Efficient Object-Oriented Software with Design Patterns*

Mario Friedrich[2], Holger Papajewski[1], Wolfgang Schröder-Preikschat[1], Olaf Spinczyk[1], and Ute Spinczyk[1]

[1] University of Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany,
{papajews,wosch,olaf,ute}@ivs.cs.uni-magdeburg.de
[2] GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany,
friedric@first.gmd.de

Abstract. Reusable software based on design patterns typically utilizes "expensive" language features like object composition and polymorphism. This limits their applicability to areas where efficiency in the sense of code size and runtime is of minor interest. To overcome this problem our paper presents a generative approach to "streamline" pattern-based object-oriented software. Depending on the actual requirements of the environment the source code is optimized with a transformation tool. The presented technique provides "scalable" software structures and thus reconciles reusability with efficiency of pattern-based software.

1 Introduction

Design Patterns [6] are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". They are widely accepted as a very useful approach to ease the design of object-oriented software. Though no specific programming language nor coding style is forced by design patterns, a pattern description typically contains a class diagram, e.g. based on OMT or UML, and sample code. The code directly reflects the class diagram in order to achieve the same reusability. It excessively utilizes "expensive" language features like object composition via pointers and polymorphism. This raises the question of the efficiency of design pattern-based software and object-oriented software in general.

In some application domains even a comparably fast object-oriented language like C++ is not fully accepted because of its expensive features like dynamic binding. An example for such domains is the embedded systems area were now the "Embedded C++"[1] standard should increase the acceptance. The same holds for the core of numerical application and (non-research) operating systems.

From our experience in the operating systems area we can state that it is hard to design and implement reusable - and at the same time efficient, overhead

* This work has been supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1.

[1] Embedded C++ is a subset of C++ omitting the above mentioned expensive features.

free - object-oriented software, but it is even harder with design patterns. A pattern helps to design the class diagram of a software module. But, what is the relationship to the program code that has to be implemented? Normally the relationship is a direct mapping. Modern visual design tools generate the source code or the code is entered in the traditional way by adapting the pattern's sample code manually. This leads to flexible reusable code: Implementations and interfaces are separated by using abstract base classes and object composition is often applied to build flexible structures of objects that can be changed even at runtime. On the other hand this reusability and flexibility poses a significant overhead: abstract base classes, e.g., lead to dynamically bound (virtual) function calls. In [5] a median execution time of 5.6% and a maximum of 29% of dynamic dispatch code for a set of sample applications which quite sparingly execute virtual function calls is documented. Object composition implies pointers in every object of a specific class, thus wasting memory in the case that a dynamic modification of object connections is not needed. Last but not least the software module is often more complicated to use than a non-reusable variant which is especially trimmed to the needs of a specific application.

Our approach to overcome this problem follows the program family concept [8]. Different versions of a software module are generated to fulfill the requirements of its various application environments. Domain specific information is used to simplify the module structure as far as possible. This is done by a source-to-source transformation tool which works like an aspect weaver (see AOP [7]) with configuration information as aspect program and the module as component code. The generated code can be fed into any standard compiler which can then generate the optimized executable.

In the remaining sections of this paper we discuss the efficiency and optimization of pattern-based software in the context of application domains with hard resource constraints in more detail. This is followed by a presentation of our results and a discussion of related work.

2 Problem Discussion

Pattern-based software should be reusable and extensible without touching existing source code. A good example of how these design goals can be achieved shows the *Strategy Pattern*. The structure of this pattern is presented in figure 1 as a UML class diagram.

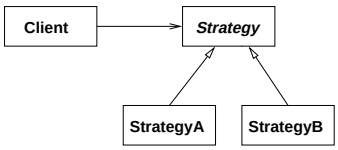

Fig. 1. The strategy pattern

The idea behind this pattern is to separate the interface and the implementation of some strategy. By doing this other strategies implementing the same interface can be easily integrated into the system without touching existing source code. It is even possible to change the strategy that is associated with each client object at runtime. The strategy pattern can be found as some kind of sub-pattern in many other design patterns. This makes it a very typical example to illustrate the overhead that may be posed on a software system if the implementation directly reflects the presented class diagram.

Now consider a concrete pattern instantiation: The client objects are memory managers in an operating system that know the start address and size of a memory block. They allow other objects to allocate and free pieces of memory from the pool they manage. The actual allocation can be done with different strategies like "best fit" or "first fit". The implementation of this little memory management subsystem should be reused in different operating system projects and may be extended by other strategies like "worst fit" in the future. Therefore the most flexible (pattern-based) implementation with a pointer to an abstract strategy base class is selected (figure 2).

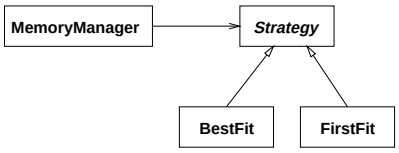

Fig. 2. The instantiated pattern – a memory management subsystem

But this extensible and reusable implementation causes an overhead in operating system projects where the flexibility is not needed. As a first problem scenario consider a system with `MemoryManager` objects that never need the "first fit" strategy. Why should they access their associated strategy object via the abstract base class? Assuming C++ as our implementation language they could simply contain a pointer to a `BestFit` object as shown in figure 3. A special purpose design like this would omit the virtual function call needed to dynamically switch between the different strategy method implementations at runtime.

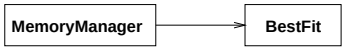

Fig. 3. Scenario 1 – single strategy class

To have an impression of the overhead that results from the abstract base class we have implemented a simple test application that creates two `BestFit` objects and a `MemoryManager` object. In this test scenario the method bodies

were left empty. Only an output statement was include to track the correct behaviour. After associating the first `BestFit` object to the `MemoryManager` one of the methods of the strategy is called. Then the second `BestFit` object is associated to the `MemoryManager` and the method is called once again. The test application has been compiled in two versions: The first using the original strategy pattern (figure 2 without the `FirstFit` class) and the second using the simplified structure from figure 3. Table 1 shows the resulting static memory consumption[2]. To exclude the constant size of the startup code and the C library output function which are magnitudes bigger than our simple test modules we present only the object file sizes here.

Table 1. Scenario 1 – memory consumption

Version 1: Original Pattern					Version 2: Simplified Structure				
text	data	bss	sum	filename	text	data	bss	sum	filename
113	12	0	125	BestFit.o	36	0	0	36	BestFit.o
34	0	0	34	Manager.o	25	0	0	25	Manager.o
146	4	20	170	main.o	164	4	12	180	main.o
293	16	20	329	all	225	4	12	241	all

The sum over all sections in all linked object files in version 1 is 329 in comparison to 241 bytes in version 2. In other words, the pattern-based implementation requires about 36.5% more memory space than the specialized implementation.

Now consider a second scenario where the `MemoryManager` does not need to switch between different strategy objects at runtime because only a single global `BestFit` strategy object exists. In this case a specialized implementation can omit the abstract `Strategy` class, too. The global "best fit" strategy can be implemented by a class `BestFit` with all methods and data members declared as static. Now all strategy methods can be called without an object pointer and the `MemoryManager` objects can get rid of them. This is a significant reduction if one considers ten or hundreds of `MemoryManager` objects. But even with only a single object the memory consumption of the simplified implementation is significantly lower than the consumption of the pattern-based version. The exact numbers are presented in table 2. Here the overhead of the pattern-based implementation in this scenario is 160%.

Design patterns make the trade-offs of the different design options explicit, thus a designer can select whether to accept the overhead of the pattern-based implementation or to implement a specialized version of the subsystem. But in both cases the designer will lose. In the first case the price is mainly efficiency and in the second it is the reusability of the subsystem together with the risk to introduce errors.

[2] The code was generated with egcs 2.90.27 on Linux/Intel. To omit unnecessary run-time type information code the compiler options `-fno-rtti` and `-fno-exceptions` were used.

Table 2. Scenario 2 – memory consumption

Version 1: Original Pattern					Version 2: Simplified Structure				
text	data	bss	sum	filename	text	data	bss	sum	filename
113	12	0	125	BestFit.o	38	4	0	42	BestFit.o
34	0	0	34	Manager.o	29	0	0	29	Manager.o
121	4	16	141	main.o	42	0	2	44	main.o
267	16	16	299	all	109	4	2	115	all

3 Optimization

The overhead documented in section 2 cannot be tolerated in many application areas and should be avoided. Instead the complexity of the system structure should scale with the actual requirements of the subsystem's environment. It should not be ruled by possible future system extensions which may never happen. At the same time the implementation should be reusable.

To achieve this goal we combine the pattern-based software design with knowledge about the actual system requirements in a configuration phase. This input is used by an optimizer that selects a specialized system structure which is free of the pattern overhead (optimization phase). For the software developer this process is transparent. The source code directly reflects the pattern-based design, thus is as reusable and extensible as the design itself. The necessary specialization is done by a source code transformation tool in the generation phase. The complete process is illustrated in figure 4.

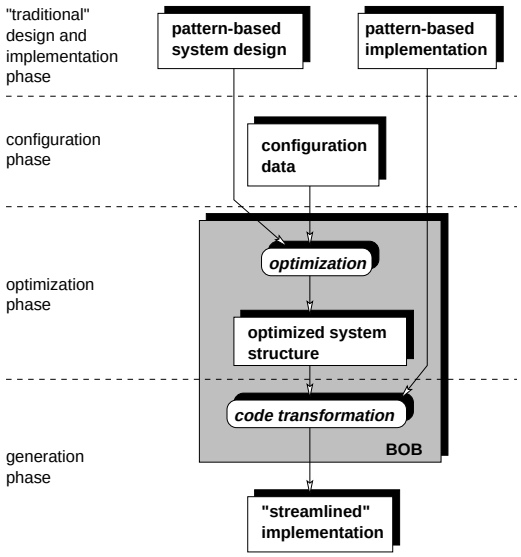

Fig. 4. The complete optimization process

The result of this approach is a tool that allows to generate a family of fine-tuned subsystem implementations from a single reusable pattern-based source code. The configuration step selects the best suited family member.

Section 4 discusses the software architecture of the "black optimizer box" (BOB) and the code transformation with further details. The remaining parts of the current section focus on the optimization phase and the rules and conditions guiding the optimization process.

3.1 Pattern Descriptions

An important prerequisite for the optimization of the pattern-based implementation is the documentation of the applied pattern. The documentation is needed to avoid a complex source analysis and the software developer should document instantiated patterns anyway. The pattern description consists of the pattern name, a name of the created pattern instance and list of classes building the pattern with their associated roles as shown in figure 5.

```
define CommunicationSystem decorator
 {
   component: Driver;
   concrete_component: Ethernet;
   decorator: Protocol;
   concrete_decorator: IP;
   concrete_decorator: Crypt;
 }
```

Fig. 5. A pattern description

Pattern instances usually can be extended without touching existing source code. With the `extend`-command (see figure 6) this is possible with the pattern description as well .

```
extend CommunicationSystem
 {
   concrete_component: CAN;
 }
```

Fig. 6. A pattern description extension

Having this description (pattern definition and extensions) the optimizer "knows" the names of the classes forming a pattern and implicitly how the connections between these classes are implemented, i.e. the class diagram (figure 8, left side). In figure 4 this corresponds to the "pattern-based system design" box.

The example subsystem shown above is a simplified communication system. It contains an ethernet and CAN bus driver. Other hardware drivers can be added easily because of the abstract `Driver` base class. With the decorator pattern it is possible to connect a protocol like IP with any hardware driver or even to create chains of protocols, e.g. to create an IP communication driver with encryption.

3.2 Requirement Definitions

The pattern instance (`CommunicationSystem` in the example) is reusable in different environments. To find an optimized system structure for the pattern-based subsystem it is necessary to specify the requirements of its environment. The requirement definition that is used for this purpose corresponds to the "configuration data" in figure 4. It is provided in the configuration phase, thus can be exchanged for different application environments.

The requirement definition mainly consists of the list of class names that are used from the pattern-based subsystem and a description in what way objects are instantiated. An example is given in figure 7.

```
require CommunicationSystem
 {
   IP [Protocol (Ethernet)];
 }
```

Fig. 7. A pattern requirement definition

This requirement says that only the `IP`, `Protocol` and `Ethernet` classes are needed (directly) by the environment of the pattern-based subsystem. The `Protocol` class should contain a pointer to an `Ethernet` object, instead of a pointer to any kind of `Driver`.

Before the system structure can be modified the requirement definition is checked against the class hierarchy given by the pattern description: All classes mentioned have to be known and a relation to another class (like "`Protocol (Ethernet)`") is restricted to the subtree below the class referenced in the original class hierarchy. Inheritance relationships cannot be changed but one can change an object composition built with a pointer to an aggregation (like "`Protocol [Ethernet]`"). It is also possible to create specialized class versions with the alias feature, e.g. "`EP is Protocol [Ethernet]; CP is Protocol [CAN];`".

3.3 Optimization Rules

With the description of the implemented system structure and the requirements in the actual application scenario a "streamlined" system structure can be automatically derived. The following simple rules are applied:

- All classes, that are not mentioned in the requirement definition and that are not base classes of them, can be removed.
- The class relation changes given in the requirement definition are applied.
- Classes, that are not referenced from classes outside their own inheritance subtree are "devirtualized", i.e. all pure virtual functions are removed and virtual functions are declared non-virtual.
- Empty classes are removed.

Figure 8 shows two examples of requirement definitions and the resulting system structures.

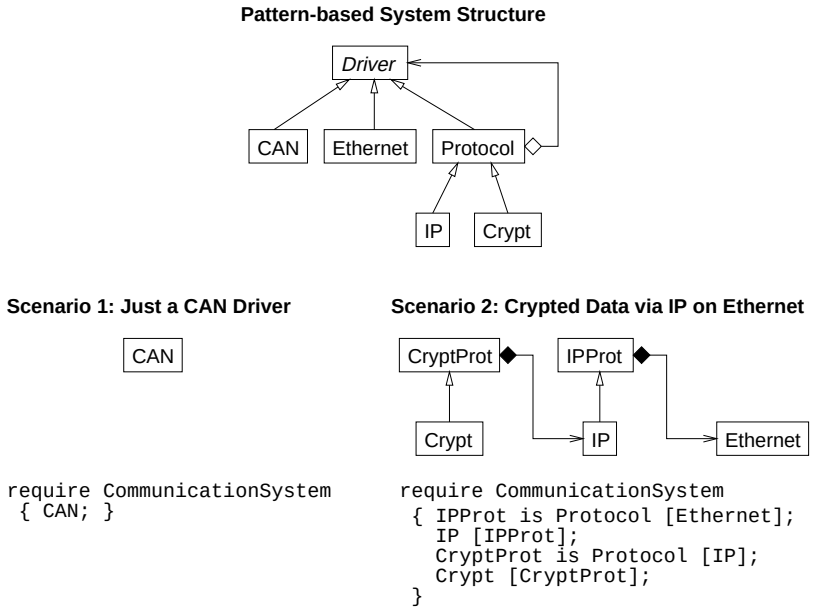

Fig. 8. Optimized system structures

3.4 Multiple Requirement Definitions

Currently only a single requirement definition that describes the demands of the environment of the pattern instance is allowed. With this restriction the environment is forced to have a common view on the classes in the pattern. A future extension of our transformation system shall handle this problem.

As an example consider a subsystem that requires just a single CAN driver like scenario 1 of figure 8 and a second subsystem requiring an IP communication via ethernet with data encryption. Both requirements do not conflict and can be merged easily.

Even if one subsystem specifies "`IP [Protocol [Ethernet]]`" and instantiates its driver objects with "`IP ip;`" and another subsystem specifies "`IP [Protocol (Ethernet)]`" a common requirement can be found automatically: The second subsystem instantiates IP objects with individual ethernet driver objects like "`Ethernet eth; IP ip (&eth);`". An algorithm has to find the "cheapest" system structure that fulfills both requirements. In the example it is the requirement definition of the second subsystem. To have optimization transparency, the source code of the first subsystem has to be adapted by creating an `Ethernet` object any time the source code exhibits a creation of an `IP` object.

4 Implementation

Figure 4 shows two actions inside of BOB: The "optimization" and the "code transformation". These two steps directly reflect the two-level system architecture that implements the optimization process.

The upper level is responsible for the optimization. It reads and analyses the pattern descriptions and requirement definitions given in the PATCON language, that was introduced by the examples in figure 5, 6 and 7. With the optimization rules described in subsection 3.3 a list of transformation commands is generated.

The transformation commands are the input for the lower level that does the code transformation. The lower level is implemented by an aspect weaver that handles the customization of object interactions in general. The actual transformation work is done by "PUMA for C++", a parser and manipulator library for C++ code, which is going to be extended to an aspect weaver environment with a plug-in interface for multiple aspect weavers in the near future.

5 Results

A prototype implementing the algorithms and transformations described in the previous sections is running. To demonstrate the results, we have implemented a communication system like that from figure 8. The difference to that system is that we only have a single "hardware" driver which implements a simple mailbox for local inter-thread communication. Our two protocols are a checksum calculation and encryption. With this "renaming" scenario 1 is just a single mailbox and scenario 2 is checksum protocol on encrypted messages. The resulting code sizes and runtimes are presented in table 3. For the measurements again the egcs-2.90.27 on a Linux/Intel system was used. The sizes are given in units of bytes and the runtimes in Pentium II clock cycles.

The code sizes contain the complete test application consisting of two threads that communicate with each other via a shared mailbox. It also consists of classes that implement abstractions of communication addresses and data packets, the mailbox code and a lot of inlined operating system functions for thread synchronization and scheduling. All of these parts are equal in the pattern-based and the optimized version. So results like those of table 1 and 2 could not be

Table 3. Code sizes and runtimes

	Scenario 1		Scenario 2	
	Pattern	Optimized	Pattern	Optimized
code	1532	1477	3676	3550
data	36	20	148	100
bss	20	16	208	188
sum	1588	1513	4032	3838
∅ receive time (sync.)	675	424	1694	1548
∅ receive time (async.)	314	307	675	518

expected. Nevertheless there is a difference of 75 bytes in scenario 1 and 194 bytes in scenario 2. This is a reduction of about 4.8% in both cases.

The runtime data shows the number of clock cycles needed for synchronous and asynchronous receive operations. In scenario 1 in both versions no virtual function call takes place and no unnecessary constructor code is executed in the pattern-based version. So we can explain the big difference (675-424) only with a better cache hit rate because of the slightly reduced memory consumption. In scenario 2 there is a speedup of 8.6% in the synchronous case where context switch times are included and 23.3% in the asynchronous case.

6 Related Work

The generation of pattern implementations is proposed in [3]. This work of IBM has much in common with our work since different implementations can be generated depending on so called "trade-offs". These trade-offs correspond to the optimization conditions in section 3. The difference is that this work mainly focuses on the increased productivity with automatic code generation tools and not the efficiency and optimization aspects.

The specialization of source code in our approach is a kind of static configuration. This can also be done with template metaprogramming as proposed in [4]. While that paper presents a general purpose implementation technique for static configuration we concentrate on design patterns. The specialized source code could also be generated with template metaprogramming, but the optimization would have to be a manual process. Our approach simplifies the software development because no template programming is necessary and only the used patterns need to be documented. The configuration information can be supplied in a problem-adequate language.

Virtual function call optimization by source-to-source transformation of C++ code was successfully applied in [1]. Their optimizer implements a class hierarchy analysis and a profile-based type feedback optimization. Similar ideas are presented in [2]. Both papers concentrate on runtime. A simplification of the whole program structure or the elimination of member variables that lead to reduced memory consumption was not documented. This would require extra knowledge about the implementation like the pattern descriptions.

7 Conclusions

We have not completed our work on this topic yet. A lot of implementation and especially documentation work still has to be done, and our experiences with the system in future projects will probably have an influence on the optimization rules and conditions, too.

Nevertheless the tools and techniques presented in this paper show how reusability and extensibility of a pattern-based software design can be reconciled with the efficiency of the resulting implementation by automatically scaling software structures. The software designer simply needs to define requirements and make the applied patterns explicit inside a pattern description file. The rest of the optimization is completely transparent. Thus we are optimistic that our approach may raise the level of acceptance of pattern-based object-oriented software in application areas with hard resource constraints. For our development of PURE [9] - a family of object-oriented operating systems that targets the area of deeply embedded systems - patterns can now be applied without risking unacceptable overhead.

References

[1] G. Aigner and U. Hölzle. Eliminating Virtual Function Calls in C++ Programs. Technical Report TRCS95-22, Computer Science Department, University of California, Santa Barbara, December 1995.

[2] D. Bernstein, Y. Fedorov, S. Porat, J. Rodrigue, and E. Yahav. Compiler Optimization of C++ Virtual Function Calls. In *2nd Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996.

[3] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.

[4] K. Czarnecki and U. Eisenecker. Synthesizing Objects. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 18–42, Lisbon, Portugal, 1999. Springer Verlag.

[5] K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. In *OOPSLA'96 Proceedings*, October 1996.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997.

[8] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[9] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998.