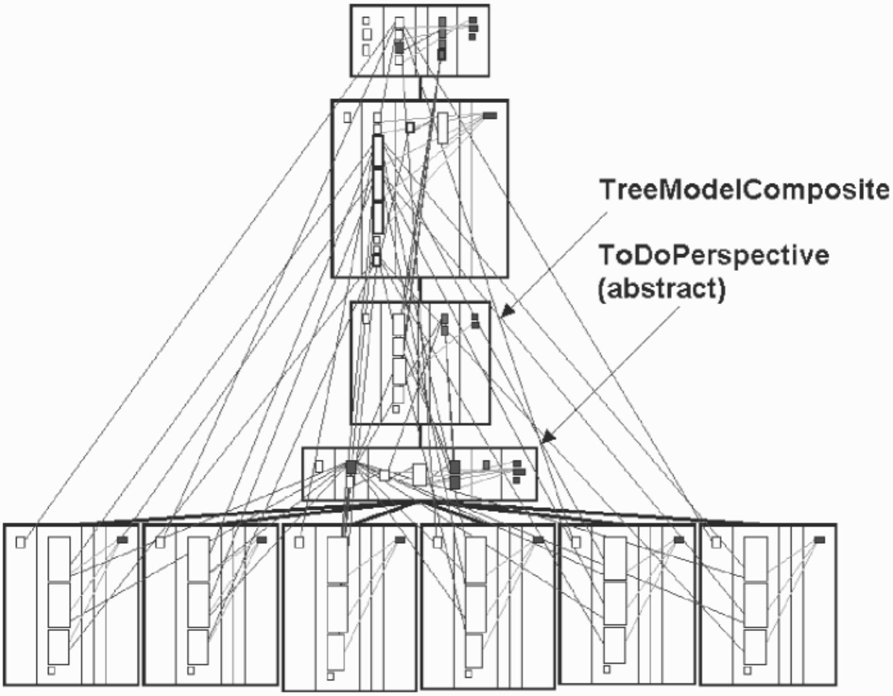

Fig. 7.5. A *Class Blueprint* view of the PerspectiveSupport hierarchy.

middle of the hierarchy also gives us hints about potential problems related to inheritance. The *Class Blueprint* of this hierarchy depicted in Fig. 7.5 shows a suspicious regularity in size among the methods implemented in the six leaf classes, hinting at duplication. The class TreeModelComposite is affected by Refused Parent Bequest: it basically ignores what is implemented in the two superclasses.

Refactoring

If we want to remove a Refused Parent Bequest disharmony from a class then we need to follow the detailed (inspection and refactoring) process depicted in Fig. 7.6. The figure has three areas (labeled A, B and C) corresponding to one of the three identified causes for a Refused Parent Bequest. Notice that some of the three causes might co-exist. Next, we are going to describe these three cases in detail.

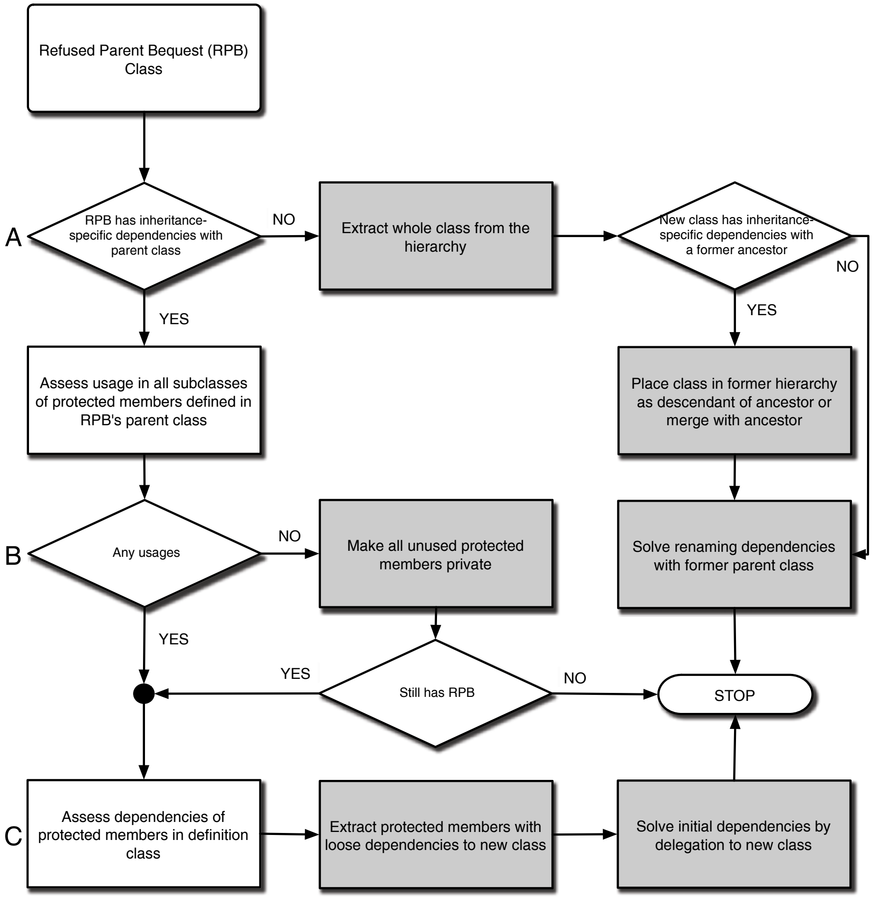
Refused Parent Bequest (RPB) Class
A
RPB has inheritance-specific dependencies with parent class
NO
Extract whole class from the hierarchy
New class has inheritance-specific dependencies with a former ancestor
NO
YES
YES
Assess usage in all subclasses of protected members defined in RPB's parent class
Place class in former hierarchy as descendant of ancestor or merge with ancestor
B
Any usages
NO
Make all unused protected members private
Solve renaming dependencies with former parent class
YES
YES
Still has RPB
NO
STOP
C
Assess dependencies of protected members in definition class
Extract protected members with loose dependencies to new class
Solve initial dependencies by delegation to new class

Fig. 7.6. Inspection and refactoring process for a Refused Parent Bequest.

Case A: False Child Class

In this case the cause of the problem is that the child class simply does not belong in the hierarchy; in other words, the hierarchy might be ill-designed. The more relevant symptom for this case is when the child class has *no inheritance-specific dependencies* on the parent class.

In some cases this goes together with the Tradition Breaker(152) disharmony. An interesting aspect is that in some cases the "false child" *does* belong to the hierarchy, but as a child class of another parent (i.e., an initial "grandparent" or ancestor). This can be found out by analyzing the dependencies between the disharmonious class and the other ancestors.

Case B: Irrelevant Bequest

In this case the Refused Parent Bequest design flaw appears as a result of the fact that the space of inheritance-specific members is over-populated with methods and attributes that have no relevance in the context of the inheritance relation.

But how do we detect that a (part of the) bequest is irrelevant? We have to count, for each protected member, the number of usages from derived classes; in case of protected methods, this includes overriding or specialization of that method in derived classes). If the number of dependencies is null, i.e., if a member is used only from inside the definition class, then it should be moved to a private scope.

Case C: Discriminatory Bequest

The third case, probably the most interesting one, is when the parent class has many child classes, and the bequest offered by it is relevant only for some of these siblings, but not for the class affected by Refused Parent Bequest. By cumulating the bequest needed by various subsets of descendants, the total bequest becomes excessively large. Consequently, the main symptoms in this case are:

- A large number of descendants.
- Often, there is more than one class exhibiting Refused Parent Bequest in the same hierarchy.
- Each descendant uses a small, non-overlapping portion of the total bequest.

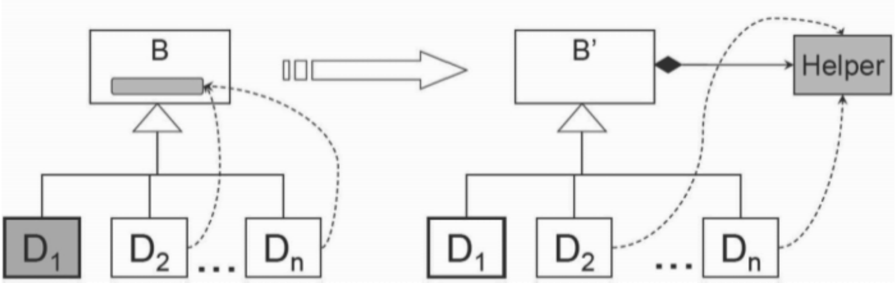

Fig. 7.7. Refactoring for Refused Parent Bequest in case of *Discriminatory Bequest*.

At first sight we could improve the design in this case by splitting class B in two classes, (B" derived from B') adding an intermediary layer in the inheritance tree and letting each initial subclass of B be derived either from B' or B" depending on which bequest they need to inherit. Unfortunately, this applies only for simple cases which involve few protected members used in common by only subsets of the derived classes.

For the general case, the situation can be improved by extracting the parts that are not used by all descendants to a helper class, and letting the parent class have a reference to an instance of the helper class (see Fig. 7.7). If this refactoring is applicable, then this could be also the sign that the base class was capturing more than a single abstraction. This way, the base class is easier to understand because it does contain less protected members which do not characterize the entire hierarchy.

7.4 Tradition Breaker

Description

This design disharmony strategy takes its name from the principle that the interface of a class (i.e., the services that it provides to the rest of the system) should increase in an evolutionary fashion. This means that a derived class should not break the inherited "tradition" and provide a large set of services which are unrelated to those provided by its base class.

Of course, it is OK for a child class to contain more intelligence than its parent i.e., to offer more services. But if the child class hardly specializes any inherited services and only adds brand new services which do not depend much on the inherited functionality, then this is a sign that something is wrong either with the definition of the child's class interface or with its classification relation. In the Suggested Refactoring (155) section we analyze in more detail the possible causes and solutions for this problem.

Applies To

Classes. If C is the name of the class, the following conditions are assumed: (1) C has a base class B, (2) B is not a third-party class and (3) B is not an interface.

Impact

When adding subclasses without examining the functionalities implemented in the superclass(es) one might break the tradition kept up by the superclasses. This could be called "disrespectful" inheritance.

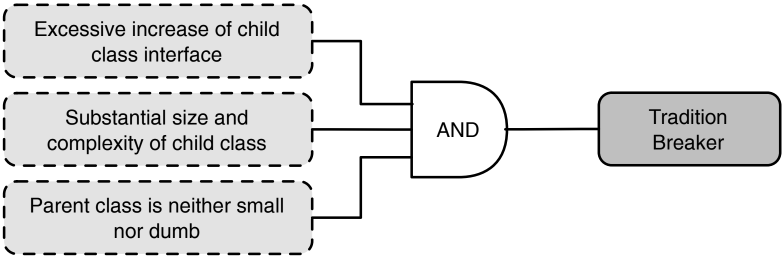
Excessive increase of child class interface
Substantial size and complexity of child class
Parent class is neither small nor dumb
AND
Tradition Breaker

Fig. 7.8. The Tradition Breaker detection strategy.

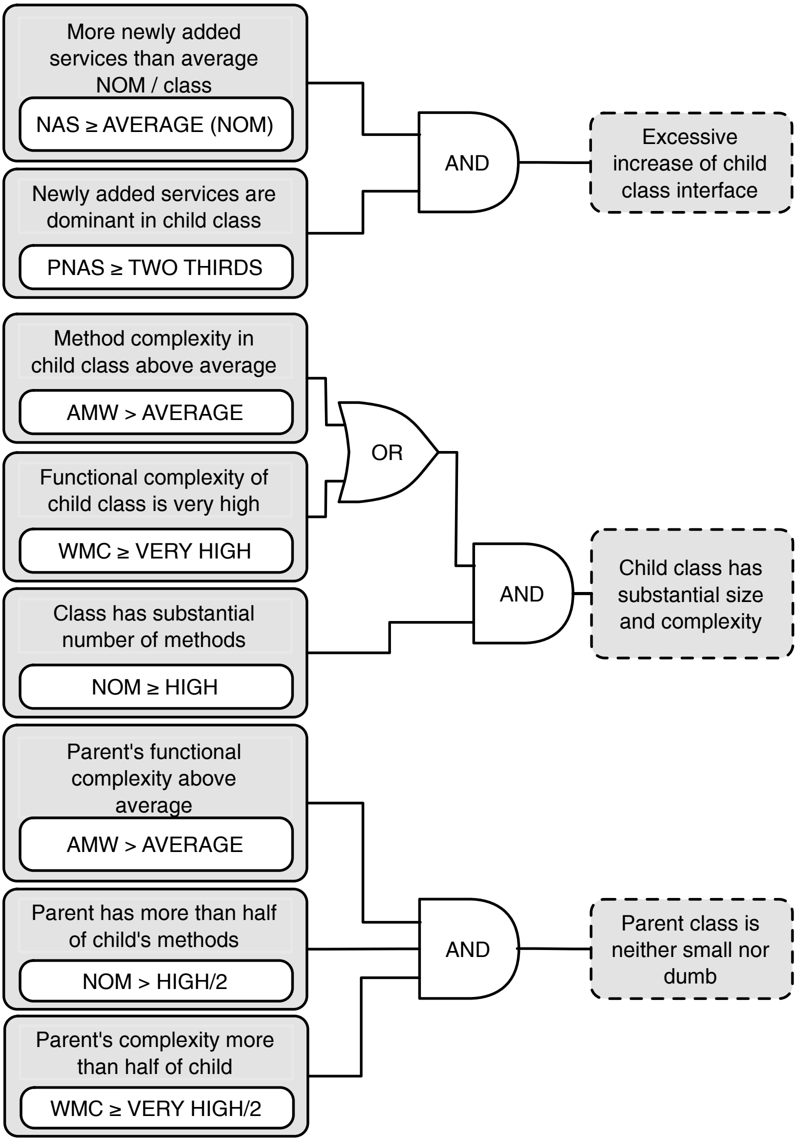
More newly added services than average NOM / class
NAS ≥ AVERAGE (NOM)
Newly added services are dominant in child class
PNAS ≥ TWO THIRDS
AND
Excessive increase of child class interface
Method complexity in child class above average
AMW > AVERAGE
Functional complexity of child class is very high
WMC ≥ VERY HIGH
OR
Class has substantial number of methods
NOM ≥ HIGH
AND
Child class has substantial size and complexity
Parent's functional complexity above average
AMW > AVERAGE
Parent has more than half of child's methods
NOM > HIGH/2
Parent's complexity more than half of child
WMC ≥ VERY HIGH/2
AND
Parent class is neither small nor dumb

Fig. 7.9. Main components of the Tradition Breaker detection strategy

Detection

In Fig. 7.8 we see a high-level view of the detection rule for a Tradition Breaker. There are three main conditions that must be simultaneously fulfilled for a class to be put on the blacklist of classes that break the inherited tradition by the interface that they define. These conditions are:

- The size of the public interface of the child class has increased excessively compared to its base class.
- The child class as a whole has a considerable size and complexity.
- The base class, even if not as large and complex as its child, must have a "respectable" amount of functionality defined, so that it can claim to have defined a tradition.

1. **Excessive increase of child class interface.** To quantify the evolution of a child's public interface compared to that of its parent, we use two measures: (1) Newly Added Services (NAS) tells us in absolute values how many public methods were added to the class; and (2) the Percentage of Newly Added Services (PNAS) which shows us the percentile increase, i.e., how much of the class's interface consists of newly added services. We used these metrics in the following way:
   a) **More newly added services than average number of methods per class.** This threshold is based on the statistical information related to the number of methods per class (see Table 2.1), using the following logic. If a class adds more new methods than the *average* number of methods (public or not) of a class then the measured class is an outlier with respect to NAS. For Java this average value[1] is 6.5.
   b) **Newly added services are dominant in child class.** We use this metric to make sure that the absolute value provided by the NAS is a significant part of the entire interface of the measured class. Therefore, PNAS is a normalized metric and we set the threshold so that NAS represents at least *two-thirds* of the public interface.
2. **Child class has substantial size and complexity.** To speak about a relevant Tradition Breaker the child class must contain a substantial amount of functionality. This means that it must have a substantial size (measured in this case by the number of methods) and accumulate a significant amount of logical complexity. Therefore we require either the average complexity or the total complexity of the class to be high. An additional requirement is that the child class has a significant number of methods (NOM). We use the following metrics (see Fig. 7.9):

[1] Computed as the average between the *lower value* and *upper value* of NOM/Class.

   a) **Method complexity in child class above average.**
   b) **Functional complexity of child class is very high.**
   c) **Class has a substantial number of methods.**

3. **Parent class is neither small nor dumb.** We cannot say that a child class breaks a tradition if the tradition defined by the parent class is insignificant. In other words, this term sets a minimal condition on the size and complexity of the parent class, i.e., this must satisfy *at least half* of the requirements imposed on the child class. Additionally, its average complexity must be higher than the average value. In this context, AMW and WMC are the two metrics used to quantify the average and the total amount of functional complexity respectively, while NOM quantifies the size of the class in terms of method number. The used metrics are:
   a) **Parent's functional complexity above average.**
   b) **Parent has more than half of child's methods** With respect to NOM, the parent class should satisfy at least *half* of the requirements we set for the child (see term "*Child class has substantial size and complexity*").
   c) **Parent's complexity more than half of child** With respect to Weighted Method Count (WMC), the parent class should satisfy at least *half* of the requirements we set for the child (see term "*Child class has substantial size and complexity*").

Example

In Fig. 7.10 we see a *System Complexity* view of the hierarchy whose root class is named FigNodeModelElement. Visually striking is that the hierarchy is top-heavy (the root class is by far the largest in terms of methods and attributes) and unbalanced (there is a sub-hierarchy on the left). Moreover, many direct subclasses of FigNodeModelElement look similar "from the outside" (i.e., they have a similar shape, pointing to a possible duplication problem), and as we will see also from the inside.

From the point of view of the disharmonies, nearly half of the classes of this hierarchy are afflicted by at least one of two classification disharmonies: Refused Parent Bequest(145) or Tradition Breaker.

Among the subclasses of FigNodeModelElement there is one in particular which is striking because it is the only one which is both affected by Refused Parent Bequest(145) and is also a Tradition Breaker, namely FigObject. Additionally this class is also a Brain Class(97) that contains two methods which are Brain Method(92).

Refactoring

If we want to remove a Tradition Breaker then we need to follow the

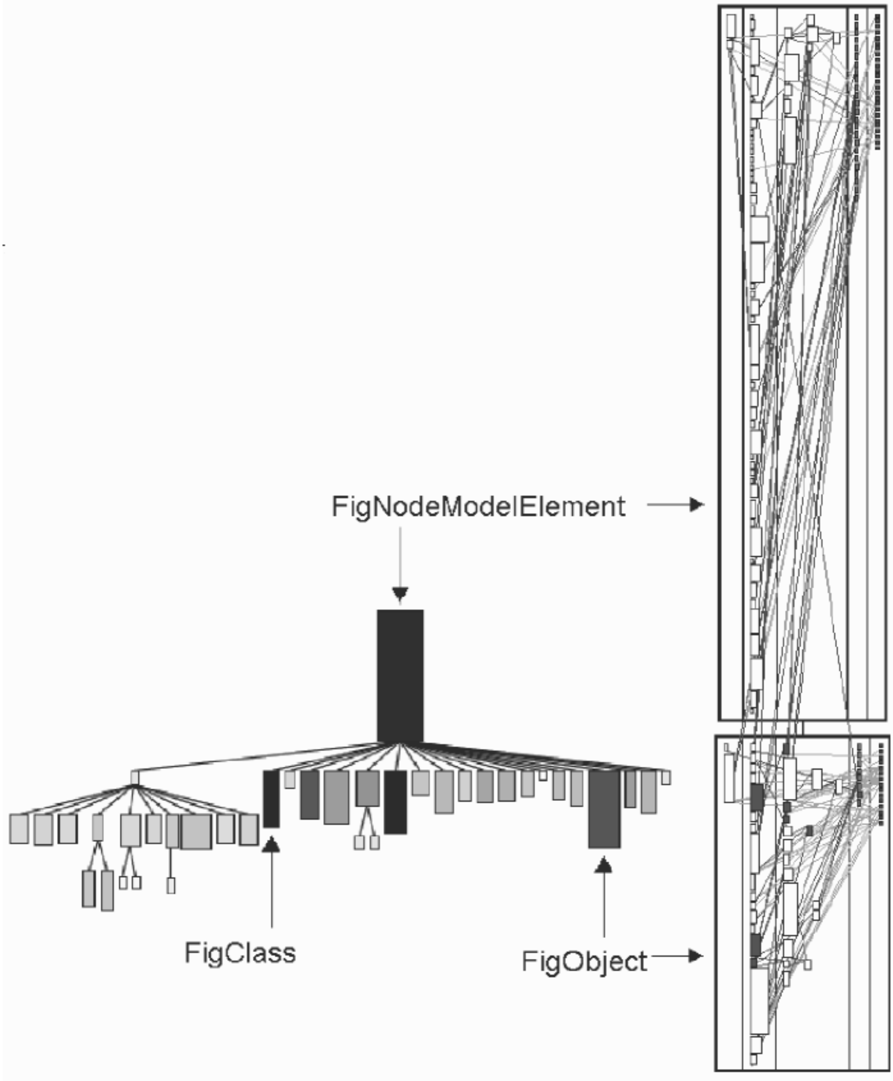

Fig. 7.10. A *System Complexity* view of the FigNodeModelElement hierarchy.

detailed (inspection and refactoring) process depicted in Fig. 7.11. The figure has four areas (labeled A, B, C and D) corresponding to one of the four identified cases, which may also co-exist, for a Tradition Breaker: irrelevant tradition in subclass, denied tradition in base class, double-minded subclass, or misplaced subclass.

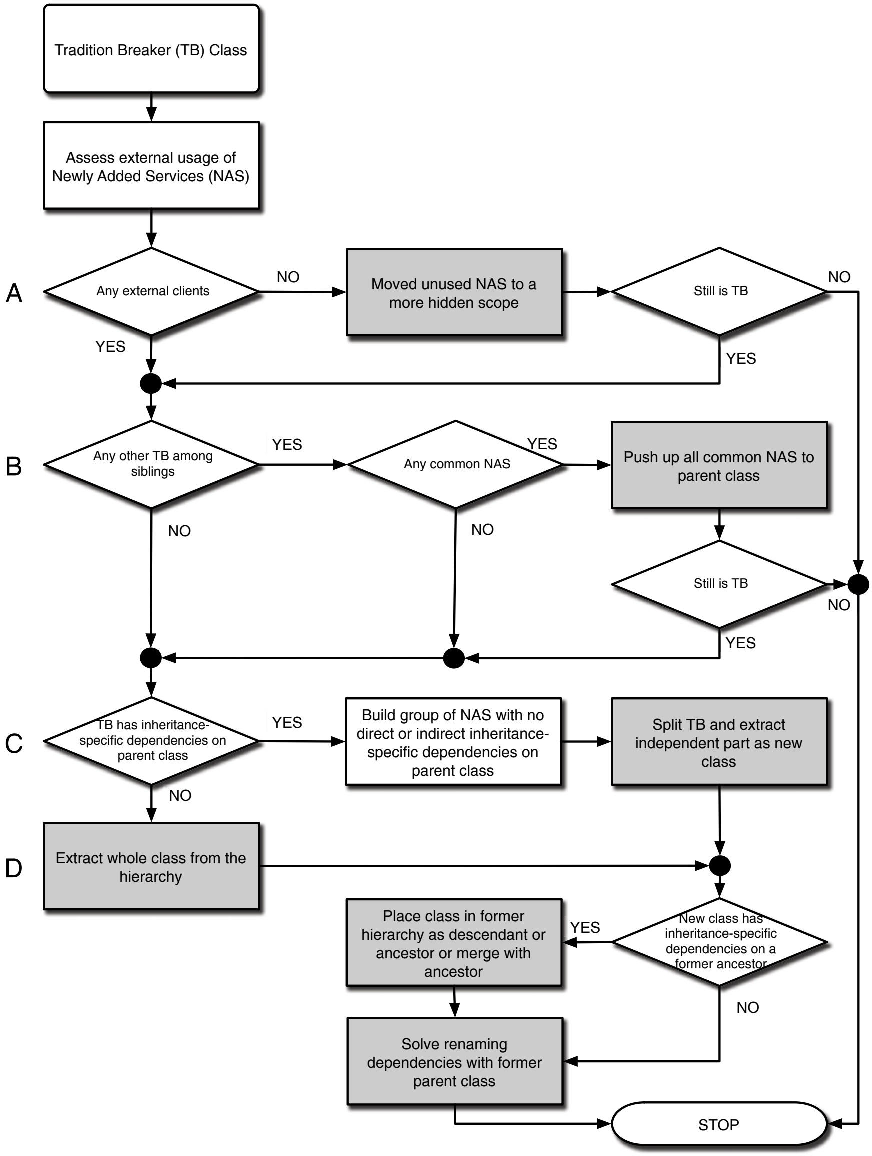
Tradition Breaker (TB) Class
Assess external usage of Newly Added Services (NAS)
A
Any external clients
NO
Moved unused NAS to a more hidden scope
Still is TB
NO
YES
YES
B
Any other TB among siblings
YES
Any common NAS
YES
Push up all common NAS to parent class
NO
NO
Still is TB
NO
YES
C
TB has inheritance-specific dependencies on parent class
YES
Build group of NAS with no direct or indirect inheritance-specific dependencies on parent class
Split TB and extract independent part as new class
NO
D
Extract whole class from the hierarchy
Place class in former hierarchy as descendant or ancestor or merge with ancestor
YES
New class has inheritance-specific dependencies on a former ancestor
NO
Solve renaming dependencies with former parent class
STOP

Fig. 7.11. Inspection and refactoring process for a Tradition Breaker.

Case A: Irrelevant Tradition

In this case the derived class has an excessively large interface, i.e., it includes in its interface methods that should have been declared protected or private. In other words, the methods newly added in the interface of the Tradition Breaker class are just helper methods, mistakenly declared public. This can be be found out by analyzing the usages of the method from other classes.

Case B: Denied Tradition

The base class does not include a set of services that are implemented in all (or most) derived classes. Consequently, it is common that some of the Tradition Breaker's siblings also show the symptoms of a Tradition Breaker. In most of these cases Duplication(102) is also present.

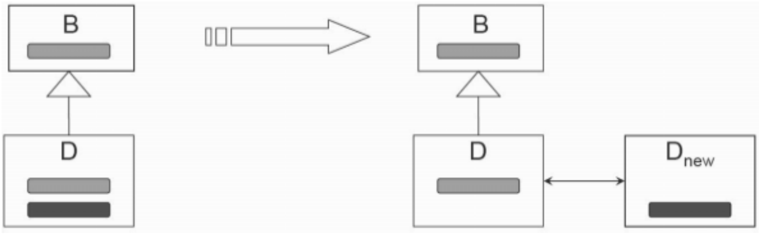

Fig. 7.12. Extracting the "second mind" of a Tradition Breaker to a separate class.

Case C: Double-Minded Descendant

In this case the problem is that the derived class is "double-minded", whereby only a part of its interface (and implementation) belongs to the hierarchy where it was placed. The part of the class that wants to stay in the hierarchy can be identified by a set of methods that override/specialize/use methods of the base class. The "other mind" of the class breaks the tradition, by doing something totally different, that has nothing in common with the base class. In this case, the part that does not belong to the hierarchy could be moved outside the hierarchy to a separate class (see Fig. 7.12).

Case D: Misplaced Descendant

The extreme case of Case C is when the whole Tradition Breaker defines a behavior that is not extending (specializing) in any way the behavior found in its base class. Thus, the interfaces of the base class and of the derived class are totally different and there is no real inheritance-specific dependencies on the base class. When this is the case, it is highly probable that the class is also affected by Refused Parent Bequest(145).

7.5 Recovering from Classification Disharmonies

Where to Start

In order to recover from from classification disharmonies (i.e., design problems related to inheritance) it is insufficient to look at the individual suspect classes; hierarchies must be analyzed as a whole. In this context it becomes important to know *how to group* the different classes identified as affected by various disharmonies and also, how to prioritize the hierarchies that need more urgent attention. In practice, we use the following criteria in selecting the hierarchies with the most significant amount of classification disharmonies:

- Hierarchies with more classes affected by classification disharmonies have priority.
- If the disharmonies are distributed on many hierarchy levels (i.e., if the sub-hierarchy affected by disharmonies is *deep*) the inspection priority for the hierarchy increases.
- Hierarchies where most distinct classification disharmonies appear have a higher priority.
- Hierarchies where other types of disharmonies (i.e., identity and collaboration) co-exist with violations of classification harmony must also be regarded with increased interest.

For the purpose of prioritizing the hierarchies to be inspected first, we mainly use the following quantification means:

- *Number (and Percentage) of Classes with Classification Disharmonies.* These numbers tell us how much the classification disharmonies are spread within the hierarchy. In addition to the absolute number of classes, we also display the percentage of disharmonious classes, as this indicator is more relevant and easier to interpret for larger hierarchies. The higher these numbers are, the higher also is the probability that the whole hierarchy must be restructured.
- *Hierarchy Depth of Disharmonious Classes.* In addition to the previous values, we found that it is important to know also how deep in the hierarchy (rooted by the class in the table) we find disharmonies. If disharmonies are propagated on many inheritance levels then such hierarchies must be definitely revisited.
- *Distinct Classification Disharmonies in Hierarchy.* Often the same disharmony (Tradition Breaker(152)) affects many subclasses of a

hierarchy. But if the number of *distinct* problems in the same hierarchy is high then the hierarchy has a more complex problem that needs to be addressed. At the same time the co-existence of some classification disharmonies (e.g., Refused Parent Bequest(145) and Tradition Breaker(152)) could help us in addressing the problem properly.

- *Distinct Number of Other Disharmonies in Hierarchy.* To have an even better overview of all possible disharmonies that appear in the same hierarchy, we also count how many *distinct* design problems, other than the classification ones (i.e., problems related to identity or collaboration), can be found.

How to Start

Assessing and improving the classification harmony of a system is a complex process, because a large number of classes are involved (i.e., all (or most) of the classes in the hierarchy) and also because the real cause of such design problems is not localized in one single class (e.g., a child class is detected, but the real cause of the problem is in the base class). Additionally, the inspection and refactoring process is painful because of the existence of various correlated design disharmonies (see Fig. 4.12) that might occur in the classes of the hierarchy and that must be solved at the same time.

Because of all these reasons, for each of the two classification disharmonies discussed in this chapter, i.e., Refused Parent Bequest(145) and Tradition Breaker(152), we addressed in detail the potential refactoring solutions. We noticed that the *order* in which the problems are addressed is very important. Therefore, we recommend inspecting and refactoring each disharmonious hierarchy in your system using the sequence described in Fig. 7.13.

Doing the refactorings in this order is important because on the one hand the refactoring action for one disharmony can have positive consequences with respect to the following ones (in the sense that the refactoring effort is reduced); but on the other hand they can also introduce additional cases of classification disharmonies that must be addressed as well.

Let us see how it happens. We start by solving the Duplication(102) problem. By doing so, it could be possible that methods are extracted from some siblings and moved to their parent class. This can contribute in some cases to a reduction – or even a total elimination – of the Tradition Breaker(152) disharmony. But at the same time,

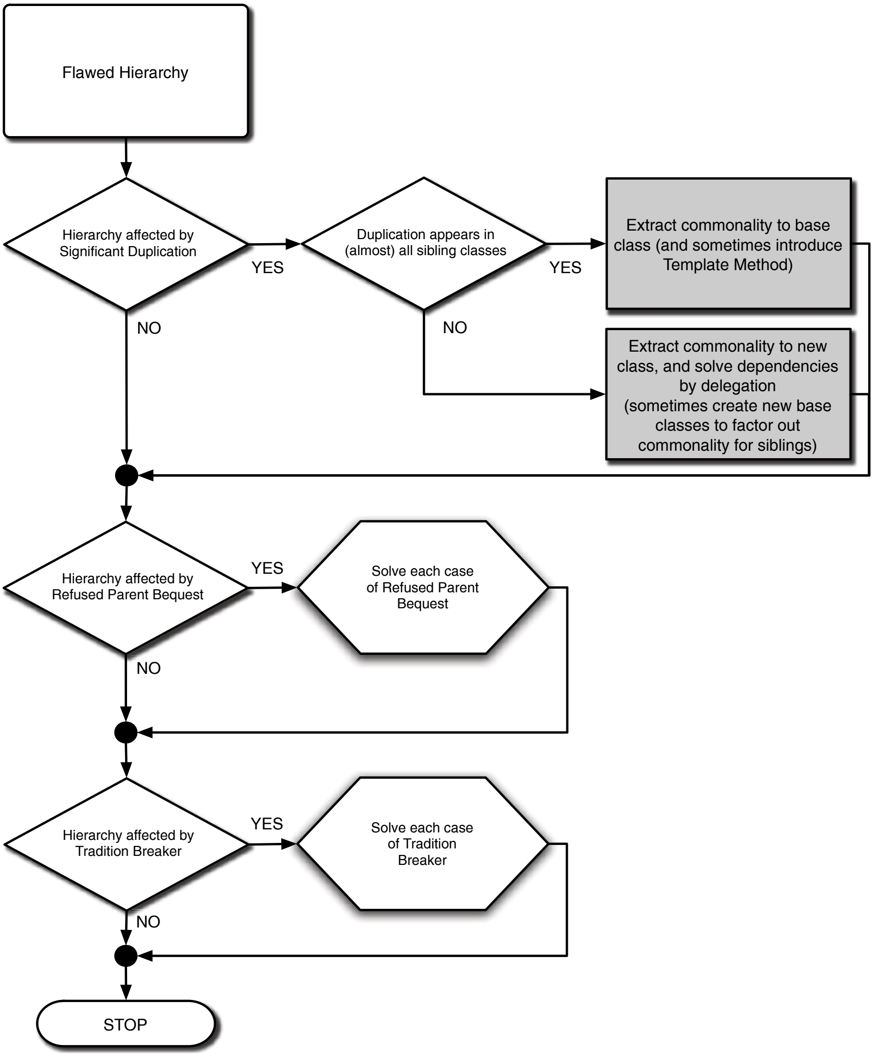
Flawed Hierarchy
Hierarchy affected by Significant Duplication
YES
Duplication appears in (almost) all sibling classes
YES
Extract commonality to base class (and sometimes introduce Template Method)
NO
NO
Extract commonality to new class, and solve dependencies by delegation (sometimes create new base classes to factor out commonality for siblings)
Hierarchy affected by Refused Parent Bequest
YES
Solve each case of Refused Parent Bequest
NO
Hierarchy affected by Tradition Breaker
YES
Solve each case of Tradition Breaker
NO
STOP

Fig. 7.13. How to address classification disharmonies.

as we move new methods to a parent class, we might cause a Refused Parent Bequest(145) disharmony for some other siblings, as the bequest provided by the parent class has increased as a result of the refactoring. Thus, it is important to deal with Duplication(102) before addressing the Refused Parent Bequest(145) and the Tradition Breaker(152) disharmonies.

Now, which one of these two problems should we address next? We suggest dealing first with Refused Parent Bequest(145), because by refactoring a part of the class (or even the whole child class) needs to be removed from the hierarchy. Thus, this provides a new perspective in dealing with the cases of Tradition Breaker(152).