Chapter 12

Cluster: Duplicated Code

DETECTION OF DUPLICATED CODE

Author(s): Matthias Rieger and Stéphane Ducasse

Intent

Detect code duplication in a system, without prior knowledge of the code. Identifying the duplicated code is a first important step towards application refactoring.

Applicability

The only prerequisite is the availability of the source code.

Symptoms.

- You already saw the same source somewhere else in the application.
- You already fixed the same error in another piece of code.
- You make a conceptual change and in adapting the software to the new concept have to edit similar pieces of code over and over again.
- You know you employed copy and paste programming during development, but do not remember exactly were it was.

Reengineering Goals. Some of the following reengineering goals are not only linked to the identification of duplicated code but also to its removal by refactoring:

Identifying unknown duplicated code. This pattern is well-suited to identify **unknown** and middle size (4 to 100 lines) of duplicated code. If you are looking for occurrences of a particular line of code, use sed- or grep-like tools or emacs (regexp and etag) facilities. If you are sure that the developers had to use copy and paste coding (e.g. your software contains about 4 millions lines of code and was developed by 2 people during one year) but want to know what has been copied and pasted, apply this pattern.

Identifying duplicated code in large scale system. Following the previous point, if you are looking for a way to identify duplicated code in a big (100'000 lines) to huge system apply this pattern.

Improving maintenance. Detection helps the maintainer of a system to make sure that some code fragment, where an error has been fixed, is not copied a number of times with the error still in it, or, complicating matters further, is fixed differently at each location by maintainers who have no knowledge of each other's activities.

Reducing maintenance cost. By detecting clones of a piece of code to be maintained and merging the code into one instance, the multiplied effort otherwise necessary to maintain all the clone instances is removed.

Improving the code readability. By identifying duplicated code and refactoring it, the size of code is reduced. The level of abstraction is elevated when similar code pieces are refactored in a new method, ultimately leading to the SMALLTALK ideal of 6 lines of code per method. In one of the FAMOOS case studies, we found a method of 6000 lines of C++ code, which is a nightmare in complexity by any standards.

Improving compilation time. The less lines of code you have, the faster your system is compiled.

Reducing the footprint of the application. The less lines of code you have, the smaller the executable of your application gets.

Related Reengineering Patterns.

- The CUT AND PASTE anti-pattern [BROW 98] explains what practices lead to code duplication. The pattern discussed here focuses on the *detection* of the duplicated code.
- Patterns describing the factoring and reorganisation of code within the class hierarchy or by creating new classes. Such patterns detail how the detected clones can be merged into a single instance.

Motivation

The duplication of code occurs frequently during the development phase when programmers reuse tried and tested code in a new context, but are reluctant or, due to severe time pressure unable, to invest the time necessary to generalise the existing code to be used in the old and the new context. Since duplication is an *ad hoc*/copy&paste activity more than something that is planned for, occurrences of duplication are not documented and have to be detected.

Process

In order to detect code duplication in an unknown system, one cannot search for specific patterns. Rather, the self similarity of the system has to be discovered. Each copy is equal or similar to its clones and this similarity is revealed by comparing the entire system to itself. This comparison is on the one hand computing intensive and on the other hand produces a remarkable amount of data of possibly copied code pieces. It is therefore necessary to automatically narrow down the candidates that have to be examined in detail by a human.

Recipe. The applicability of the recipe is based on the availability of a tool for duplication detection.

1. Start with an automatic search for clones. The tool should create a database of all locations where code duplication possibly occurred.
2. Deciding on the level or size of duplication that is interesting, filters are defined that remove the uninteresting candidates.
3. For each clone family (i.e. $n \geq 2$ copies of the same piece of code) that is left after the filtering step, a list of source code locations, possibly already with citations of the offending code pieces, is presented to the maintainer so s/he can decide on how to remove the duplication.

Note that the recipe in this pattern does not concern itself with the actual problems of refactoring the code.

Difficulties. The approaches used to compare actual pieces of code work on syntactical representations. Therefore, one cannot detect duplicated *functionality* that does not bear any syntactical resemblance.

Language Specific Issues.

Language dependency can stem from the parser that transforms the source code in the format that is used for comparisons by the tool. Depending on this format, the parser can be of variable complexity. For example, comparing the source code as text with only minimal transformations, e.g. removing comments and superfluous white space, only needs a very simple lexer, which keeps language dependency at a low level. Comparing abstract syntax tree of the source code, however, requires a full blown parser. The complexity of the first transformation step thus correlates directly with language dependency.

Tools

Tool support is vital for applying this pattern.

- We have implemented a SMALLTALK tool called DUPLOC (see Chapter 17), which is specifically aimed at supporting a visual approach of code duplication detection. At the moment, the tool uses textual comparisons only. It allows the user to compare source code file by file, enabling him to examine the source code by clicking on the dots. Noise filtering can be done by removing uninteresting lines.
- DUP [BAKE 92] is a tool that detects parameterised matches and generates reports on the found matches. It can also generate scatter-plots of found matches.
- DOTPLOT [HELF 95] is a tool for displaying large scatter-plots. It has been used to compare source code, but also filenames and literary texts.
- DATRIX [MAYR 96b] is a tool that finds similar functions by comparing vectors of source metrics.

Discussion

This pattern is valuable to apply if your system has the symptoms identified above or if your reengineering goals belong to the set of the mentioned reengineering goals. It is also advisable, though, to apply it as a precautionary measure in the maintenance process as a *code investment* [BROW 98]. If you plan to revamp an old system, duplication detection can help to plan parts of the effort.

Moreover, if your system should be migrated from one paradigm to another one—e.g. from COBOL to an object oriented language like SMALLTALK —and you suspect duplicated code, this pattern is valuable to identify which parts of the old system have been duplicated. Assessing the similarities and differences of the parts will also improve your understanding of the systems functionalities.

The approach that has been taken in the development of DUPLOC (Chapter 17) has the following advantages:

- It is lightweight: it does not use complicated algorithms like elaborate parsing techniques.
- It is visual: the human eye is built to detect configurations and this can be fully exploited with a matrix visual representation.
- It is language independent: Since we use textual comparison, the tool is language independent to a high degree and can be used for a number of languages without a change.

Technical. The algorithm that is used to compare the source lines determines what level of fuzziness is allowed to recognise a match. The simplest algorithm—which compares the source lines character per character—finds only exact matches. More complicated algorithms (see for example [BAKE 95]) can find *parameterised matches*. Parameterised matches point out the possibility to refactor code into a parametrisable function, where exact matches emphasise more the repetitive structures in the source code.

Known Uses

The pattern has been applied in biology research to detect DNA sequences [PUST 82]. In the context of software reengineering, the pattern has been applied to detect duplicated code in FAMOOS case-studies containing up to 1'000'000 lines of C++. It also has been applied to detect duplicated code in a COBOL system of 4 millions of line of code. The DUP tool [BAKE 92] has been used to investigated the source code of the X-Window system, and DATRIX has investigated multiple versions of a large telecommunications system, wading through 89 million lines of code all in all [LAGU 97]. DOTPLOT [HELF 95] has been used to detect similarities in man-files, literary texts and names from file systems.

Chapter 13

Cluster: Improving Flexibility

REPAIRING A BROKEN ARCHITECTURE

Author(s): Holger Bär and Oliver Ciupke

Intent

Detect and remove dependencies between packages of a system that aren't allowed according to the designated system architecture. These architecture breaking dependencies may prohibit the exploitation of the architecture's advantages and cause unexpected effects at maintenance work.

Applicability

This pattern is only applicable if the system to re-engineer should suit a certain architecture like *Model View Controller* (MVC), should be layered or should obey other documented restrictions concerning the dependencies between its packages.

To clarify the further discussion we note that a dependency between two classes located in different packages implies a dependency between the corresponding packages with the same direction.

Symptoms.

- If you are to carry out a change on the system which is supported by the system's documented architecture, e. g. replacing the top level package in a layered architecture or adding a view to the model in a MVC architecture, the effort is higher than expected. This is due to extra dependencies breaking the architecture and resulting in a cascade of changes to the rest of the system.
- Analyzing the system one encounters forbidden dependencies between packages, e. g. model classes depending on their visual representation in an MVC architecture.

Reengineering Goals.

- If the conformance to a certain architecture is proven or recovered the benefits of the architecture can be exploited, e. g. it's easy to add a new view to a model within the MVC architecture.
- Understandability: the dependency constraints of the architecture reduce the number of dependencies.

Motivation

The following example describes a typical three tier architecture for business applications with a user interface, application logic layer and database layer. The architectural restriction on the dependencies between these three packages is that the user interface may depend on the application logic which may depend on the database, but nothing more.

Initial Situation. In our example the application logic layer implements financial transaction management and offers a service named reportTransactions (from, to) to report the transactions for a certain period of time. Figure 13.1 shows the three packages and their dependencies. Evidently there's a dependency breaking the architecture from the application logic layer to the user interface: the call new ListOutput (reportList) to create a new output window for lists offered by the user interface layer. The reason for introducing this call instead of returning the resulting list to the user interface layer might be the fear of performance penalties.

In general reasons leading to a broken architecture are:

- Altering the system without having understood the architecture.
- A system architecture which seems to have performance penalties that can be overcome by breaking the architecture.
- Favoring "quick-and-dirty" instead of "nice-and-clean".

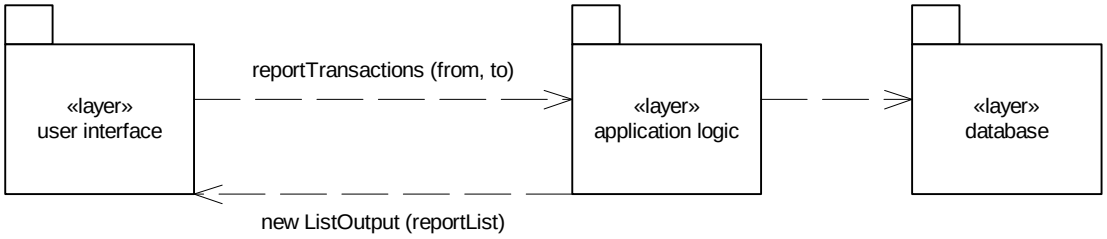


Figure 13.1: A broken architecture

Final Situation. The solution for the problem described above is quite straightforward. Just let the service computing the report return the transaction list to the user interface instead of itself displaying the list. But first it's not always that easy and second it's only easy after the dependency breaking the architecture has been found within the whole system.

Structure

There is no common problem structure for breaking an architecture because architectures themselves do not have a common structure. So a target structure is missing also.

Process

Detection.

1. Analyze the actual high level dependency structure, i.e. the dependencies between packages.
2. Search for dependencies which are not allowed by the documented architecture. You can do this

   Manually: Visualize the dependency graph with a graph layout tool and search manually for dependencies breaking the architecture.

   Automatically: The process can be automated with a tool that is able to analyze and manipulate graphs or relational queries on the given data.

(a) Set up a second graph containing the packages and the allowed dependencies between them according to the documented architecture.

(b) Compute the set of actual dependencies minus the set of allowed dependencies. The result are the dependencies breaking the architecture.

3. To find architecture violations in a system that should be layered search for cycles in the dependency graph.

Recipe.

A violating dependency exists either between two packages where no dependency is allowed or the dependency is just in the wrong direction like the one in our motivating example. In the first case there is no general solution, but in the second case the dependency can be reversed in a generic way:

1. Create a new abstract class with the same interface as the target of the dependency.
2. Replace the dependency on the target class by one on the new abstract class.
3. Let the target of the dependency inherit from the new abstract class. Now both the original source and target of the dependency are dependent on the abstract class.
4. Move the abstract class to a package where both the source and the target package may depend on it. In the case of reversing the dependency, this is the package containing the source of the dependency.

Figure 13.2 shows a broken architecture. Package P2 may depend on package P1, but P1 is not supposed to depend on P2. Actually class B depends both on classes C and D which makes P1 dependent on P2.

Figure 13.3 shows the solution to the problem. Instead of C and D, B calls now the abstract classes C_abstr and D_abstr. C and D inherit from their abstract counterpart and implement the methods called by B.

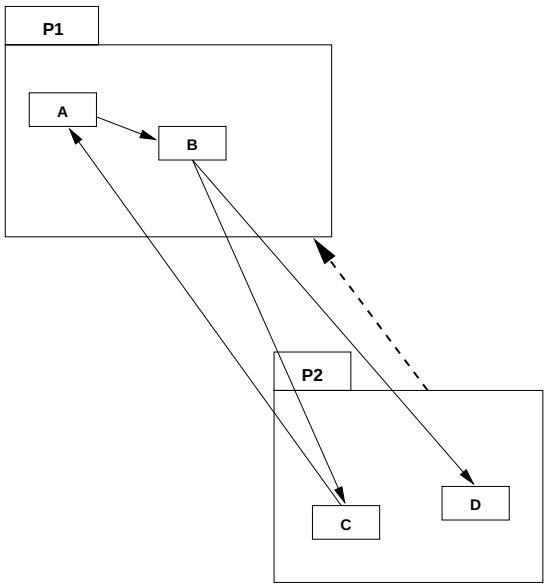


Figure 13.2: A broken architecture

There are also special solutions for similar problems like a model that has to update its various views without knowing how many views there are and of which type they are. This problem is solved by the OBSERVER pattern [GAMM 95].

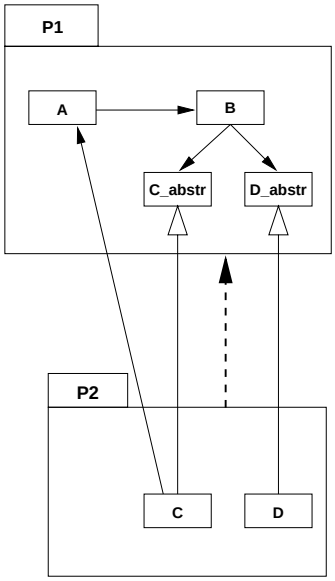


Figure 13.3: Dependencies reverted to fit the documented architecture

Difficulties. The new abstract class used by the source of the forbidden dependency B can be seen as an interface defined by B that supplier classes must implement. So this interface needn't contain all methods of the former target (C resp. D), but only the methods B needs.

In our example in the figures 13.2 and 13.3 there was only one class, class B, having a forbidden dependency on class C. If there is more than one class having a forbidden dependency the solution is a bit more complicated:

- Define only one interface per package that is used by all dependants on C.
- For dependants in different packages create one abstract class per package and let class C implement all of them or move the abstract class(es) into a package which every affected package may depend on.

Language Specific Issues. The transformation for reversing dependencies is only generally applicable for languages that allow an inheritance relation to be added to a pure abstract class (interface in JAVA). This is the case for C++ and JAVA and also for SMALLTALK , because in SMALLTALK there is no need to inherit from a pure abstract class.

Tools

Tools support is available for the following tasks of the detection section:

- Produce static structure graphs from source code.
  The tool set GOOSE contains too parsers, RETRIEVER and TABLEGEN with different advantages which can generate design information from C++ code in a format readable further tools.
- Visualizing a graph.
  You can use VCG, a graph layout tool, with a great variety of hierarchical layouts or Graphlet offering a set of layout algorithms with quite different aproaches. Unfortunately Graphlet (Version 2.8) has problems with printing the graphs.

- Setting up a new graph
  can be done with a graph editor like Graphlet.
- Finding cycles in a graph.
  Execute the command

  ```
  reView strongComponents < graph.gml | printCycles
  ```

  of the tool set GOOSE with graph.gml replaced by your graph file.
- Computing the difference between two graphs.
  The tool set GOOSE lets you convert graphs in a relational ASCII format. Filter off any other information besides first three collumns containing the type, source and target of the relation in these ASCII files with the Unix command cut -f1-3. Then use the Unix command comm -23 followed by the two files on which the difference should be computed.

Known Uses

In one of the FAMOOS case studies, there was an architecture defined with a base line framework and different products on top of this framework. When analysing the code, a class of the base line framework was found to inherit from several product classes. This kind of dependency was forbidden by the architecture definition. To repair this, an interface class was introduced from which the product classes inherited. This way, the framework was no longer dependent on the products, which made the system easier to change and decreased compile times.

A further example for an successful architecture clean up is the change in the event model of the Java Development Kit from the Version 1.0 to 1.1 (...cite). In this case the OBSERVER pattern was applied[1]. The observer pattern [GAMM 95] is a special form of the general principle to introduce an abstract interface to decouple classes.

[1] In the JDK, the Observer is called Listener.

TRANSFORMING INHERITANCE INTO COMPOSITION

Author: Benedikt Schulz

Intent

Improve the flexibility and comprehensibility of your design by transforming an inheritance relationship into a component relationship and delegating of a set of methods to this component.

Motivation

The following example occurred in a project which aimed at visualising hydraulic data of river parts. The data was visualised in a two-dimensional diagram which changed over time. The user of the system got the impression of seeing a film because of this animation.

The most crucial part in the system concerning efficiency was the subsystem which was responsible for drawing lines on the screen: For every new frame of the animation the complete set of lines representing the data had to be redrawn.

Initial Situation. In the first version of the system drawing lines was handled by the GDI subsystem of the Win32s operating system. This was pretty efficient until a new requirement came into play. The customers wanted to be able to change properties of the lines like colour, thickness, style, etc. The GDI subsystem was not able to draw lines with customisable thickness in an efficient way however: The system was showing rather a slide show than a film. The initial design is depicted in Figure 13.4.

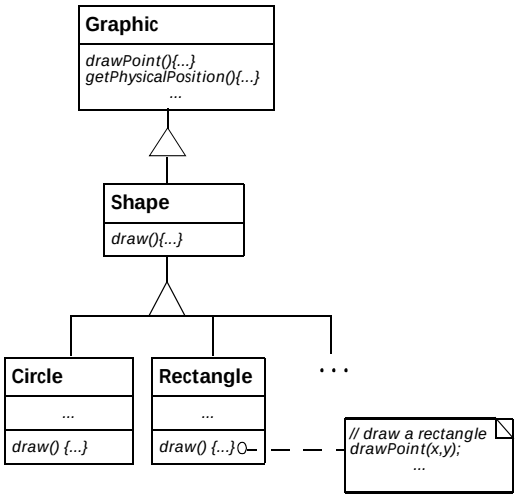


Figure 13.4: Initial Situation

Some experiments with a new technology called DirectDraw (that is also a subsystem of the operating system) revealed its superiority and thus the project manager decided to replace GDI with DirectDraw.

This led to serious problems: Since the class responsible for drawing lines was using functionality of GDI by inheritance it was not possible just to replace it by DirectDraw. DirectDraw had a different interface and so the implementation of a lot of methods which were responsible for drawing lines had to be changed.

Final Situation. To avoid similar problems in the future the project manager decided not only to change the the drawing system but additionally to introduce a flexible new design which should allow for easy exchange of different drawing systems.

The new design got its flexibility mainly from one change: Instead of relying on inheritance to reuse functionality, a component relationship together with the concept of delegation was used. This means that a `Shape`-object no longer ”knows” (directly or via inheritance) how to draw points but it rather ”knows” an object which ”knows” how to draw the points. Since objects can even be changed during run-time of the system the flexibility of the system was significantly improved. The final design is depicted in Figure 13.5 where new or changed entities are marked grey.

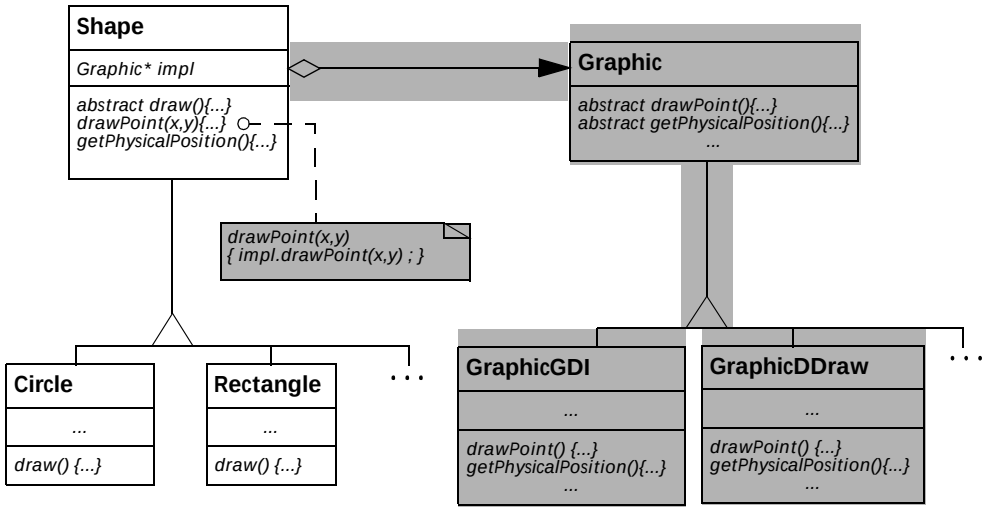


Figure 13.5: Target Structure

Some weeks after the redesign of the system it was revealed that the DirectDraw subsystem was not automatically installed on all systems running Win32s. But since the system could check whether DirectDraw was installed or not during run-time and since the drawing system was made exchangeable during run-time this new fact did not lead to any problems.

In the end the target structure is an instance of the Bridge design pattern [GAMM 95]. (It was not possible to use a Singleton Graphic acting as a facade to the libraries, because Graphic is not stateless and can have different states for different Shape-objects.) The Transforming Inheritance into Composition pattern is nevertheless not equivalent to the Bridge design pattern, because it not only describes “good” target structures but rather the process of applying the Bridge design pattern to an existing object-oriented legacy system.

Applicability

Transforming Inheritance into Compositionis applicable whenever you recognise during the reviewing of your legacy system that *you should have used* one of the following design patterns *but you have not used* them:

- Bridge [GAMM 95],
- Strategy [GAMM 95] or
- State [GAMM 95][DYSO 96].

All of these design patterns make use of the Objectifier design pattern [ZIMM 95] and the technique of delegation.

The application of this pattern is difficult if the inheritance relationship is deeply nested in the hierarchy because breaking the hierarchy means that all the methods which were inherited (and this can be a large number) have to be delegated. Therefore the inheritance relationship is *not* removed in a variant of the Transforming Inheritance into Composition reengineering pattern which will be discussed later.

The reengineering pattern should not be used in the following cases:

- Inheritance *is* the appropriate modelling technique for the problem (e.g., if there is a *is-a* relationship between two classes).
- Introducing delegation would be too expensive with respect to efficiency. This has to be considered especially when the delegation takes place within a loop which is processed a lot of times.
- In statically typed languages: Clients use the two classes related via inheritance polymorphically and you do not want to change these clients.

Symptoms. The application of this pattern can improve your design if you encounter one of the following problems:

- For a certain problem you should have used the *Bridge*, *Strategy* or *State* design pattern but in the system you are reengineering these design patterns have not been used. You know how to use the respective design patterns when you are building a new system but you do not know how to apply them to an existing design.
  - You want to be able to change the implementation of an abstraction in a more flexible way, maybe even at run-time (*Bridge* design pattern). The actual design does not allow for this kind of flexibility.
  - You want to extend the class system with new classes which share the same interface but differ in their behaviour (*Strategy* design pattern). The actual design does not allow for this kind of flexibility.
  - You have a lot of conditional statements in your code because the behaviour of an object depends strongly on its current state. You want to get rid of these conditionals (*State* design pattern).
- The inheritance relationship was established mainly for code reuse. The code which was the reason for using inheritance now has to be changed and so you want to remove the inheritance relationship because it is no longer appropriate. You do not know how to do this without changing the functionality of the system.

Reengineering Goals. The goal of the Transforming Inheritance into Compositionreengineering pattern is to help software engineers to apply a design pattern relying on the Objectifier design pattern and delegation to an existing design. In particular the pattern aims at

- increasing run-time flexibility. This is achieved because after the application of the reengineering pattern you will be able to change the component during run-time.
- increasing static flexibility (configurability). This is achieved because after the application of the pattern you will be able extend the component class hierarchy independently from the abstraction.
- increasing comprehensibility. This is achieved because the reengineering pattern can remove inheritance for code reuse which is hard to understand from your system.

Related Reengineering Patterns. The Transforming Inheritance into Compositionreengineering pattern is related to all design patterns which rely on the Objectifier design pattern [ZIMM 95] and delegation like

- Bridge
- Strategy
- State

Structure

The problem structure is depicted in Figure 13.6. Transforming Inheritance into Compositionleads you to the target structure depicted in Figure 13.7

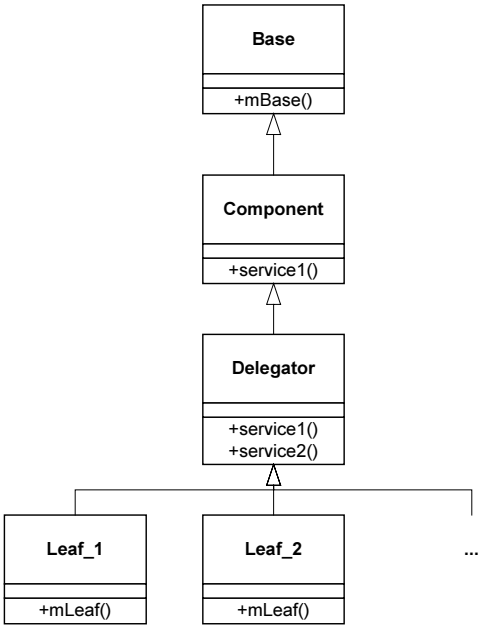


Figure 13.6: Problem Structure for the reengineering pattern

Participants.

- **Base** is the root of the inheritance tree.
- **Component** (Graphic) is the class which gets cut out from the inheritance hierarchy to serve as a provider of certain services. The inheritance relationship to `Base` may remain in existence.
- **Delegator** (Shape) is the class which uses services from `Component` by inheritance in Figure 13.6. After application of the reengineering pattern in Figure 13.7 `Delegator` will make use of these services by delegation.
- **Leaf_1, Leaf_2, ...** (Circle, Rectangle, ...) are the leaves of the inheritance hierarchy
- **Component_A, Component_B, ...** (GraphicGDI, GraphicDDraw, ...) are the subclasses of `Component` implementing the services of their super-class in different ways..

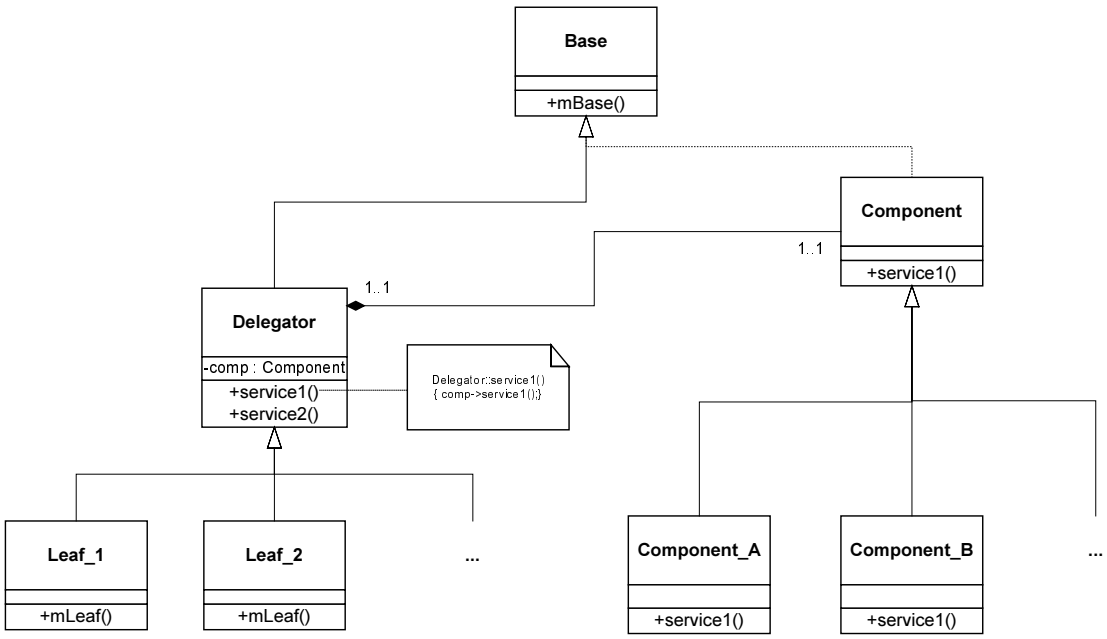


Figure 13.7: Target Structure for the reengineering pattern

Collaborations.

- `Delegator` makes use of **service1** (drawPoint) provided by `Component`. This is done
  - in the problem structure by executing inherited methods from `Component` whereas
  - in the target structure the execution of these methods is *delegated* to `Component`.

Consequences.

- Positive benefits
  - Transforming Inheritance into Compositionsolves an important and basic reengineering problem and the application of the reengineering pattern allows for the introduction of several known design patterns [GAMM 95].
  - Since abstraction and implementation are separated, changing the implementation does not require recompilation but only rebinding of the system.
  - The implementors of `service1` can be designed to form a separate inheritance tree. (This is suggested by the class `ComponentA` in Figure 13.7.) This is impossible before the application of the reengineering pattern.
- Negative liabilities
  - The execution of `service1` provided by `Delegator` will take longer in the target structure because it has to be delegated. This may be critical if `service1` is needed a lot of times.
  - The target structure is slightly more difficult to implement since the attribute of `Delegator` named `comp` has to be initialised whenever a new instance of `Delegator` is created and destroyed whenever that instance is deleted.

Process

The process mainly relies on the idea of combining the approach of considering design patterns as operators [ZIMM 97] (rather than building blocks) and the refactoring approach presented in [JOHN 93]. This idea is presented and discussed in detail in [SCHU 98b].

Detection.

Since violations against flexibility issues can only be detected if you know where flexibility is needed and which kind of flexibility (e.g., run-time flexibility, configurability) is needed, algorithmic detection is difficult. However, you can

- ask people who designed and implemented the system if there is a case where they wanted to be able to change the implementation of an interface at run-time and this was not possible.
- look for methods with a large amount of conditional statements. The behaviour of an object may depend strongly on its internal state (Type Check Elimination within a Provider Hierarchy).
- look for two classes, one inheriting from the other, which are never used polymorphically. This means that a variable declared as super-class is never used for an instance of the subclass.

Recipe. In this section we show how to apply tTransforming Inheritance into Composition and what kind of reengineering operations have to be applied. If we name entities (like classes, methods and attributes) we refer to the participants of the problem structure depicted in Figure 13.6 and the target structure depicted in Figure 13.7.

1. Create a new attribute `comp` of `Component` in the class `Delegator`. Change the constructor method of `Delegator` so that it initialises the attribute `comp` with a new instance of `Component`. If you plan to add several subclasses of `Component` later on (you should do so!) than add a new formal argument to the constructor method of `Delegator` which will serve as an indicator of which concrete subclass of `Component` to use.
2. Copy all the signatures of the methods from `Component` which are visible to `Delegator` to `Delegator`. For each added method add an implementation which delegates the execution of the method to the corresponding method of `Component`. For an example, see the implementation of `Delegator:service1()` in Figure 13.7.
3. Remove the inheritance relationship between `Component` and `Delegator`. Caution: In statically typed languages you will not be able to use an instance of `Delegator` polymorphically as an instance of `Component` after this step. In particular it is not possible any more to cast instances of `Delegator` to `Component`.

Difficulties.

If you decide to introduce an additional formal parameter to the constructor of `Delegator` then every piece of code that creates an instance of `Delegator` has to be changed. In languages which support default values for formal parameters this problem can be resolved by defining an appropriate default value (e.g., `Component` if this class is not made abstract).

If there is no way to avoid polymorphism between `Delegator` and `Component` but you still have strong reasons to apply Transforming Inheritance into Composition and you are using a statically typed language, you can omit removing the inheritance relationship between `Component` and `Delegator`. You should be aware of the fact, that you might have the following problem: The class `Component` has two parts:

- One part of the methods represents set of utility services. You made `Delegator` inherit from `Component` because you wanted to be able to use these services without re-implementing them.
- The other part of the methods represents the *real* interface of `Delegator`. You made `Delegator` inherit from `Component` because you wanted to establish an *is-a* relationship between `Delegator` and `Component` to be able to use instances of both classes polymorphically.

In this case consider splitting the `Component` class into two separate classes.

Language Specific Issues.

- In C++ you should implement the attribute `comp` as a pointer. Otherwise you will not be able to use polymorphism for the inheritance tree with root `Component`.
- In dynamically typed languages like SMALLTALK it is not necessary that two classes are related via an inheritance link to use them polymorphically. This means, for example, that you can still use instances of `Component` and `Delegator` together in one container object.

Discussion

Since the detection of the problem structure is far away from being an algorithmic, tool supported process, you should not explicitly look for this problem structure. But since software development is an iterative process you will find the problem structure while trying to extend or modify your system. Once you have found the problem structure in your code, you should strongly consider the application of Transforming Inheritance into Composition.

The relevance of this reengineering pattern is high: In a lot of companies which were early adopters of the object-oriented paradigm, the maturity of the software engineers concerning object-oriented technology was low. This resulted in an overuse of inheritance, mainly for code reuse. These software defects can be removed by the application of the reengineering pattern.

The concept of delegation and the Objectifier design pattern [ZIMM 95] are the fundamentals of this reengineering pattern and the resulting target structure is closely related to the Bridge, Strategy and State design patterns [GAMM 95]. A good understanding of these design patterns helps to use the reengineering pattern.

Tools

The detection of pairs of classes which are never used polymorphically can be done with the tool-set *Goose* [BÄR 98][CIUP 99]. *Goose* can not only detect missing polymorphism but a lot of other design defects which occur in object-oriented systems.

Since the application of the reengineering pattern relies on the application of refactorings [OPDY 92] you can use every tool which supports this technique, such as the *Refactoring Browser* [ROBE 97a] for SMALLTALK , which is the most advanced refactoring tool. The Refactoring Browser is described and available for free at `http://st-www.cs.uiuc.edu/~brant/Refactory/`.

For a subset of C++ we implemented a prototype to support refactorings. This tool is called *RefaC++* and described in [MOHR 98]. *RefaC++* can perform a subset of the refactorings presented in [OPDY 92] and can also apply the Bridge design pattern automatically.

Known Uses

Transforming Inheritance into Compositionhas been applied in the following known cases:

- The reengineering pattern was applied with success in the project described in the motivation section. It was possible to increase the flexibility of the system so that the new requirement (DirectDraw not available on every Win32s installation) could be fulfilled without problems.
- We are analysing and flexibilising a graphical information system for a German middle-sized enterprise. We found several design flaws which have been corrected by applying this reengineering pattern.
- [ROBE 96] describes how frameworks evolve. In the White-box Framework design pattern [ROBE 96] the engineer is encouraged to use inheritance for reuse because it is easier to understand and to reuse. In later stages of the framework development inheritance has to be replaced by polymorphic composition.

DISTRIBUTE RESPONSIBILITIES

Author(s): Holger Bär and Oliver Ciupke

Intent

Distribute the responsibilities equally among the classes of an object-oriented system to prevent large, hardly maintainable and reusable classes.

Applicability

A responsibility is a description of a service offered by a class. It is fulfilled by a set of publicly accessible methods.

Symptoms. If the responsibilities aren't distributed among the classes, there will be one or more classes incorporating a lot of responsibilities. Such classes, called *multiple responsible classes* (MRC) from now on, result in the following symptoms.

- If you ask for the responsibilities of a MRC, you get long and unclear answers.
- The MRC is used by other classes for different purposes (low level MRC).
- The MRC uses a lot of classes (high level MRC or *manager class*).
- Functional enhancements somewhere in the system often require changes in one of the high level MRCs.
- A MRC is mostly large in lines of code and number of methods, because many responsibilities result in many methods resulting in many lines of code for concrete classes.
- High level MRCs can hardly be reused because too many design decisions of the specific application are coded into them.
- Maintenance work on MRCs is hard, because there is no boundary between the different responsibilities, so that it's unclear where to change the class for a certain maintenance action and which the effects of the change are.

Reengineering Goals.

- Understandability: classes with many responsibilities are hard to understand, because the responsibilities are mixed together, i. e one can't identify the individual responsibilities and understand their implementation and collaboration with other classes in isolation.
- Flexibility: classes with few responsibilities allow fine grained adoptions by subclassing or replacing a class.
- Reusability: classes are normally reused as a whole. Therefore it's unlikely that MRCs are reused because the particular combination of responsibilities needed in its original application is unlikely to occur in another one.

Related Reengineering Patterns.

- large classes
- large methods
- structural programming
  Classic structural programming principles applied to an OO-language often lead to one central manager class operating on several dumb data classes.

Motivation

Initial Situation. The UML diagram in Figure 13.8 shows a manager class, AccountManager, together with two passive classes, AccountData and BarChart. The responsibilities of the manager class are

1. to process user input in OnCalculateSummary,
2. to do the summary calculation using Get...Transaction methods to query the transaction data from class AccountData,
3. and to present the results with the help of class Screen.

So the manager class has three responsibilities and implements nearly the whole functionality. Major design decisions like how the summary is calculated and presented and the reactions on the user input are hard coded in this class thus making it hardly reusable.

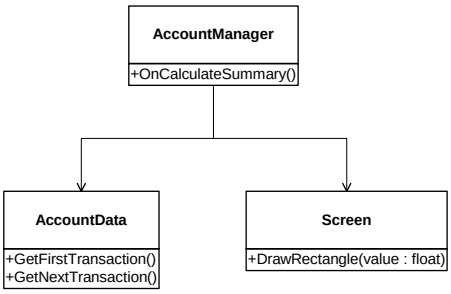


Figure 13.8: Example of a manager class with two passive classes.

Final Situation. We can distribute the three responsibilities of the manager class among three classes: UserInteraction, Account and BarChart. In this design all classes besides UserInteraction have a high potential for reuse.

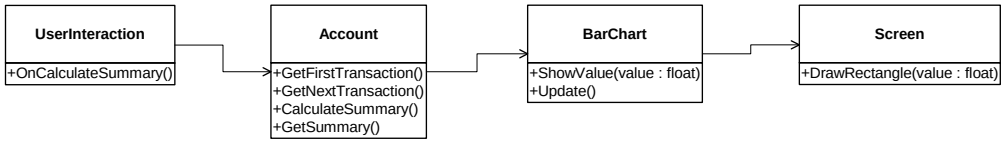


Figure 13.9: The improved example with distributed responsibilities.

Structure

The structure of the problem and the target structure differ both between high level MRCs and low level MRCs.

High level MRC problem structure.

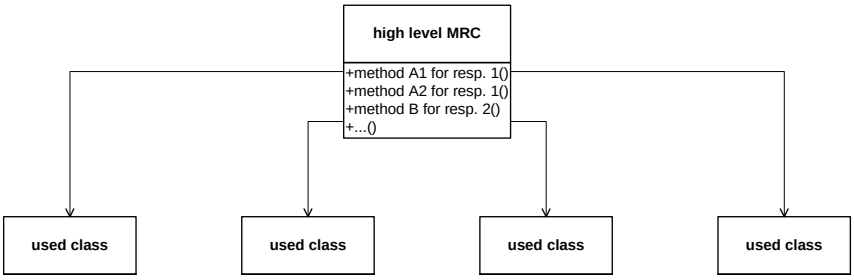


Figure 13.10: Problem structure of a high level MRC.

Participants. The high level MRC shows a broad interface with a set of methods per responsibility.

Collaborations. High level MRCs often use many other classes to fulfill their numerous responsibilities.

Low level MRC problem structure.

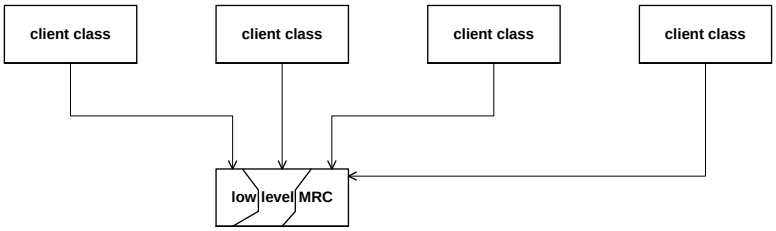


Figure 13.11: Problem structure of a low level MRC.

Collaborations. The various clients use different responsibilities of the low level MRC.

High level MRC target structure.

For high level MRCs there is no general target structure. The goal is to distribute the responsibilities. Good candidates for receiving responsibilities are the used classes. But sometimes it's necessary to define a new class like BarChart of the motivating example. The manager class itself will be reduced in size (lines of code, methods) or will disappear completely.

Low level MRC target structure.

Participants. The low level MRC has been split into several classes according to the responsibilities of the MRC and the parts used by the client classes.

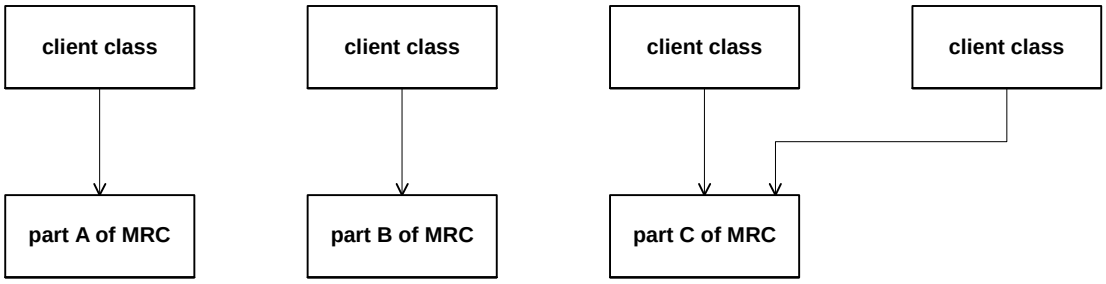


Figure 13.12: Target structure of a low level MRC.

Collaborations. The client classes of the low level MRC now only use those parts of the MRC they actually need.

Consequences.

- The responsibilities of the high level MRC are distributed moving the code closer to the data it works on.
- In case the manager class disappears completely there is no central control any more. The design has made a step towards autonomous interacting objects.
- The remainder of the high level MRC and the parts of the low level MRC are smaller, easier to understand and exhibit more potential for reuse.
- Instead of many dependencies on one low level MRC the application of this pattern leads to a set of classes each with a lower number of dependents.
- The smaller ”part classes” of the low level MRC are more stable than the original class simply because they encapsulate less design decisions. So together with the previous topic the compilation times after changes to the system will be reduced.
- In both cases the number of classes may increase.
- The distribution of responsibilities may affect the efficiency of the system.

Process

Detection.

- MRCs are normally the largest classes in a system, program or package both in lines of code and in number of methods.
- To find high level MRCs search for classes with manager, man, driver, initiator and so on in their name.
- Classes that use many other classes are also good candidates for high level MRCs. They can be found by looking for classes with high values for coupling metrics like the CBO metric [CHID 94].
- Low level MRCs are used for different purposes. So the implementation of their responsibilities are likely to not communicate with each other. Therefore these classes will often exhibit low cohesion. There are numerous cohesion metrics, e. g. the TCC metric [BIEM 95].

The conjunction of the size and coupling criteria and optionally the name criterion should produce satisfying results for the detection of high level MRCs as should the conjunction of the size and cohesion criteria for low level MRCs.

Recipe.

- Search for candidate MRCs as described in the detection section above. The next steps depend on whether you have detected a high level or a low level MRC.
- High level MRCs
  Try to distribute the responsibilities of the manager class to other classes. Good candidates for receiving responsibilities are the classes used by the manager class. It may be necessary to define a new class like BarChart of the motivating example. The manager class itself will be reduced in size (lines of code, methods) or will disappear completely.
- Low level MRCs
  1. Determine the parts of the low level MRC.
     There are two ways to determine the parts — one considers the use of the class by its clients, the other one examines the class' internal structure:
     (a) Analyse the use of the class by its clients. Note for each type of client the features of the MRC (methods and public attributes) it uses. Find a partition of the feature set so that each client uses only one or few parts.
     (b) Although cohesion metrics indicate whether a class could and should be split, they do not directly indicate where to split the class. You can get good suggestions for splitting by computing the minimum cut on the undirected graph containing all methods and attributes of the class as nodes and all method calls and variable accesses within the class as edges[2]. The minimum cut algorithm computes such a partition of the graph in two sets of nodes that the number of edges from one set to the other is minimised. Splitting a class according to this partition leads to two classes with minimal communication between them. Of course this splitting step can be applied to the two sets recursively until the resulting classes are small enough.
  2. You may use one of the mentioned partitions of the MRC or a combination of both to split the MRC. In cases where there is no optimal partition (e. g. client uses more than one part or there is more than zero communication between the parts) the partition often needs some manual fine tuning to end up with a set of reasonable classes.
  3. Split the MRC according to the partition and reorganise its context.

Difficulties.

- The detection of MRCs is not very precise, especially for low level MRCs. The key point in detecting a low level MRC is to recognise that it is used by different types of clients for different purposes.
- Determining the partition of the MRC can hardly be done fully automatically because the parts must be reasonable classes.

Tools

Several prototype tools have been developed within the FAMOOS project which can help to detect and to solve this problem.

[2] All edges have weight 1; parallel edges are allowed for modelling multiple calls and accesses

- A visualisation of the static system structure at the right level of abstraction, e.g. with the tools within the tool set GOOSE can help detecting central classes or subsystems.
- TABLEGEN computes the TCC cohesion metric and also coupling metrics.
- The computation of minimum cuts to determine the partition of a MRC can be done with REVIEW, a tool also developed within the FAMOOS project.
- GOOSE' relational representation of design information enables to search for classes with a high out-degree of usage. The classes with the highest out-degrees are good candidates for being high level MRCs.

Known Uses

During restructuring in one of the FAMOOS case studies, there was a big class found which incorporated responsibilities for several different products. This class was split into several pieces to make the program more flexible with respect to frequent changes [RITZ 98].

USE TYPE INFERENCE

Author: Markus Bauer

Intent

It is hard to understand the structure and the workings of a software system written in a dynamically typed language because of the lack of type declarations. Therefore add type annotations to the program code which document the system and which can additionally be used by sophisticated reengineering tools.

Applicability

Apply this pattern when reengineering systems that are written in *Smalltalk* or in a similar, dynamically typed programming language, where you have only limited knowledge about the system. Typical situations could be:

- You have to maintain and/or modify the software system, but you have only limited knowledge about its inner workings. You are interested to learn, which types of objects of the system are manipulated by some code your are working on, but this is difficult since you do not have type declarations in your source code that provide you with that information.
- You want to support a reengineering task by some tools, but these tools rely on type information for the system's variables and methods. Most reengineering tools rely on such type information. Examples include (but are not limited to) the *Smalltalk Refactoring Browser* [ROBE 97a][3] or tools that calculate software product metrics (like those described in [CHID 94]).
- You want to reengineer or rewrite the system using a statically typed programming language, but to achieve this, you need appropriate type declarations for the system's variables and methods.

Problem

In dynamically typed systems, the lack of static type information (i.e. the lack of type declarations for variables and method signatures) makes some reengineering tasks difficult or impossible, since such type information usually represents prominent parts of a system's semantics.

Motivation

Consider some code fragments[4] for a dynamically typed application that manipulates drawings. Such an application might have a class `Container` for storing some objects. Figure 13.13 shows a method `add` that is used to add objects to the container.

For reengineering purposes we might be interested in an answer to the following question: What kind of objects can be stored in the container, that is, of what types are the objects, that are passed as arguments to the `add`-Method?

[3] The current implementation of the Refactoring Browser does not infer precise types for the system's entities though, it relies on (unprecise) heuristics instead.

[4] We present code examples in a syntax close to Java. Since we deal with dynamically typed code, we just omit Java's type declarations.

```
add: anObject
   contents add: anObject.
   anObject draw.
   "..."
```

Figure 13.13: Method `add` in class `Container`.

Forces

- To learn about the types of objects that are manipulated by some code you are looking at, you might consider manually tracing the execution of your code and guess what's going on in your system, but for larger systems, this is an infeasible and error-prone task.
- You could also try to capture that information by looking at method and variable names, but in many legacy systems naming conventions do not exist or do not provide enough information about the object types and the manipulations that are made with them (see our example above). Even worse, you can't be sure that the names do not lead you to wrong conclusions.
- To migrate from a dynamically typed language to a statically typed language, you could apply approaches that do not rely on type information, like those proposed for the translation of Smalltalk applications to Java in [ENGE 98]. These approaches simulate Smalltalk's dynamic type system in Java. The resulting code, however, is not authentic Java code and is hard to understand and maintain. Such code additionally has the usual shortcomings of untyped code: it is not type safe.

Solution

Find out what types the variables and method parameters have and put this information into the source code, using type annotations or comments.

In more detail:

1. Perform a program analysis of your dynamically typed object oriented legacy system.
2. Use the results of the program analysis to determine type information for the program's variables, including global and local variables, parameters and return values of methods. Based on this type information, add type annotations to the program's source code.
3. Use these type annotations to understand how your legacy system works or as additional semantic information to more sophisticated reengineering tools.

This technique is called *type inference*, because you infer the type of an object at a certain place in the code by tracing its way from its creation to the current place.

If we can enrich the code of our example application with type annotations (see figure 13.14) by using the techniques described below we can easily find an answer to the question we asked above: Our `Container` holds points, lines, splines, ..., so it has obviously something to do with some geometrical shapes that make up a drawing.

We learn from this example that type annotations like those given in figure 13.14 make code much easier to understand and that they contain valuable information about the inner workings of a system.

```
add: anObject
   " {Container} × {Point, Line, Spline,...} → {} "
   contents add: anObject.
   anObject draw
   "..."
```

Figure 13.14: Method `add` annotated with type information.

Process

Type inference usually can't be done manually for reasonable large and complex applications. Therefore, we have to automate the task of computing type information for variables and method signatures.

To implement a tool or other means to get the information, we observe that during the runtime of the system, type information propagates through the system's expressions and statements: Upon creation, each object has a certain type assigned to it, and this type information is spread to all expressions and statements (including variable and method parameter expressions), that do some operations with the object. Thus, to infer types for the variables and methods of the system, we need to inspect object creations and the data flow through the system.

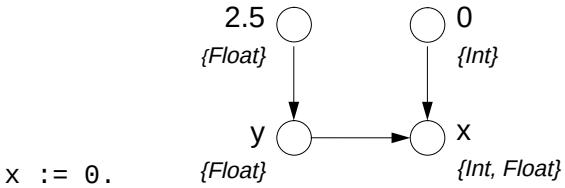
Basically we can do this in two ways: We either can execute the application and collect the type information we are interested in during its runtime *(dynamic type inference*) or we can use static program analysis techniques *(static type inference)* to analyse the applications source code and compute how the type information flows through the application's expressions. We will cover both approaches in some more detail below.

Dynamic type inference. With dynamic type inference, we modify the application or its runtime environment, to have it record the runtime type information for us.

1. Determine the most common execution paths through your program, that is, determine the most common usage scenarios of your legacy system. In some cases you might be able to use already existing testing scenarios for this. In other cases, determining these common usage scenarios might be difficult, especially if you don't know much about the system.
2. Instrument the code with instructions that record the data flow through your system and that collect the runtime types of the system's variables. [RAPI 98] describes how to modify the runtime libraries of a Smalltalk environment to achieve this with only minor changes to the application's code.
3. Run the system and have it execute the most common usage scenarios you collected in step 1.
4. Use the recorded runtime type information to put type annotations into the source code.

Static type inference. With static type inference, we need a tool that reads in the complete source code of the application and analyses it to construct a data flow graph. This is done by representing the application's expressions as nodes in the graph, and by modelling the dependencies between them as edges. The dependencies that are taken into account to construct the data flow graph are given by the following rules:

1. An *assignment* `var := expr` generates a data flow from the right hand side expression `expr` to the variable `var` on the left hand side.
2. A *variable access* generates a data flow from the variable being accessed to the surrounding expression.




```
x := 0.
y := 2.5.
x := y.
```

Figure 13.15: Data flow graph.

```
max: b
   (self < b)
      ifTrue: [^b]
      ifFalse: [^self]
```

Invocation of `max`:

```
x := 1 max: 2
```

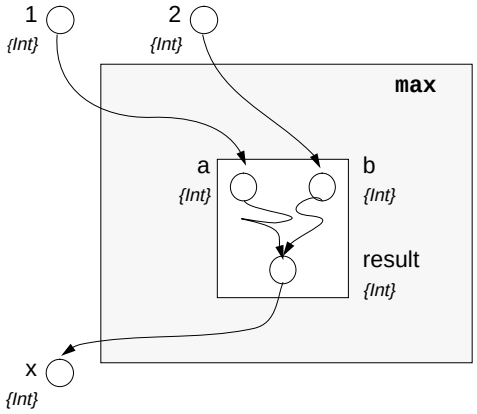


Figure 13.16: Data flow across method boundaries

3. A *method invocation* generates a data flow from the actual argument expressions to the formal arguments of the invoked method, and from the result of the invoked method to the invoking expression.

A data flow graph for a short piece of code is shown in figure 13.15.

For each node the tool then tries to compute the set of classes the corresponding expression can hold instances of. It starts by determining type information for the program's literal expressions and object creation statements (which are represented as source nodes in the graph) and moves that information along the edges through the graph. Each node then carries the union set of all type information of its predecessors. In figure 13.15, for example, the node for `x` carries the type information $\{Int, Float\}$, since it depends on the type information of the nodes for `y` and `0`.

Some subtle problems arise, whenever method invocations cause data flows across method boundaries (as given by rule 3). Such a case is shown in figure 13.16.

There are some well proven techniques to allow for an analysis which keeps track of these inter-method data flows in an efficient and practicable way. One of these is Agesen's Cross Product Algorithm [AGES 95][5].

[5]There are other algorithms that also allow the tracking of data flow across method boundaries, for example [PALS 91], [OXHØ 92], [PLEV 94], but Agesen's algorithm is superior to most of these, because it is easy to understand and computes precise type information in a very efficient way[AGES 94].

The basic idea is to create separate sub graphs for each method and link all those subgraphs together in an appropriate and efficient way.

After the graph has been complexly built up and all type information has been propagated through it, the type information associated with the graph's nodes can be used to annotate the source code of the application.

Discussion

A problem of using type inference to reveal some information about a legacy system arises from the fact that we analyse the data flow through an application. To make our approach work, we have to analyse the complete source code of an executable application (including libraries), or, if we are using dynamic type inference, we have to execute an adapted version of the system. This might be a problem in some cases when parts of the source code are not available and/or a runnable version of the system cannot be produced. Furthermore, frameworks and class libraries cannot be analysed without application code using or instantiating them. Then, however, the inferred types are only valid in the specialised context of the particular application.

Static type inference algorithms usually have to overcome some difficulties: static analysis is complex and the results are often unprecise. Agesen's static type inference algorithm, as sketched above, addresses these difficulties in an appropriate way[6]. However, since the algorithm is very complicated it is difficult to implement it in a correct way and produce a reliable tool out of it. This is an issue, if you can't use one of the already existing tools (see for example [LI 98]).

However, once a tool for performing such an analysis has been built, it can be used on other reengineering projects as well and then it quickly pays of its rather high development costs.

Dynamic type inference has serious limitations when being applied to larger systems: You have to ensure that the most important parts of the system are covered by the analysis in a sufficient way, which might not be feasible for larger systems if you do not have test cases or usage scenarios available.

Related Reengineering Patterns

Type annotations document the inner workings of a legacy system. We can therefore see type inference as a technique to improve your knowledge about the legacy system. Thus, this pattern relates with all other reengineering patterns that describe *reverse engineering techniques*, i.e. analyses of the source code of legacy systems to extract additional semantic information and improve the understanding of the systems.

Known Uses

ObjectShare has used type annotations (like those that can be computed by applying this pattern) to document large parts of the source code to the *Visualworks Smalltalk* environment. This emphasises that type annotations are of great help understanding source code.

The GOOSE tool set (and related tools) that support the reengineering of C++ applications by visualising software structures [CIUP 97], checking design heuristics [BÄR 98] and calculating software metrics [MARI 97] can analyse Smalltalk applications after type inference is used and the source code is enriched with type annotations.

The University of Stuttgart, Germany, has developed a tool called *Smalltalk Explorer* which is used to explore existing Smalltalk applications. It heavily relies on the type inference algorithm presented here.

[6] A detailed discussion of the algorithm, especially regarding complexity and precision can be found in [AGES 95] or in [BAUE 98].

Type annotations are used to allow for an easy navigation through unknown Smalltalk code by documenting which classes are manipulating which other classes and by introducing hyperlinks between them[LI 98].

The type inference algorithm is also used to facilitate a mostly automatic translation of dynamically typed Smalltalk applications into statically typed Java applications [BAUE 98]. Since most of Smalltalk's concepts can be mapped upon suitable Java concepts the most prominent issue is to infer appropriate static types for the resulting Java code. This is done by computing type annotations (as described above) and transforming them into type declarations. In more detail, to map a type annotation to a type declaration, a class must be found (or created by refactorings), that is a common abstraction to all classes included in the type annotation.