Support Reengineering by Type Inference – A Reengineering Pattern[*]

Markus Bauer
bauer@fzi.de

September 28, 1999

Abstract

Reengineering patterns describe and discuss techniques that support reengineering tasks. They help a software engineer to understand the structure and the inner workings of a software system, to identify design problems and to improve the system in order to make it more flexible and extensible.

In this paper we present a reengineering pattern that shows how you can use *type inference* to facilitate the reengineering of systems that are written in *Smalltalk* or a similar dynamically typed object oriented language.

Keywords: object orientation, reengineering, pattern, type inference, Smalltalk

Introduction

The ability to reengineer object oriented legacy systems and transform them into more flexible and extensible systems has become a matter of vital interest in today's software industry. To help software engineers to do this, the FAMOOS project (see `http://dis.sema.es/projects/FAMOOS`) develops a handbook of good reengineering practices and techniques. To present these techniques in a uniform and easily accessible way a reengineering pattern format was developed.

On the following pages, we introduce a reengineering pattern conforming to an improved version of that pattern format[1]. The pattern describes how you can support the reengineering of a system written in a dynamically typed object oriented language by enriching its source code with type annotations.

[*]Copyright © 1999 by Markus Bauer, 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.

[1]Some patterns following an older version of this format have already been submitted to EuroPLoP '98, even more patterns will show up in the upcoming FAMOOS handbook.

Use Type Inference

Thumbnail

It is hard to understand the structure and the workings of a software system written *Smalltalk* (or in any other 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.

Context

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* [RBJ97][2] or tools that calculate software product metrics (like those described in [CK94]).
- You want to reengineer or rewrite the system using a statically typed programming language[3], 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.

Example

Consider some code fragments for a dynamically typed application that manipulates drawings. Such an application might have a class `Container` for storing some objects. Listing 1 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?

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

[3]Banking houses and insurance companies, for example, are often interested in replacing existing Smalltalk applications with statically typed Java or C++ applications.

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

Listing 1: 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 to manually trace the execution of your code and guess on what's going on in your system, but for larger systems, this is an infeasable 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 [EK98]. These approaches simulate Smalltalk's dynamic type system in Java. The resulting code, however, is not authentic Java code and hard to understand and maintain. Additionally such code 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 listing 2)[4] 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 listing 2 make code much easier to understand and that they contain valuable information about the inner workings of a system.

[4]A type annotation for a method is denoted as a comment in the Smalltalk-like code examples and has the form " $R \times A_1 \times \ldots \times A_n \to X$ ", where $R, A_1 \ldots A_n, X$ are sets containing types (classes of the system). This means, that the receiver object of a method send can have the types in R, the ith argument object can be of one of the types in A_i, and the result of the message send has one of the types in X. Thus, the method `add` in listing 2 is used for a `Container`, accepts a `Point`-, `Line`- or `Spline`-object as an argument and returns nothing (or `self`, respectively.)

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

Listing 2: Method `add` annotated with type information.

Implementation

Type inference ususally can't be done manually for reasonable large and complex applications. Therfore, 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 [ASU86] *(static type inference)* to analyze 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. [RBFDDar] 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 has to read in the complete source code of the application and analyzes it to construct a data flow graph. This is done by representing the application's expressions as nodes in the graph, and by modeling 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.

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 1.

```
x := 0;
y := 2.5;
x := y;
```

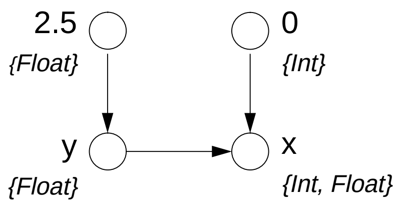

Figure 1: Data flow graph.

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 1, 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 2.

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 [Age95][5]. The basic idea is to create seperate sub graphs for each method and link all those subgraphs together in an appropriate and efficient way.

After the graph has been completly 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 analyze the data flow through an application. To make our approach work, we have to analyze 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 analyzed without application code using or instantiating them. Then, however, the inferred types are only valid in the specialized context of the particular application.

[5]There are other algorithms that also allow the tracking of data flow across method boundaries, for example [PS91], [OPS92], [PC94], 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[Age94].

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

Invocation of `max`:

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

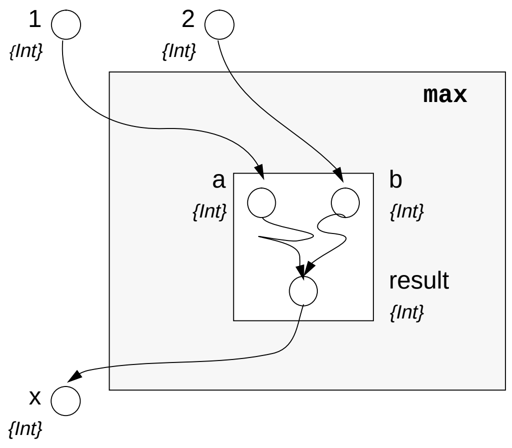

Figure 2: Data flow across method boundaries

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 [Li98]).

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 feasable for larger systems, at least, 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.

Also, this pattern is related with the reengineering pattern *Missing Technical Documentation*, which describes a set of general techniques to improve the documentation of a software system.[7]

Known Uses

ObjectShare has used type annotations (like those that can be computed by applying this pattern) to

[6]A detailed discussion of the algorithm, especially regarding complexity and precision can be found in [Age95] or in [Bau98].

[7]The pattern *Missing Technical Documentation* is included the FAMOOS handbook's collection of reengineering patterns. Its thumbnail is given by *Improve the understandability of a software system by adding comments to the system's source code that provide (higher level) design information.*

document large parts of the source code to the *Visualworks Smalltalk* environment. This emphasizes 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 visualizing software structures [Ciu97], checking design heuristics [BC98] and calculating software metrics [Mar97] can analyze 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. 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[Li98].

The type inference algorithm is also used to facilitate a mostly automatic translation of dynamically typed Smalltalk applications into statically typed Java applications [Bau98]. 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.

Acknowledgements

I would like to thank all people who helped me writing early versions of this pattern, in particular Oliver Ciupke and Benedikt Schulz. Thanks are also due to Jutta Eckstein and Christa Schwanninger who, as EuroPLoP shepherds, suggested a lot of improvements to the pattern and its structure and taught me a lot on writing a pattern in general. Further thanks go to Stéphane Ducasse and Sander Tichelaar for additional comments, as well as to all people who "workshopped" this paper during EuroPLoP'99.

References

[Age94] Ole Agesen. Constrained-based type inference and parametric polymorphism. In *Proceedings of the First International Static Analysis Symposium (SAS '94)*, volume 864 of *LNCS*. Springer-Verlag, 1994.

[Age95] Ole Agesen. The Cartesian product algorithm. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *LNCS*, pages 2–26, Berlin, GER, August 1995. Springer-Verlag.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[Bau98] Markus Bauer. Reengineering von Smalltalk nach Java. Master's thesis, Institut für Algorithmen und Datenstrukturen, Universtät Karlsruhe, 1998.

[BC98] Holger Bär and Oliver 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.

[Ciu97] Oliver Ciupke. Analysis of object-oriented programs using graphs. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology – Ecoop'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 270–271, Jyväskylä, Finnland, March 1997. Springer.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[EK98] R. L. Engelbrecht and D. G. Kourie. Issues in translating Smalltalk to Java. In Kai Koskimies, editor, *Compiler Construction 98*, volume 1383 of *LNCS*. Springer, 1998.

[Li98] Jinhua Li. Maintenance support for untyped object-oriented systems. `http://www.informatik.uni-stuttgart.de/ifi/se/people/li/`, 1998.

[Mar97] Radu Marinescu. The use of software metrics in the design of object oriented s ystems. Master's thesis, University P̈olytechicaT̈imisoara, sep 1997.

[OPS92] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349. Springer-Verlag, New York, N.Y., 1992.

[PC94] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, October 1994.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. Technical Report DAIMI PB - 345, Computer Science Department, Aarhus University, March 1991.

[RBFDDar] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. Dynamic type inference to support object-oriented reengineering in Smalltalk. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS (forthcoming). Springer-Verlag, 1998, To appear.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Journal of Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.