

Collaboration Disharmonies

6.1 Collaboration Harmony Rule

Collaboration disharmonies are design flaws that affect several entities at once in terms of the way they collaborate to perform a specific functionality.

The principle of low coupling is advocated by all the authors that propose design rules and heuristics for object-oriented programming. Although having different forms or emphases they all converge in saying that coupling of classes should be minimized. Yet, a tension exists between the aim of having low coupled systems and the fact that an amount of collaboration among objects (and thus coupling) is necessary in all non-trivial systems. Responsibility-driven approaches stress the fact that classes should implement well identified responsibilities often by delegating work to others and collaborate with a clearly identified and limited set of collaborators [WBM03].

A harmonious collaboration is one that maintains a balance between the inherent need for communication among the entities (i.e., methods and classes) of a system and the demand to keep this coupling to a minimum. The collaboration harmony rule is:

Collaborations should be only in terms of method invocations and have a limited extent, intensity and dispersion

Collaboration Rule

Collaborations should be only in terms of method invocations and have a limited extent, intensity and dispersion

Rationale

The idea behind this rule is summarized by Lorenz and Kidd ¹:

You want to leverage the services of other classes, but you want to have services at the right level, so that you want to know only about a limited number of objects and their services. [...] If you had to interact with all the indirectly related objects, we'd have a tangled web of interdependencies and maintenance would be a nightmare [LK94].

The rule refers both to outgoing and incoming dependencies. *Excessive* outgoing dependencies are undesirable because the more one uses the others, the more *vulnerable* (to changes and malfunction) one becomes. *Excessive* incoming dependencies are also undesirable because the more one is used by the others, the more responsible and thus *immutable* (i.e., rigid, stable, less evolvable) one becomes. At the same time, note that excessive incoming dependencies may also be a good sign of design and functionality reuse, with one condition: the used interfaces are *stable*. An example is given by class libraries implementing collections or common infrastructure. Additionally, it is important to take into account the important role of stable interfaces to support changes. Interfaces play an important role in shielding clients from specific implementation concerns hence reducing the impact of changes.

Practical Consequences

- **Limit collaboration intensity** – *Operations should collaborate (mainly unidirectional) with a limited number of services provided by other classes.*

¹ The rule is also very much related to Pelrines's Object Manifesto which states: *Be private: do not let anybody touch your private data. Be lazy: Delegate as much as possible*

- **Limit collaboration extent** – *Operations (and consequently their classes) should collaborate² with operations from a limited number of other classes.*

This is a restatement of “A harmonious system must have services defined at the proper level, so that you need to collaborate directly only with a limited number of other abstractions” [LK94].

- **Limit collaboration dispersion** – *The collaborators (i.e., invoked and/or invoking operations) of an operation should have a limited dispersion within the system. Thus, one should try to make an entity collaborate closely only with a selected set of entities, with a preference for entities (in decreasing order) located in the (0) same abstraction; the (1) same hierarchy; the (2) same package (or sub-system).*

² *The term Collaborate* refers both to the active (i.e., call another operation) and to the passive (i.e., be called (invoked) by another operation) aspects.

6.2 Overview of Collaboration Disharmonies

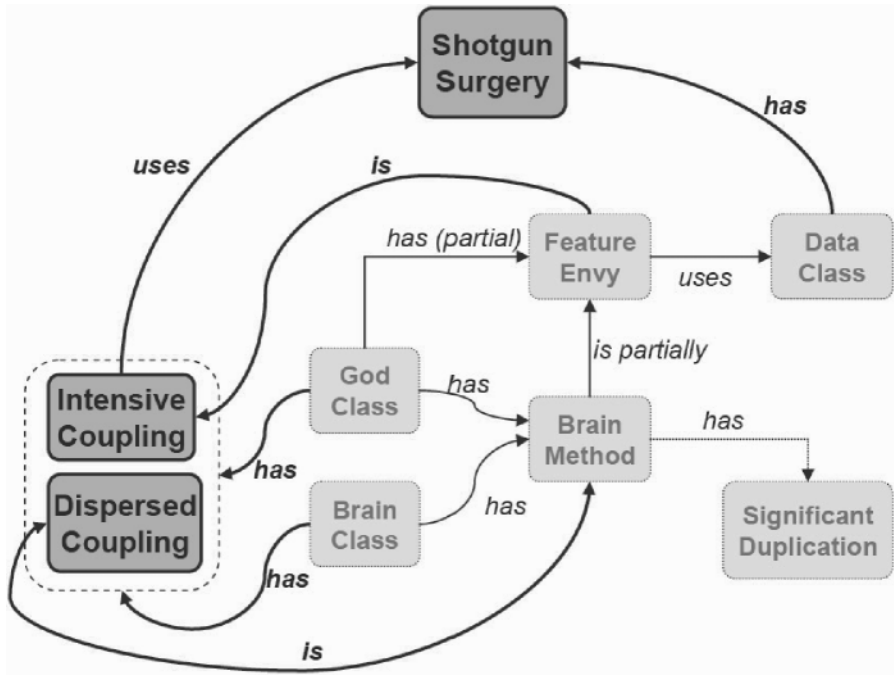


Fig. 6.1. Correlation web of collaboration disharmonies.

The *Collaboration Rule* shows that, especially concerning outgoing coupling, the problem is an *excessive* number of operations which are called from the *disharmonious* operation. A second important aspect is the distribution (dispersion) of these called operations on classes.

Considering the practical consequences above, we can say that an operation is disharmonious in terms of collaboration if it has too many invocations of many other methods.

We capture these collaboration disharmonies using two *detection strategies*, namely Intensive Coupling(120) and Dispersed Coupling(127). While the former captures the case where the method intensively uses a reduced number of classes (invoking lot of method of a particular class), the latter deals with the situation where the dependencies of the disharmonious method are very much dispersed among many classes (invoking methods from too many classes).

In a collaboration, not only the server methods can be disharmonious, but also the client code. Fowler [FBB⁺99] mentions the case when a small change in a part of a system causes lots of changes to many classes, dispersed all over the rest of the system. They call this bad smell Shotgun Surgery. Inspired by this we captured the disharmony in which a method is excessively invoked by many methods located in many classes (Fig. 6.1), and as a tribute to our inspiration source we called it Shotgun Surgery(133).

Fowler's Shotgun Surgery smell can also take the form of a piece of code which is replicated over and over again in various methods, belonging to various classes which might otherwise not look coupled to each other. For example, when a class is a Data Class(88), its clients often duplicate functionality that would be normally be under the responsibility of that class. Thus, for such cases the Duplication(102) disharmony can also be considered a collaboration disharmony.

6.3 Intensive Coupling

Description

One of the frequent cases of excessive coupling that can be improved is when a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes (see Fig. 6.2). In other words, this is the case where the communication between the client method and (at least one of) its provider classes is excessively verbose. Therefore, we named this design disharmony Intensive Coupling.

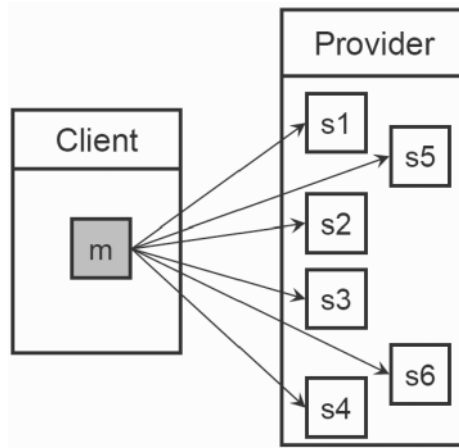


Fig. 6.2. Illustration of Intensive Coupling

Applies To

Operations i.e., methods or standalone functions.

Impact

An operation which is intensively coupled with methods from a handful of classes binds it strongly to those classes. Oftentimes, Intensive Coupling points to a more subtle problem i.e., the classes providing the many methods invoked by the Shotgun Surgery method do not provide a service at the abstraction level required by the client method. Consequently, understanding the relation between the two sides (i.e., the client method and the classes providing services) becomes more difficult.

The *detection strategy* is based on two main conditions that must be fulfilled simultaneously: the function invokes many methods and the invoked methods are not very much dispersed into many classes (Fig. 6.3).

Additionally, based on our practical experience, we impose a minimal complexity condition on the function, to avoid the case of configuration operations (e.g., initializers, or UI configuring methods) that call many other methods. These configuration operations reveal a less harmful (and hardly avoidable) form of coupling because the dependencies can be much easily traced and solved.

The *detection strategy* is composed of the following heuristics (see Fig. 6.3):

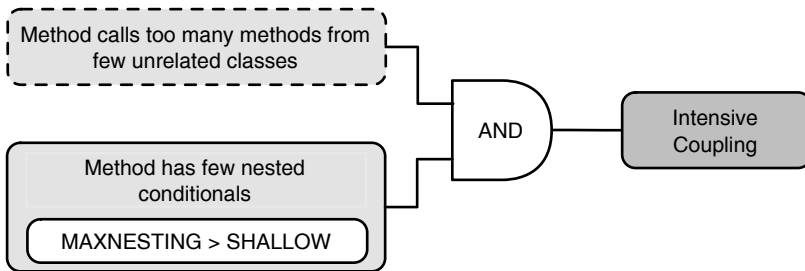


Fig. 6.3. Intensive Coupling detection strategy.

1. **Operation calls too many methods from a few unrelated classes.**

The basic condition for a method or function to be considered as having an Intensive Coupling is to call many methods belonging to a few classes (Fig. 6.4). By “unrelated classes” we mean that the provider classes are belonging to the the same class hierarchy as the definition class of the invoking method. We distinguish two cases:

- a) Sometimes a function invokes many other methods (more than our memory capacity) from different classes. Usually among the provider classes there are two or three from which several methods are invoked.
- b) The other case is when the number of invoked methods does not exceed our short-term memory capacity, but all the invoked methods belong to only one or two classes. Thus, the number of methods invoked from the same provider class is high.

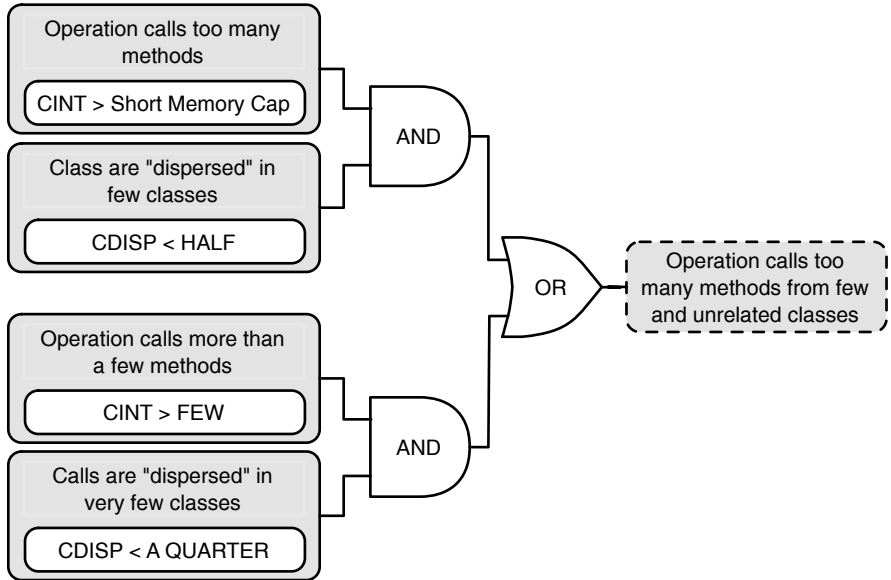


Fig. 6.4. In Intensive Coupling operation calls too many methods from a few unrelated classes

Therefore, we have two branches: one for detecting intensive couplings which are concentrated in one or two classes, and another one dedicated to the more general case when the dispersion ratio of the invoked methods is below 50%.

The used heuristics in the first case are:

- a) **Operation calls too many methods.** Too many refers to a number greater than the number of items that can be memorized by the short-term memory. If the caller operation is a method, than only those provider methods are counted that are outside the scope of the caller's definition class.
- b) **Calls are dispersed in a few classes.** The methods invoked by a client operation have a low grade of dispersion, i.e., the provider methods belong to a few classes. The threshold tells us that in average more than two methods are invoked from the same provider class.

The used heuristics in the second case are:

- a) **Operation calls more than a few methods.**

- b) **Calls are dispersed in very few classes.** The called methods have a very low grade of dispersion, i.e., the threshold tells us that in average more than two methods are called from the same provider class.
2. **Operation has nested conditionals.** A function that calls many methods, but is flat – in terms of the nesting level of its statements – is less complex and from our experience this coupling cases prove to be often less relevant. In many cases such methods are initializers or configuration functions that are less interesting for both understanding and improving the quality of a design. Therefore, as mentioned earlier, we set this condition so that the calling function should have a non-trivial nesting level.

In Fig. 6.5 we see that `ClassDiagramLayouter` is intensively coupled with a few classes, especially with `ClassDiagramNode`. The blue edges represent invocations between the methods in the classes. The red nodes represent non-model classes, i.e., Java library classes.

Example

The classes have been laid out according to the invocation sequence: above `ClassDiagramLayouter` we place all classes that use it, while below it are all classes whose methods get used, i.e., invoked by its methods.

In Fig. 6.6 we see that `ClassDiagramLayouter` is coupled to `ClassDiagramNode` mainly because of four large methods, two of which have previously been detected as a Brain Method(92): (1) `layout`, (2) `weightAndPlaceClasses` (3) `rankPackagesAndMoveClassesBelow` and (4) `layoutPackages`.

In more detail, the method `weightAndPlaceClasses` invokes 11 methods of the class `ClassDiagramNode` which by looking at its *Class Blueprint* seems to be a mere data holder without complex functionality. The same goes for the method `layout` which uses 6 methods of `ClassDiagramNode`. It looks as, after a few iterations, `ClassDiagramLayouter` could eventually become a God Class(80).

A strongly suggested refactoring in this case is splitting those methods, since they do several things at once, as their names suggest. For example, `weightAndPlaceClasses` could be split into a method that weighs and another one that places the classes.

The prediction about `ClassDiagramLayouter` eventually becoming a God Class(80) or at least a complex class is supported by the fact that so far as the other classes in these figures are concerned, `ClassDiagramLayouter` uses only small parts of them. This does not re-

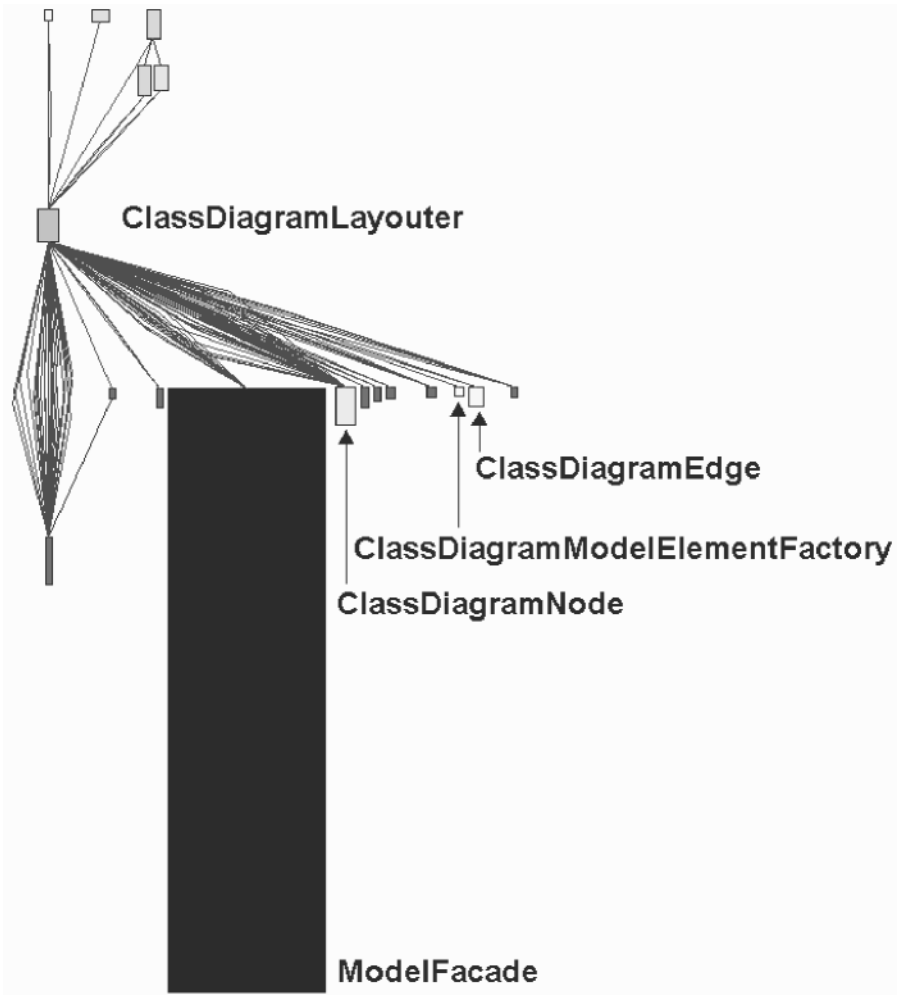


Fig. 6.5. The class `ClassDiagramLayouter` is intensively coupled with a few classes, especially `ClassDiagramNode`. The red classes are non-model classes, i.e., belong to the Java library. The classes have been laid out according to the invocation sequence: above `ClassDiagramLayouter` we place all classes that use it, while below it are all classes whose methods get used, i.e., invoked by its methods.

ally represent a problem, although some of the coupling relationships seem to be very weak and probably do not require much work to be cut off and decrease the couplings of `ClassDiagramLayouter`.

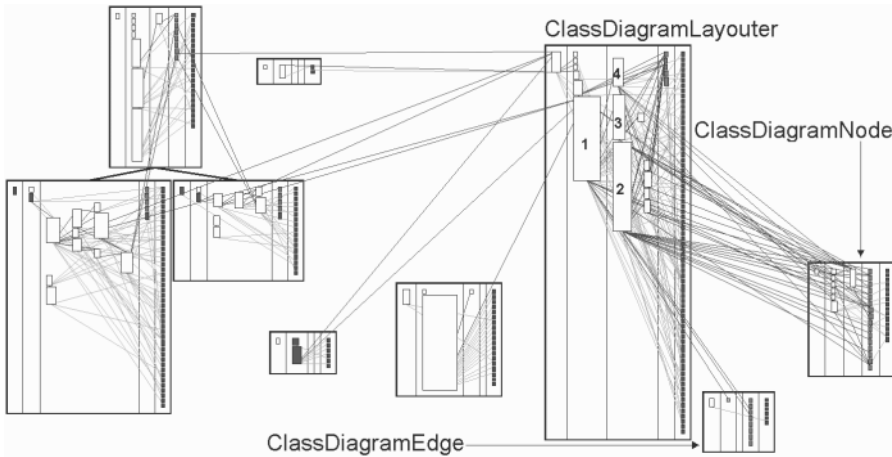


Fig. 6.6. The class `ClassDiagramLayouter` is intensively coupling with a few classes, especially `ClassDiagramNode`.

In the case of an operation with Intensive Coupling the intensity of coupling is high, while the dispersion is low. This guarantees that we will find one or more *clusters of methods* invoked from the same (provider) class. Therefore, a first refactoring action is to *try* to define a new (more complex) service in the provider class and replace the multiple calls with a single call to the newly defined method (see Fig. 6.7).

Refactoring

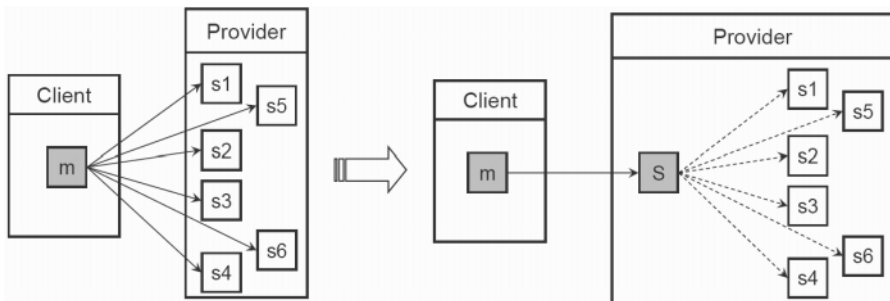


Fig. 6.7. The essence of the refactoring solution in case of Intensive Coupling

If this cluster of methods invoked from the same class consists mainly of lightweight methods, some of which are affected by Shotgun Surgery(133), then it is highly probable that the aforementioned

refactoring will also have a positive impact on the design quality of the class that contains those lightweight methods, in the sense that the provided class will offer higher-level services.

The main reasons for reducing coupling are not that the code will look cleaner after. Most of the time reducing coupling is required to be able to use one component without the others or to make easier the replacement of one component by another one. Therefore having smaller communication channels is an important task. However, reducing coupling between classes is a complex task. Indeed we can reduce the metrics values by grouping or factoring the methods belonging to the same class and tunnelling thus the communication between the classes. However such a practice even if it can improve the overall design of the system by making more precise the communication channel between the classes should not hide that often reducing coupling is a more complex. Indeed either a dependency was useless and this is easy to fix it or it is necessary and moving it around will not solve the root of the problem. To reduce coupling often requires to change the *flow* of the application or to introduce extra indirections. In addition the coupling can change over time and run-time registration mechanisms such as Transform Type Checks to Registration [DDN02] may be the solution to decouple clients and providers of services.

Finally, coupling or dependencies are often the results of misplaced operations, therefore it is worth checking if the Law of Demeter [LH89] or reengineering patterns like Move Behavior Close to the Data and Eliminate Navigation Code [DDN02] can be applied.

6.4 Dispersed Coupling

This disharmony reveals a complementary aspect of coupling than the one described as Intensive Coupling(120). This is the case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes (see Fig. 6.8). In other words, this is the case where a single operation communicates with an excessive number of provider classes, whereby the communication with each of the classes is not very intense i.e., the operation calls one or a few methods from each class.

Description

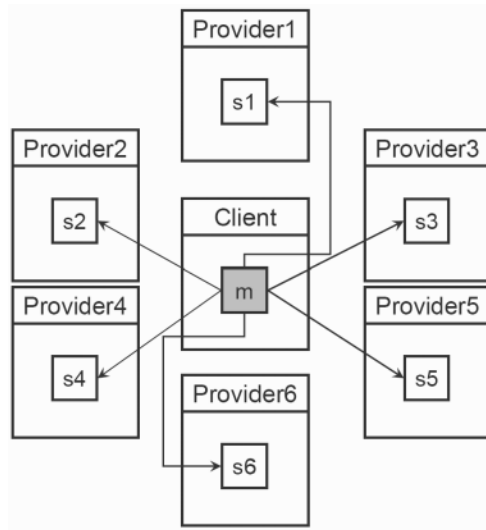


Fig. 6.8. Illustration of Dispersed Coupling

Operations, e.g., methods or standalone functions.

Applies To

Dispersively coupled operations lead to undesired ripple effects, because a change in an dispersively coupled method potentially leads to changes in all the coupled and therefore dependent classes.

Impact

Detection

The detection rule is defined in the same terms as the the one defined for Intensive Coupling(120), with only one complementary difference: we capture only those operations that have a high dispersion of their coupling (Fig. 6.9). The *detection strategy* in detail is:

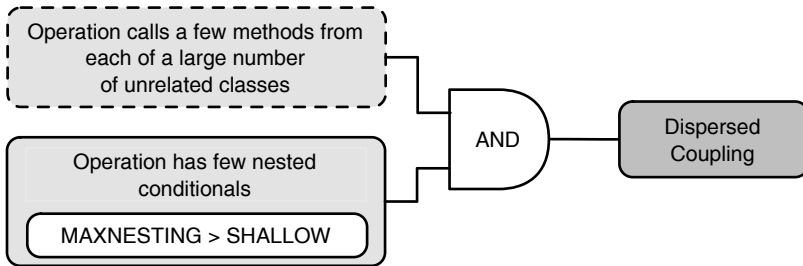


Fig. 6.9. Dispersed Coupling detection strategy

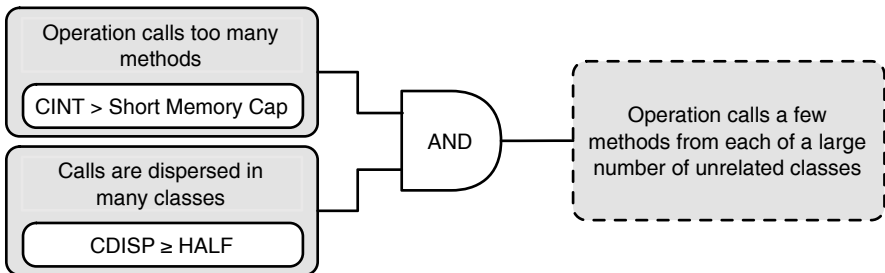


Fig. 6.10. In Dispersed Coupling operation calls a few methods from each of a large number of unrelated classes.

1. **Operation calls a few methods from each of a large number of unrelated classes.** This term of the detection rules imposes two conditions: an intensive coupling, i.e., the invocation of many methods from other classes (CINT - Coupling Intensity), and a large dispersion among classes of these invoked operations (CDISP - Coupling Dispersion). The metrics used in this case are the same as those already used in the context of detecting Intensive Coupling(120).

2. **Operation has few nested conditionals.** Exactly as for Intensive Coupling(120), we also set the condition that the calling operation should have a non-trivial nesting level, to make sure that irrelevant cases (like initializer functions) are skipped.

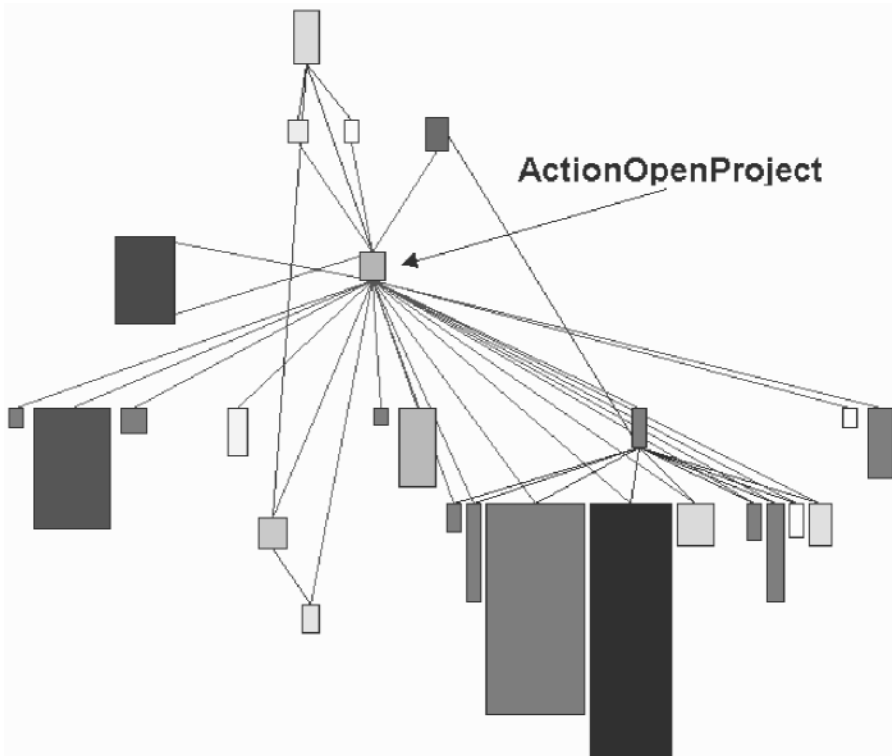


Fig. 6.11. The class `ActionOpenProject` is coupled with many classes. The red classes are non-model classes, i.e., belong to the Java library. The blue edges represent invocations.

An interesting example of Dispersed Coupling is found in class `ActionOpenProject`. We see in Fig. 6.11 that the class is coupled with many other classes. Even ignoring the calls to non-model classes (colored in red) we still see that this methods of this class invoke methods located in many other classes, resulting in a great dispersion of the invocations. Looking closer at the methods of `ActionOpenProject` we discover that most of the coupling in this class is caused by two

Example

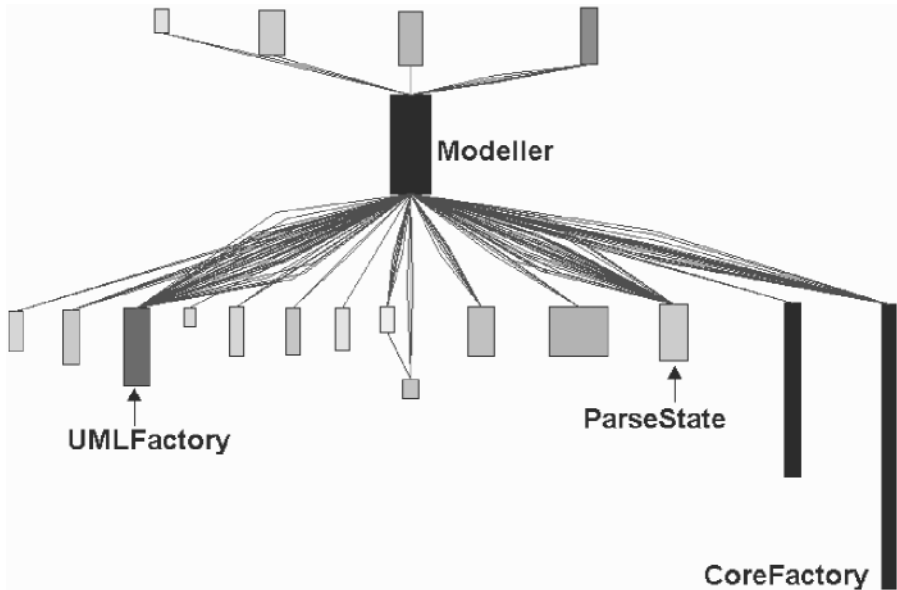


Fig. 6.12. The class `Modeller` is coupled with many classes and suffers itself from many other problems.

methods, i.e., `actionPerformed` and `loadProject`, both revealed by the *detection strategy* as being affected by Dispersed Coupling. By looking closer at these two methods we identify another interesting aspect: each of them is also a Brain Method(92). Moreover, some fragments of `actionPerformed` (exactly those fragments where many invocations appear) are also duplicated in three methods from sibling classes of `ActionOpenProject`. All these facts determine us to believe that the cause of the excessive coupling is the improper distribution of functionality among the methods of the system. In other words, the Dispersed Coupling detected in these methods is an additional sign that the two methods are doing more than one single task.

Another relevant example of Dispersed Coupling is found in a class already mentioned in the context of the Brain Method(92) disharmony, i.e., `Modeller`. This class has two methods affected by Dispersed Coupling. Again, one of the two methods (`addImport`) is a Brain Method(92); the other one (`addClassifier`) is also significantly large and complex. As we see in Fig. 6.12, `Modeller` is indeed dispersively coupled with many classes and especially coupled with `ParseState`,

CoreFactory and UMLFactory. A manual analysis of both these two methods has revealed two different causes:

1. Methods are too large and not focused on a single task. As a result, in each method, we find a portion of code that reveals an intensive coupling towards the ParseState class. Furthermore, inspecting ParseState we found out that this class provides only very simple services, which are composed into higher-level services by the client methods. This partially explains the need for many different methods invocations from that class.
2. In addClassifier apart from the aforementioned aspect, we found that the Dispersed Coupling disharmony is partially due to invocation chains that break the Law of Demeter [LH89]

Refactoring an operation affected by Dispersed Coupling is not a straight forward action. It needs more contextual information to proceed, but here are a few hints from our experience in dealing with this problem:

Refactoring

- In many cases the operation that exhibits Dispersed Coupling is also a Brain Method(92). In this case, the detailed knowledge about coupling will support the refactoring of the operation in terms of Brain Method(92). In other words, if you encountered a Brain Method the refactoring should address both aspects simultaneously.
- For the other cases (rather rare) the refactoring process should be centered on identifying called methods that are *lightweight* and/or affected by Shotgun Surgery(133), always with the question in mind: Isn't there anything in the client method (i.e., the one affected by Dispersed Coupling) that could be moved to one of the lightweight methods that it invokes.
- Calling many methods from lots of classes might also have another cause than the excessively large size of the invoker method. The cause might be that the operation invokes the wrong classes, i.e., that it is coupled to classes that are at lower abstraction level than the client method [Rie96]. Thus, it would be necessary to identify the proper abstraction and let the client function communicate with that class. Although this sounds easy, it is hard to accomplish because it requires a good understanding of the system domain to be able to introduce a new abstraction into the system.

As mentioned while discussing refactoring solutions for Intensive Coupling(120), eventually coupling or dependencies are often the results of misplaced operations, therefore it is worth checking if the Law of Demeter [LH89] or the reengineering patterns Move Behavior Close to the Data and Eliminate Navigation Code [DDN02] can be applied.

6.5 Shotgun Surgery

Not only *outgoing* dependencies cause trouble, but also *incoming* ones. This design disharmony means that a change in an operation implies many (small) changes to a lot of different operations and classes [FBB⁺99] (see Fig. 6.13). This disharmony tackles the issue of strong *afferent* (incoming) coupling and it regards not only the coupling *strength* but also the coupling *dispersion*.

Description

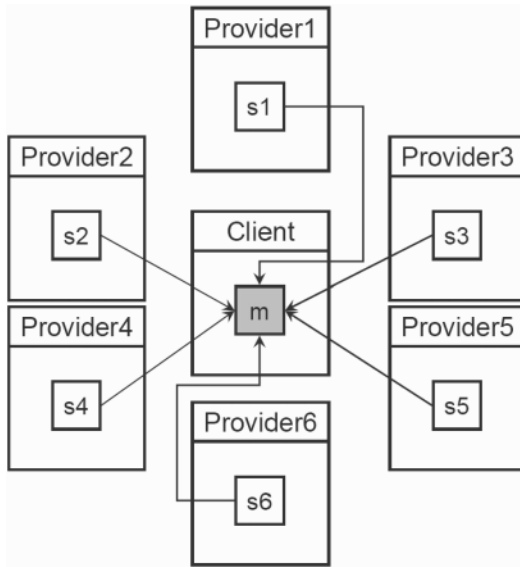


Fig. 6.13. Illustration of Shotgun Surgery

Operations, e.g., methods or functions.

Applies To

An operation affected by Shotgun Surgery has many other design entities depending on it. Consequently, if a change occurs in such an operation myriads of other methods and classes might need to change as well. As a result, it is easy to miss a required change causing thus maintenance problems.

Impact

Detection

We want to find the classes in which a change would significantly affect many other places in the system. In detecting the methods most affected by this disharmony, we consider both the *strength* and the *dispersion* of coupling. In contrast to Intensive Coupling(120) and Dispersed Coupling(127), here we are interested exclusively in *incoming dependencies* caused by function calls. In order to reveal especially those cases where dependencies are harder to trace, we will count only those operations (and classes) that are neither belonging to the same class nor to the same class hierarchy with the measured operation.

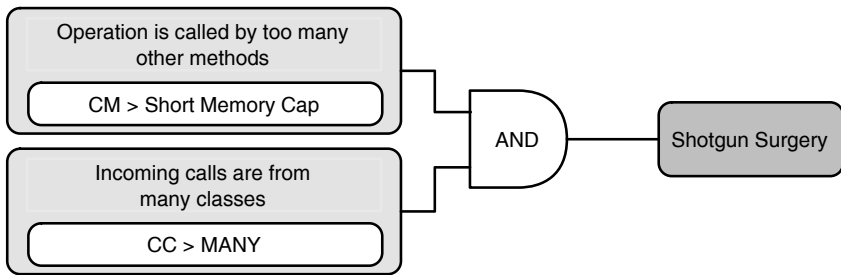


Fig. 6.14. Shotgun Surgery *detection strategy*

Based on all the considerations above, the detection technique is now easy to describe (see Fig. 6.14). First, we pick up those functions that have a *strong* change impact, and from these we keep only those that also have a high *dispersion* of changes. The *detection strategy* in detail is:

1. **Operation is called by too many other operations.** When a change in the measured operation occurs we must fix all the other operations that depend on it. If this exceeds our short-term memory capacity the risk of missing a dependency increases. This justifies both the selection of the metric and of the threshold. An alternative way to quantify the strength of incoming dependencies is to count the *number of calls* instead of *the number of callers* (like CM (Changing Methods) does). The metric called *Weighted Changing Method* (WCM) defined it [Mar02a] does just that.
2. **Incoming calls are from many classes.** Using this metric and this threshold has the following rationale: assuming that we have

two operations, and that a change in each of them would affect 20 other operations, from these two, the one for which the 20 clients are spread over more classes is worse than the other one. In other words, if all dependencies were to come from methods of a few classes then the potential changes that need to be operated on these client methods would be more localized, reducing thus the risk of missing a needed change. As a consequence, the maintenance effort (and risk) involved in managing all changes would be more reduced. Therefore, we use the CC (Changing Classes) metric to quantify the *dispersion* of the changes, so that only those Shotgun Surgery functions causing most maintenance problem are detected.

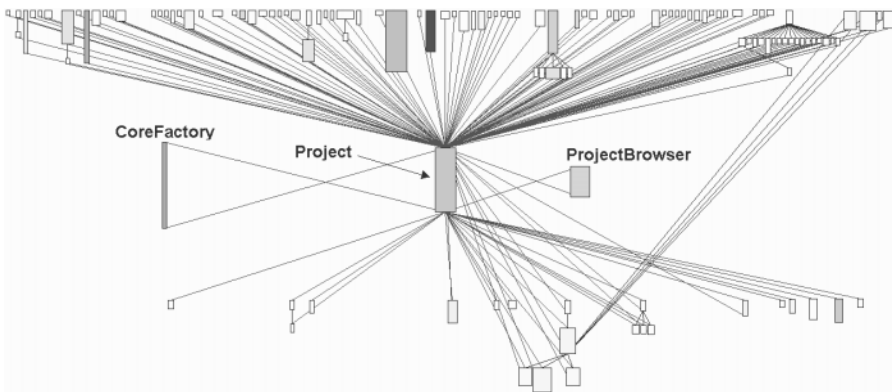


Fig. 6.15. Project provides an impressive example of a class with several methods affected by Shotgun Surgery(133). Due to these methods, Project is coupled with 131 classes (ModelFacade has been elided from the screenshot). Furthermore, the class has cyclic invocation dependencies with ProjectBrowser and CoreFactory. In the figure, the classes above Project depend on it, while Project itself depends on (i.e., invokes methods of) the classes below it.

In Fig. 6.15 we see an extreme case of Shotgun Surgery(133) that involves several methods of class Project. The class is coupled with 131 classes (10% of *ArgoUML*) and has cyclic invocation dependencies with the classes ProjectBrowser and CoreFactory (the second largest class in the system). The classes above Project depend on it, while Project itself depends on (i.e., invokes methods of) the classes below

Example

it. The view reveals how fragile the system is if a major change is performed on the class `Project`. Lots of classes in the whole system are potentially affected by changes.

Refactoring

How should we deal with Shotgun Surgery? We identified a number of refactoring options:

- Move more responsibility to the classes defining Shotgun Surgery methods, from the client classes of these functions. Do this especially when the definition classes of the Shotgun Surgery methods is *small* and/or *not complex* and/or it is or has a tendency to become a Data Class(88). Move Behavior Close to the Data [DDN02] presents step by step how to move behavior close to the data it uses and can be helpful here.
- The Shotgun Surgery methods that are very large and complex (e.g., tending to become a Brain Method) should be especially analyzed and taken care of by the maintainers of the system (e.g., by increasing the number of test cases for the method, or by trying to refactor it). We recommend this as such methods have a higher potential for change and/or malfunction potentially having a severe impact on the rest of the system. Identifying clearly the stable interfaces in the system is a good way to reduce the candidates for refactorings.

6.6 Recovering from Collaboration Disharmonies

Where to Start

As already mentioned in the previous chapter, in practice we need some criteria to prioritize the collaboration harmony offenders, so that we know which classes and methods are the most dangerous ones, the ones that require most our attention and that need a refactoring action to would improve the design. We use the following criteria in prioritizing the classes which host classification disharmonies:

- Classes that contain a higher number of disharmonious methods have priority
- Classes that are affected by other types disharmonies go first in order to reveal relation to other aspects of harmony.

How to Start

The three collaboration disharmonies Shotgun Surgery(133), Intensive Coupling(120) and Dispersed Coupling(127) address the issue of coupling from two complementary perspectives: While the issue of excessive coupling is addressed by Shotgun Surgery(133) from the provider's viewpoint, the other two tackle the same issue from the client's perspective. These two perspectives are like the two faces of the same coin: therefore, they cannot be addressed separately in a refactoring process.

Next we will provide some advice on how to approach the problem of coupling using the aforementioned *detection strategies*.

1. For each operation affected by Intensive Coupling group the invoked methods by their definition class. There will be one or more such groups containing 3 or more methods from one single class.
2. After that, collect the groups of "3-or-more-methods" (from the same provider class) from all operations affected by Intensive Coupling, and try to identify common groups. For those groups of methods that are used together in several client operations, try to introduce a new method in the provider class, and replace the multiple calls with a single call of the new method. Such a refactoring could have multiple beneficial consequences:
 - If these groups contain methods affected by Shotgun Surgery (and they usually do) the refactoring would reduce the number of clients for these methods, and thus reduce their incoming coupling. In many cases such a refactoring would help them recover from the Shotgun Surgery(133) disharmony.

- Provider classes for groups of “3-or-more-methods” are often lightweight classes, i.e., they do not have very much functionality, sometimes even being reported as being a Data Class; additionally, such classes may participate in violations of the Law of Demeter. Such refactorings would move some of the functionality to them, thus improving also the identity harmony.
3. Dispersed Coupling is a design problem that oftentimes affects Brain Method(92), because of the following reason: An excessively large and complex operation is almost always non-cohesive, doing more than one thing; and therefore there will be many invocations to methods from many classes. Thus, methods affected by Dispersed Coupling should be first checked to see whether they are also detected as Brain Method. If so, the Brain Method problem should be solved first, as this might eliminate as well the Dispersed Coupling.
 4. Assume now that we consider the Brain Method problem as solved, but there are still methods affected by Dispersed Coupling. In this case, collect the groups of invoked methods from all the operations affected by Dispersed Coupling. Look at these groups trying to identify *clusters of methods* invoked from multiple client operations (affected by Dispersed Coupling(127)). For each such cluster, check if this is not an invocation chain, thus violating the Law of Demeter [LR89], and try to remove it [DDN02]. After all, you should check for such invocation chains in all methods affected by Dispersed Coupling, as in our experience these violations of the Law of Demeter are (apart from the operation being a Brain Method(92)) the primary cause of this disharmony.

As a final remark, note that if the aforementioned *detection strategies* will not flag anymore certain classes or methods as being affected by Shotgun Surgery— as a result of applying the proposed refactorings — it does not necessarily mean that you will be safe. We consider the proposed refactoring as a first step towards often more challenging issues. Rationalizing the communication between classes is a good approach to better understand deeper problems. As we already pointed out, changing the coupling between classes is not simple, since one dependency may still force you to load a complete package and moving dependencies around is often not as trivial as it seems. Some solutions may lead you to rethink totally the flow of the communication within your application or to introduce dynamic mechanisms to deal with the temporal aspects of the dependencies [DDN02].