Transforming Inheritance into Composition A Reengineering Pattern[*]

Benedikt Schulz Thomas Genßler

27th September 1999

Abstract

Transforming Inherintance into Composition is a reengineering pattern describing a solution to a recurring reengineering problem. The problem is that transforming an inheritance relationship into a component relationship using delegation without affecting correctness and functionality of systems is a time-consuming and error-prone task. Solving this problem leads to more flexible and more comprehensible systems. The problem is recurring, because cases of misuse of inheritance are very common in almost every object-oriented system. Solving the problem is time-consuming and error-prone because the necessary changes possibly affect the whole system and thus cannot be performed locally.

Introduction

Object-oriented design patterns such as those presented in [GHJV95] describe a solution for a recurring design problem. They help software engineers in designing a system based on the requirements they discovered during the analysis phase. The right use of design pattern leads to a system which is flexible enough to allow for easy modification – sometimes even at run-time – of those parts of the system which were considered to be a matter of change in later versions of the system due to a change of the initial requirements.

The world is hostile, though: It is likely that requirements will change unexpectedly and in an unforeseeable way. This requires flexibility in parts of the system which was not foreseen and therefore these parts will have been designed in an inflexible way. This is where *reengineering patterns* come into play: Reengineering patterns describe solutions to recurring reengineering problems. These reengineering problems occur when new or changed requirements cannot be satisfied by easy and local changes to the design.

To draw a clear dividing line between design patterns and reengineering patterns: Whereas design patterns describe a "good design" reengineering patterns describe how to change a legacy design into a good design. Since these changes are far away from being trivial or local, the description of the "good design" as in [GHJV95] does not enable software engineers to perform the necessary changes.

This paper presents the reengineering pattern *Transforming Inheritance into Composition* which supports the software engineer in making parts of the system more flexible by introducing the powerful technique of delegation. The format of the reengineering pattern was developed during the FAMOOS project which aims at the reengineering of object-oriented legacy systems into object-oriented frameworks. The format is based on the design pattern format [GHJV95] but it contains some new sections:

[*]Copyright © 1999 by Benedikt Schulz (`bschulz@fzi.de`) and Thomas Genßler (`genssler@fzi.de`), Forschungszentrum Informatik, Haid–und–Neu–Str. 10–14, 76131 Karlsruhe, Germany. Permission is hereby granted to copy and distribute this paper for the purpose of the EuroPLoP'99 conference and its proceedings.

- In the Structure section we describe two structures: A problem structure which suggests the application of the reengineering pattern and a target structure which is the result of the application.
- Since reengineering patterns describe transformations of software systems there is a section on the process of these transformations.
- Performing program transformations by hand is a difficult and error-prone task. Tools can help significantly in the application of reengineering patterns, therefore there is a section about tool support.

We used this form to communicate the lessons-learned of the project to the "normal" software engineer to help solve reengineering tasks.

Thumbnail

In some cases the inheritance relationship between classes is too inflexible and hard to understand. Therefore replace the inheritance relationship by a component relationship and delegate 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 1.

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

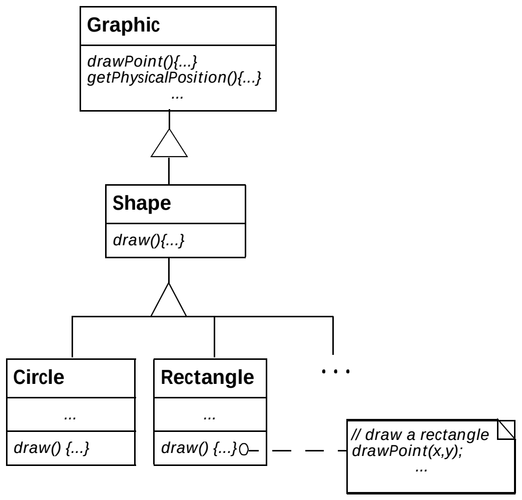

Figure 1: Initial Situation

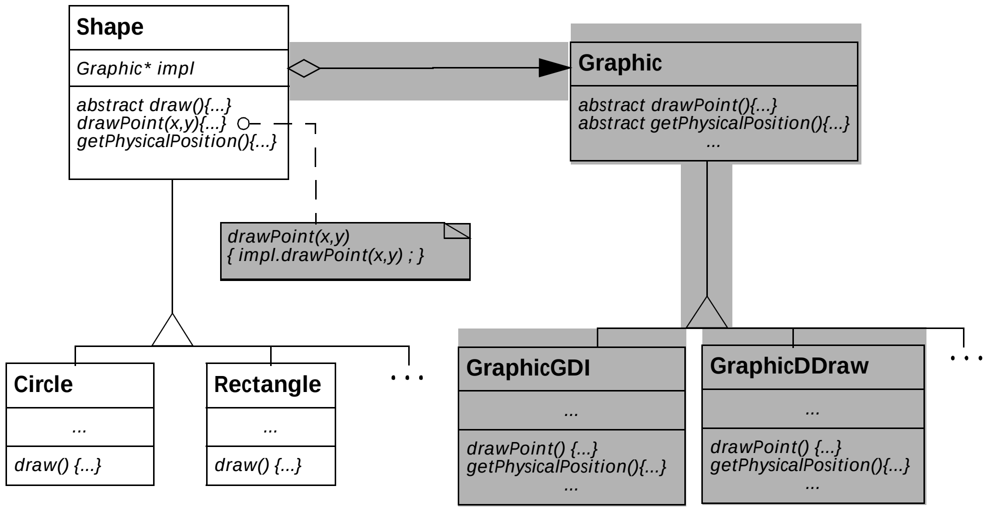

Figure 2: Target Structure

"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 2 where new or changed entities are marked grey.

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 [GHJV95]. (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.

Context

You review a legacy system. You look at a certain inheritance relationship and find out that

- *you should have used* a design pattern based on the Objectifier design pattern [Zim95] and the technique of delegation (e.g., Bridge [GHJV95], Strategy [GHJV95] or State [GHJV95][DA96]) *but you have not used* it.
- it 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.

Problem

In some cases, the inheritance relationship is not the appropriate solution. The relationship would have been better modeled using a component relationship and delegation. Transforming your legacy design to the new design without changing the functionality of the system is difficult and error-prone.

Forces

- The application of the reengineering 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.
- 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.
- The application of the reengineering pattern can improve your design if you encounter one of the following problems:
  - 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).

Reengineering Goals. The goal of the Transforming Inheritance into Composition reengineering 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 reengineering 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 Patterns. The Transforming Inheritance into Composition reengineering pattern is related to all design patterns which rely on the Objectifier design pattern [Zim95] and delegation like

- Bridge
- Strategy
- State

According to [Zim97] this reengineering pattern does not only describe a suitable target structure for a certain problem (like a design pattern) but the process of how to apply a design pattern to an existing design.

Structure

The problem structure is depicted in Figure 3. The Transforming Inheritance into Composition reengineering pattern leads you to the target structure depicted in Figure 4

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 3. After application of the reengineering pattern in Figure 4 `Delegator` will make use of these services by delegation.
- **Leaf_1, Leaf_2, ...** (Circle, Rectangle, ...) are the leafs 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..

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`.

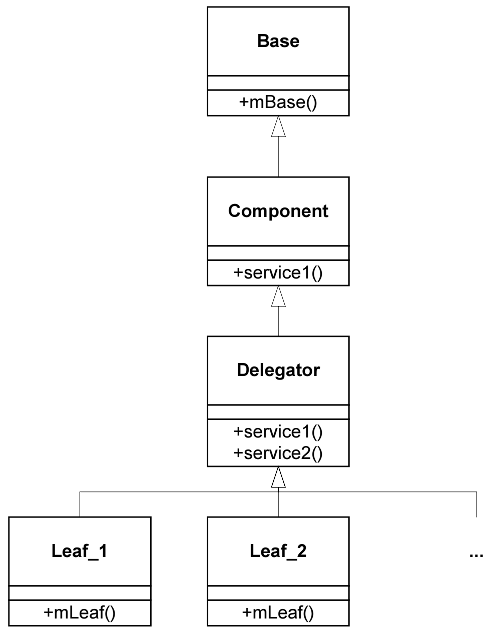

Figure 3: Problem Structure for the reengineering pattern

Consequences. The advantages and disadvantages of the target structure in comparison to the problem structure are discussed in the following:

- Positive benefits
  - Transforming inheritance into composition solves an important and basic reengineering problem and the application of the reengineering pattern allows for the introduction of several known design patterns [GHJV95].
  - 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 4.) 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 [Zim97] (rather than building blocks) and the refactoring approach presented in [Opd92]. This idea is presented and discussed in detail in [SGMZ98].

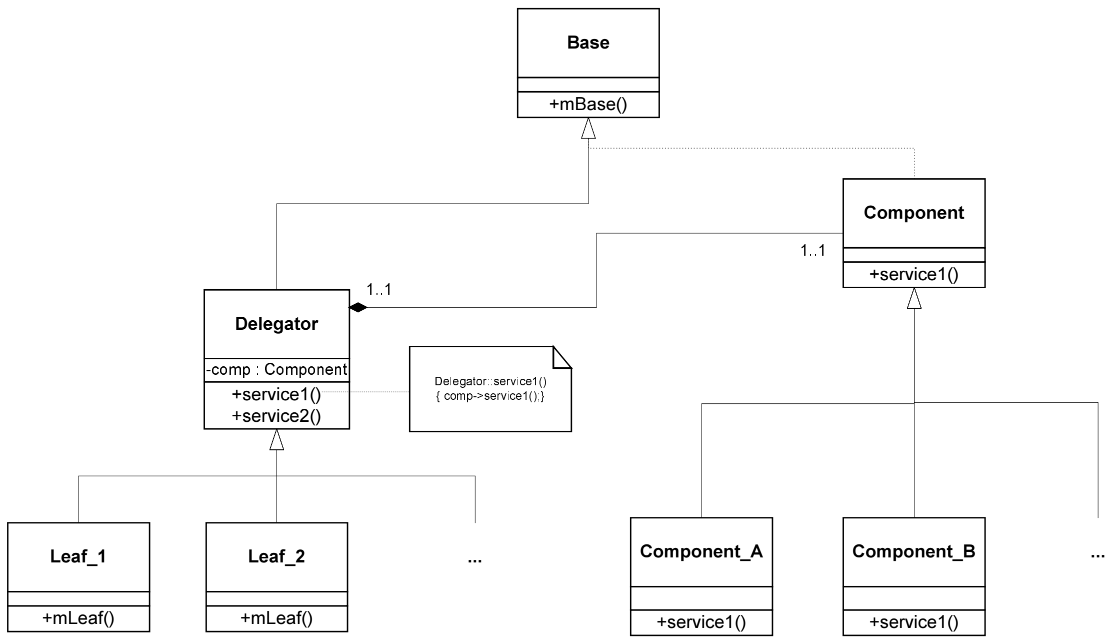

Figure 4: Target Structure for the reengineering pattern

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, an 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.
- 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 the Transforming Inheritance into Composition reengineering pattern 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 3 and the target structure depicted in Figure 4.

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 4.

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 the Transforming Inheritance into Composition reengineering pattern 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 spliting the `Component` class into two separate classes.

Language Specific Issues.

- In C++ you should realise 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 the Transforming Inheritance into Composition reengineering pattern.

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 [Zim95] are the fundamentals of this reengineering pattern and the resulting target structure is closely related to the Bridge, Strategy and State design patterns [GHJV95]. A good understanding of these design patterns helps to use the reengineering pattern.

Tool

The detection of pairs of classes which are never used polymorphically can be done with the tool-set *Goose* [BC98][Ciu99]. *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 [Opd92] you can use every tool which supports this technique, such as the *Refactoring Browser* [RBJ97] 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 [Moh98]. *RefaC++* can perform a subset of the refactorings presented in [Opd92] and can also apply the Bridge design pattern automatically.

Currently, we are developing a refactoring tool for JAVA which will be integrated into the commercial CASE tool *Together/J*. We already support the introduction of several design patterns (e.g., Bridge, State, Strategy) into object-oriented systems and there will soon be more supported design patterns.

Applied

Transforming Inheritance into Composition has 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.
- Currently 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.
- [RJ96] describes how frameworks evolve. In the White-box Framework design pattern [RJ96] 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.

References

[BC98] H. Bär and O. Ciupke. Exploiting design heuristics for automatic problem detection. In Stéphane Ducasse and Joachim Weisbrod, editors, *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report, June 1998.

[Ciu99] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. Accepted at the TOOLS USA '99 conference, March 1999.

[DA96] P. Dyson and B. Anderson. State patterns. In *First European Conference on Pattern Languages of Programming*, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Moh98] B. Mohr. Reorganisation objektorientierter Systeme. Masters thesis, Forschungszentrum Informatik (FZI) an der Universität Karlsruhe (TH), March 1998.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[RBJ97] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. http://st-www.cs.uiuc.edu/users/brant/Refactory/RefactoringBrowser.html, April 1997.

[RJ96] D. Roberts and R. Johnson. Evolving frameworks - a pattern language for developing object-oriented frameworks. http://st-www.cs.uiuc.edu/users/droberts/evolve.html, 1996.

[SGMZ98] B. Schulz, T. Genßler, B. Mohr, and W. Zimmer. On the computer aided introduction of design patterns into object-oriented systems. In *27th Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE, 1998.

[Zim95] W. Zimmer. Relationships between design patterns. In J. Coplien and D.C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Zim97] W. Zimmer. *Frameworks und Entwurfsmuster*. Ph.D. thesis, Universität Karlsruhe, 1997.

Acknowledgements

Parts of this work have been funded by the European Union under the projects FAMOOS no. Esprit 21975 and TROOP no. Esprit 27291. Many thanks to the people that have commented on earlier versions of this document: Markus Bauer, Oliver Ciupke, Serge Demeyer, Stéphane Ducasse and Sander Tichelaar. We thank our shepard at EuroPLoP 1999, James Noble, for his valuable comments that helped to improve the pattern a lot. Last but not least we'd like to thank the people who workshop'ed the paper at the conference.