Static and Metaprogramming Patterns and Static Frameworks

Philipp Bachmann
Institute for Medical Informatics and Biostatistics
Clarastrasse 15
CH–4058 Basel, BS
Switzerland

Revision: 1.5 *

September 24, 2006

Abstract

The classic UNIX principle to write code that generates code instead of writing this code yourself [Ray03] experiences a revival these days. Much research was done, the techniques are better understood now, and the generation tools were refined.

This pattern catalogue consists of adaptations of the Gang of Four design patterns [GHJV96f] Abstract Factory, Adapter, Strategy, and Visitor to the metaprogramming level. It shows that replacing runtime polymorphism by static polymorphism helps to move variation from the code level up to the meta level. Some of the patterns proposed are especially useful to facilitate portable code.

The patterns shown can be used to build static Frameworks [RJ98]. A simple example is also presented.

For all patterns proposed usage examples in popular existing applications or libraries were identified.

Each pattern presentation is accompanied with an example. These examples show sample code in C++. The template metaprogramming capabilities of C++ [AG05, CE00, VJ03a] allow to express both the code level and the meta level in the same programming language.

Contents

Chapter 1

Overview

Code generation can help to assemble a series of applications from the same set of separate parts at compile time, to explicitly represent the building plan in the generation software, and to allow for future adaptations by changing the building plan.

Generative programming [CE00] provides another way to deal with variation additionally to patterns based on runtime polymorphism. Domain specific languages (DSLs) describe how a system should be generated. If generative programming is available and understood some points of variation can be moved up from the software built into the DSL. This often leads to better optimization opportunities.

With generative programming at hand the recurring problem of software portability can be solved in an appropriate and more elegant way than without. Usually portability means cross platform portability. The availability of a product for many platforms can provide a significant competitive advantage as can be seen e.g. with Adobe FrameMaker, which is one of the very few publishing tools available for different operating systems. The term platform not only refers to a certain hardware architecture and operating system, but can also refer to other software subsystems the product interfaces with, e.g. database management systems from different third party vendors. Portability also has a temporal aspect. Portable code more likely can still be built in the future with future versions of operating systems, libraries, and compilers. As software often needs to be maintained for periods much longer than once planned this is an important goal. Building software with portability in mind can reduce risk and cost, which becomes manifest if the customers demand the product on a platform not requested before.

Different roles need different qualifications during the development process of portable applications:

1. Architects need a good sense of where to build variation in and where not.

2. Designers need to come up with a solution on how to represent variation in the software.
3. Implementors need in–depth knowledge on the concrete platform.

Patterns were identified on the metaprogramming level. This pattern catalogue assists you in both understanding variability necessary for portable applications and filling the gaps where supportive libraries are not available or can not be used. Goals always aimed at were to centralize the configuration and to reduce the need for conditional compilation.

The following table lists the patterns proposed.

Section	Title	Intent
2.1	Static Strategy	Delegate certain tasks a class implementation needs to perform to another class or instance. Allow for static configuration of the implementation of these tasks.
2.2	Static Visitor	Different from the Visitor pattern the Static Visitor pattern does not depend on compile time polymorphism at all. It breaks the dependency cycle present in the original Visitor design pattern.
2.3	Static Adapter	Adapt a series of different interfaces to a common interface. The decision which interface is actually to be adapted can be done at compile time. It has to be ensured that regardless of which interface is going to be adapted each adaptation results in the same interface not to break client code.
2.4	Static Abstract Type Factory	The Static Abstract Type Factory provides an extensible means to associate expressions on the level of the domain specific language with application data types.
3	Static Framework	The purpose of writing portable code is not only to provide an application for a variety of platforms, but to do so in a way that on each platform certain requirements concerning performance are met. Static Frameworks assist you in writing code that can be adapted more easily to multiple platforms while making sure that on each platform the application can fulfill its original purpose.

The presentation of each pattern follows the style well known from [BMR+00c] and [SSRB02d].

For the code examples C++ was chosen as the programming language, because its generative programming capabilities allow to stick with a single language for both the basic and meta level.

What can be said at all can be said clearly.

LUDWIG JOSEF JOHANN WITTGENSTEIN: Preface of Tractatus logico–philosophicus [Wit60]

Chapter 2

Metaprogramming Patterns

This chapter proposes the use of generative and template metaprogramming techniques [AG05, CE00, VJ03a] to express classic patterns by the Gang of Four [GHJV96f] on the metaprogramming level.

After reading this chapter the reader will know how to refactor code to move points of variation from the code level to the metaprogramming level.

As domain specific languages (DSLs) statically describe a system, using metaprogramming patterns can help with the design of portable code the same way as patterns help with the design of the concrete implementation for one platform.

Metaprogramming variants of other Gang of Four patterns can be found e.g. in [CE00, pp 224–234].

2.1 Static Strategy

An instance or class based bahavioral pattern

2.1.1 Also known as

Policy [Ale03, pp 27–51], [VJ03b, pp 429–436]

2.1.2 Intent

Delegate certain tasks a class implementation needs to perform to another class or instance. Allow for static configuration of the implementation of these tasks.

2.1.3 Example

Suppose a stack is to be implemented. The stack abstract data type itself does not depend on how exactly memory is allocated. Suppose this independence should be reflected by the design of the stack data structure to allow for plugging in different allocation strategies, for example one using allocation on the heap, one using allocation in shared memory segments, and another one allocating memory in terms of memory mapped files.

Traditionally such allocation code would be dynamically added by means of the Strategy [GHJV96k] design pattern as sketched in Listing 2.1.

Listing 2.1: Dynamically injecting a stack with an allocator

```
struct AllocatorIf {
  virtual ~AllocatorIf() {}
  virtual void *allocate(std::size_t) =0;
  virtual void deallocate(void *) =0;
};

struct NewAllocator : public AllocatorIf {
  void *allocate(std::size_t n) {
    return operator new(n);
  }
  void deallocate(void *p) throw() {
    operator delete(p);
  }
};

class IntStack {
  AllocatorIf *allocator_;
  ...
public:
  // Doesn't take ownership over allocator
  explicit IntStack(AllocatorIf &a) : allocator_(&a), ... {}
  ...
  void pop() throw() {
    // Calls "allocator_->deallocate()"
    ...
  }
  void push(int data) {
    // Calls "allocator_->allocate()"
    ...
  }
};
```

The details of `IntStack` do not matter here. They are similar to the implementation shown below in Listing 2.2. An implementation of the `AllocatorIf` interface is injected into `IntStack` on construction. Note that different instances using different allocators do not differ in type.

The Template Method design pattern [GHJV96j] was another implementation option. A disadvantage compared to Strategy was that implementation details factored out into subclasses can not be reused as Strategies can.

As the allocator can not be changed during the lifetime of the instance of `IntStack` with the exception of a potential assignment operator, using dynamic polymorphism may be considered too much of a good thing. So a way is sought to statically bind allocators to the stack instance while still separating allocator from stack code.

2.1.4 Context

The implementation of certain classes representing e.g. abstract data types consists of different concerns that crosscut each other [KLM+97]. These concerns are often bound to the class, not to its instances, and must then be kept immutable for consistency reasons.

2.1.5 Problem

How to inject implementation details into a class to allow for a flexible way to replace these details?

2.1.6 Forces

- Abstract data types are by definition independent of a special implementation. Their representation in code should be decomposed into a generic essence and implementation details to keep code duplication to a minimum even in the case that the implementation details need to be adapted to use the code within another environment.
- The decomposition into several parts should not result in runtime overhead.
- The implementation details itself should be general enough to be reused in the context of other abstract data types.

2.1.7 Solution

Separate an abstract data type into its essence and an exchangeable class or instance of another class it delegates implementation details to. Define a concept for the classes that represent these implementation details. Statically configure the abstract data type with the type of a model of this concept.

A first sketch of the solution is shown in Table 2.1.

Participants

AbstractDataType Class template that delegates implementation details to the Static Strategy it gets statically configured with. A Concrete Strategy might be offered as a default Static Strategy.

Client Client code instantiates the Abstract Data Type template for a Concrete Strategy.

Table 2.1: Class–Responsibility–Collaboration Cards

AbstractDataType	
Delegates implementation details to	StaticStrategy
Provides interface to	Client

(a) Abstract Data Type

Client	
Instantiates template	AbstractDataType, Concrete Strategy
Calls member function of	AbstractDataType

(b) Client

ConcreteStrategy	
Binds template argument of	AbstractDataType
Model of	StaticStrategyConcept
Encapsulates algorithm implementation	

(c) Concrete Strategy

StaticStrategyConcept	
Declares interface to algorithm	

(d) Static Strategy Concept

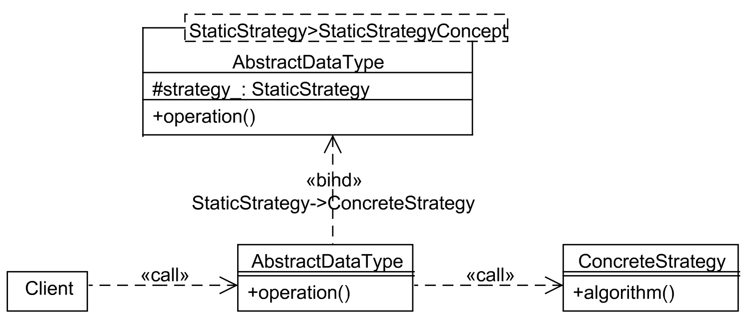

Figure 2.1: Class diagram illustrating Static Strategy

ConcreteStrategy Provides algorithms that can be used by Abstract Data Types within their implementations. Concrete Strategy is a model of Static Strategy Concept.

StaticStrategyConcept Defines interface each Concrete Strategy has to conform to. Static Strategy Concepts provide a contract to Abstract Data Types the latter can program against.

Figure 2.1 sketches the participants and their relations to each other.

Dynamics

The Client binds the Abstract Data Type template passing a Concrete Strategy. The Client instantiates the resulting type. It then calls member functions, which in turn delegate some implementation details to the Concrete Strategy.

Rationale

Some configuration issues can be decided early at compile time. In fact, some Abstract Data Types only work correctly, if their Strategies will remain fixed during the life time of the instance of the respective Abstract Data Type. Assembling code at compile time instead of virtual calls at runtime results in fewer indirections and less bias against inlining.

2.1.8 Resulting Context

Implementation details were factored out of the Abstract Data Type. The AbstactDataType is more reusable than before, and the Static Strategies can also be used to determine the implementation details of other Abstract Data Types. The Client can define its own Static Strategies.

Pros and Cons

The Static Strategy pattern has the following benefits:

1. *No runtime overhead.* As the compiler binds Concrete Strategy to Abstract Data Type, at runtime everything is readily prepared.

2. *Extensible.* If the Static Strategy Concept was published, the Client can replace a Concrete Strategy with customized implementations.

The Static Strategy pattern has the following liability:

1. *No relationship among different instantiations of Abstract Data Type.* Abstract Data Types bound to different StaticStrategies do not relate to each other. If it is intented to assign them to each other, there have to be special member function templates to enable this [Suta, Sutb], [Mey05b].

Additionally to these general pros and cons the following implementation specific ones were identified.

The implementation technique of the Static Visitor pattern shown has the following liabilities:

2. *No concept of Concrete Strategy.* The Concrete Strategies have to be models of the same concept Static Strategy Concept: They all have to provide member functions of the same names. Such concepts cannot currently be expressed in C++. There are matured proposals to overcome this issue, e.g. [DRS06, GS06].

3. *Definitions must be inlined.* This technique reveals implementation details in header files. This might not be appropriate.

2.1.9 Implementation

Once a decision was made what is the essence of the Abstract Data Type and which parts better should be factored out into a Static Strategy, the data structure repesenting Abstract Data Type has to be made statically configurable by turning it or its member functions into templates. A Static Strategy Concept has to be developed that declares the interface between Abstract Data Type and the Concrete Strategy.

Example Resolved

The code shown in listing 2.2 shows a stack data structure capable of storing integral numbers only and a simplified version of the Allocator concept of the C++ Standard Library. The `NewAllocator` shown as an example for all models of the simplified allocator concept acquires memory from and releases it to the freestore.

Listing 2.2: Statically injecting a stack with an allocator

```
struct NewAllocator {
  static void *allocate(std::size_t n) {
    return operator new(n);
  }
  static void deallocate(void *p) throw() {
    operator delete(p);
  }
};

template< typename Allocator =NewAllocator > class IntStack {
  struct IntNode {
    int data_;
    IntNode *next_;
    IntNode(int data,IntNode *next)
      : data_(data), next_(next) {}
  };
  Allocator allocator;
  IntNode *top_;
public:
  IntStack() : top_(0) {}
  IntStack(const IntStack &rhs) : top_(0) {
    try {
      for(const IntNode *ci=rhs.top_;0!=ci;ci=ci->next_)
        push(ci->data_);
    }
    catch(...) {
      clear();
      throw;
    }
  }
  ~IntStack() {
    clear();
  }
  IntStack &operator=(const IntStack &rhs) {
    IntStack tmp(rhs);
    swap(tmp);
```

```
    return *this;
  }
  void swap(IntStack &rhs) throw() {
    IntNode *const tmp=top_;
    top_=rhs.top_;
    rhs.top_=tmp;
  }
  void clear() throw() {
    while(top_)
      pop();
  }
  bool empty() const {
    return !top_;
  }
  void pop() throw() {
    assert(top_);
    IntNode *const tmp=top_;
    top_=top_->next_;
    // Call destructor to get plain, raw memory
    // (not really necessary here because of trivial destructor)
    tmp->~IntNode();
    // Delegate deletion to Allocator
    allocator.deallocate(tmp);
  }
  void push(int data) {
    // Delegate allocation to Allocator
    IntNode *node=allocator.allocate(sizeof(IntNode));
    try {
      // Construct instance into raw memory
      new(node) IntNode(data,top_);
      top_=node;
    }
    catch(...) {
      allocator.deallocate(node);
      throw;
    }
  }
  int top() const {
    assert(top_);
    return top_->data_;
  }
};
```

Note that different instances using different allocators differ in type in contrast to the version using the Strategy pattern as shown in Listing 2.1. Concepts are less restrictive than interfaces regarding to the exact signatures of member functions prescribed; the above stack will also compile bound to allocators with e.g. non–static member functions.

The dynamics of `IntStack< NewAllocator >::push()` is shown in Figure 2.2.

2.1.10 Variants

Depending on the purpose of the Strategy there are two different implementation options with respect to the granularity of configuration possible. Either the

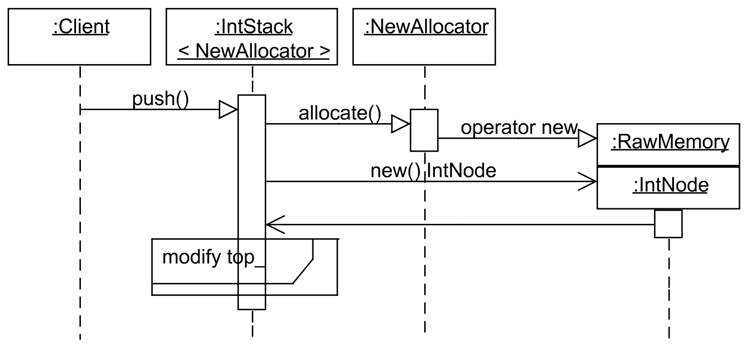

Figure 2.2: Sequence diagram illustrating Static Strategy

Strategy affects the whole class template Abstract Data Type and remains fixed during the whole lifetime of the template instantiation, or the member functions of Abstract Data Type are declared as templates, such that for each member function template and on each call a different Strategy can be chosen. This description concentrates on the first option. The second one can be implemented similar to the Static Visitor pattern (see Section 2.2).

2.1.11 Known Uses

Examples of Static Strategy can be found in existing software.

C++ Standard Library Allocator concept

The C++ Standard Library contains various containers, e.g. associative containers like `std::map<>`, arrays like `std::vector<>`, and list like structures like `std::stack<>`. All of these delegate allocation of their elements to a Static Strategy that must be a model of the Allocator concept.

C++ Standard Library StrictWeakOrdering concept

The associative containers of the C++ Standard Library additionally pose the requirement on their so called keys that for their type a binary function or function object exists that is a model of the concept StrictWeakOrdering. In other words the keys must be strict weakly ordered, and this order is represented by a comparison function or function object. The respective container instance delegates the task of comparing two keys to this.

The Standard C library contains e.g. the Quicksort implementation `qsort()`. It uses the Strategy pattern instead to both make the function independent of a special data type and delegate the comparison to user code, not the Static Strategy pattern and thus forbids inlining of the comparison function.

C++ Standard Library Algorithms

The C++ Standard Library provides a lot of algorithms that map a unary function to each element of a container. They cast popular uses of loops into function templates. On instantiation these templates are configured by an Iterator [GHJV96h] type and the type of the unary function, which in fact is a Static Strategy.

If the Static Strategies furthermore statically reflect their argument and return types using certain member type definitions e.g. by inheriting from `std::unary_function<>` or `std::binary_function<>` the functions can be chained: Binary functions can be turned into unary ones using one of the binders `std::bind1st<>` or `std::bind2nd<>`, and unary functions can be negated using `std::negate<>`.

2.1.12 Related Patterns

The Static Visitor pattern also inverts control flow.

The Strategy design pattern uses (runtime) polymorphism to allow for substitution of a concrete strategy by another implementation. The Static Strategy pattern is its static counterpart.

2.2 Static Visitor

The Visitor design pattern [GHJV96d] uses both polymorphism twice and overloading once. One purpose of polymorphism with the Visitor pattern is to keep the `accept()` member function general enough to accept more than one specific Visitor. Inside of each Visitor class overloading is used to let the compiler statically choose the correct member function. The other use of polymorphism in the Visitor design pattern is to allow for double dispatch. The class diagram corresponding to the original Visitor pattern is shown in Figure 2.3. The use of overloading shows, that the original Visitor pattern already contains strong aspects of compile time decisions.

2.2.1 Intent

Different from the Visitor pattern the Static Visitor pattern does not depend on compile time polymorphism at all. It breaks the dependency cycle present in the original Visitor design pattern.

2.2.2 Example

Consider you have a fixed set of classes representing the different entities a file system consists of. One of these classes represents a directory. Directories are Composites [GHJV96i] that contain instances of classes within the given set including directories. A user might want to traverse the directories recursively and apply arbitrary functions on the elements encountered. From a UNIX shell he or she would use `find -exec` to do so.

A traditional implementation looks as shown in Listing 2.3.

Listing 2.3: Classic Visitor with double dispatch

```
class File;
class Directory;

struct VisitorIf {
  virtual ~VisitorIf() {}
  virtual void visit(File &) =0;
  virtual void visit(Directory &) =0;
};

struct FileSystemElementIf {
  virtual ~FileSystemElementIf() {}
  virtual void accept(VisitorIf &) =0;
};

class File : public FileSystemElementIf {
  ...
public:
  void accept(VisitorIf &v) {
    v.visit(*this);
  }
  ...
};

class Directory : public FileSystemElementIf {
  ...
  typedef std::list< FileSystemElementIf * > list_type;
public:
  void accept(VisitorIf &v) {
    v.visit(*this);
    list_type ls;
    ...
    for(list_type::iterator in(ls.begin());ls.end()!=in;++in)
      (*in)->accept(v);
  }
  ...
};
```

The details of `File` and `Directory` do not matter here. They are similar to the implementation shown below in Listing 2.4.

Concrete visitor classes have to realize the interface `VisitorIf`.

It is worth noting that the visitor interface depends on the (incomplete) types of all possible elements the file system can consist of, and that `FileSystemElementIf`, the interface all file system elements realize, depends on the (incomplete) visitor interface. This cyclic dependency can also be seen in the accompanying Figure 2.3 and could hardly be tighter. Adding another file system element class not only requires its definition, but also requires the modification of the visitor interface and thus of all concrete visitors. The latter is a hard task and can even be impossible as the supplier of the file system class hierarchy might not have control over all visitor classes. Therefore this implementation

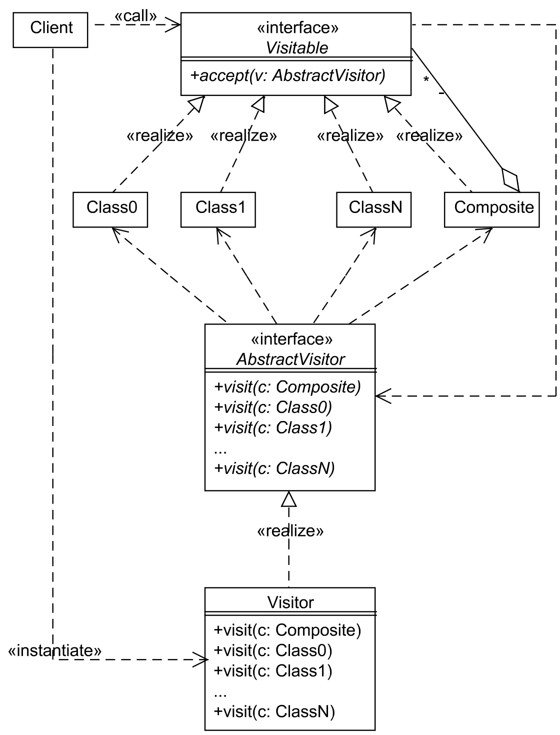

Figure 2.3: Class diagram illustrating the classic Visitor with double dispatch [GHJV96d]

applies only if the class hierarchy to visit is nearly stable. Furthermore strong dependencies can lead to much longer compilation times, if code was changed.

It is also worth noting that this implementation supports double dispatch due to the fact that the function `FileSystemElementIf::accept()` is lately bound by means of the `virtual` specifier. The implementation of `Directory` exploits this fact—it does not depend on any concrete file system element class.

As the hierarchy of all classes implementing `FileSystemElementIf` needs to be stable anyway, however, it might be beneficial to replace runtime by static polymorphism.

2.2.3 Context

A fixed set of classes is given. Some of them are object structures that can aggregate instances of classes from the set and thus instances can be arranged in a hierarchical manner. Changes to this set can nearly be ruled out.

2.2.4 Problem

Different algorithms will be applied to the instances arranged in the hierarchy possibly using different traversal strategies. The algorithms are not known in advance. So it is not an option to add all algorithms to the set of classes given. The problems therefore are as follows: How to *a priori* add minimal functionality to each of the classes the set consists of to allow for maximal extensibility regarding applying arbitrary user defined algorithms to each instance reachable though an instance aggregating others? How to shield the traversal from the user? How to offer typesafe extensibility?

2.2.5 Forces

- A user may want to traverse the object structure both just to accumulate data and to change the elements.

- Dependencies and associations among classes should be kept to a minimum, especially cyclic ones.

2.2.6 Solution

Instead of repeatedly adding functionality to each class the set consists of once and for all equip these classes with a member function template accepting an instance of any class that is a model of some visitor concept. Whenever new algorithms should be applied to the hierarchy of instances these algorithms will have to be represented by an appropriate visitor class. The visitor can differentiate between the different classes of the set by means of overloading.

A first sketch of the solution is shown in Table 2.2.

Table 2.2: Class–Responsibility–Collaboration Cards

Client	
Instantiates	Visitor
Passes Visitor to	ObjectStructure

(a) Client

ClassN	
Offers interface to	Visitor

(b) Class N

ObjectStructure	
Accepts	Visitor
Contains instances of	ClassN
Traverses through its instances of	ClassN

(c) Object Structure

Visitor	
Declares member function overload for each	ClassN
Is model of	VisitorConcept
Potentially accumulates some state	

(d) Visitor

VisitorConcept	
Declares interface to	ObjectStructure

(e) Visitor Concept

Participants

ClassN One class of a bounded and known set of classes. Object Structure aggregates one or more instances of these classes. Each class likely provides an interface that differs from the interfaces of the other classes contained in the set. The Visitor interacts with Class N by calling its member functions.

Client The Client intends to execute a function on all elements directly or indirectly contained within Object Structure. To do so it instantiates a Visitor that represents the function and passes it to Object Structure.

ObjectStructure A special variant of Class N. A collection of instances of Class N and other classes from the well–known, bounded set. Object Structure provides a template member function to accept any Visitor that is model of Visitor Concept. Often this function is responsible to traverse the member instances and call the member function of the Visitor for each instance encountered.

Visitor A model of Visitor Concept that overloads a member function prescribed by the concept for all classes similar to Class N. If some of these classes have a common superclass, then the Visitor might only overload its member function for the superclass.

VisitorConcept All Visitor classes must be models of a Visitor Concept to offer Class N and Object Structure a single way to use Visitors.

Figure 2.4 sketches the participants and their relations to each other.

Dynamics

The Client instantiates a Visitor and passes it to the instance of Object Structure. The Object Structure traverses through its elements and repeatedly and potentially recursively calls the member function on the Visitor instance prescribed by Visitor Concept passing the current element to the Visitor. Because of strong typing the compiler binds this function call early to the appropriate function overload.

The dynamics of Static Visitor is shown in Figure 2.5.

Rationale

It is well known that the application of the Visitor design pattern in its original version introduces cyclic dependencies: The Visitor depends on Class N and its siblings, and every class that accepts a Visitor depends on the Visitor class. Therefore the visitor especially works if the set of classes is fixed and bounded. Then the Visitor helps to add arbitrary functionality to existing classes without the need to modify them.

In the original publication of Visitor polymorphism is used twice: First, it enables double dispatch. The Object Structure can contain elements of arbitrary classes as long as each class implements an interface that enables its instances to accept a Visitor. Second, it makes the accept member function independent of a concrete Visitor class.

Here the second aspect is substituted by static polymorphism.

Now classes accepting visitors do not depend on any visitor interface any more. Instead they accept instances of all visitor classes that are models of the same visitor concept. As no virtual call is involved any more, the traversal through the class hierarchy and the application of the visitor happen without indirection and can be inlined by the compiler.

2.2.7 Resulting Context

The Clients can apply arbitrary functions to the elements of Object Structure without knowledge in how to traverse it. Class N and its siblings do not have to be modified to add functionality common to all of them. For each new task a new Visitor class will be developed.

Pros and Cons

The Static Visitor pattern has the following benefits:

1. *No virtual calls to Visitor.* As the Visitor is statically bound to the parameter of the accept member function of Object Structure and Class N, the calls to the overloaded member functions of the Visitor instance are direct and can be inlined.

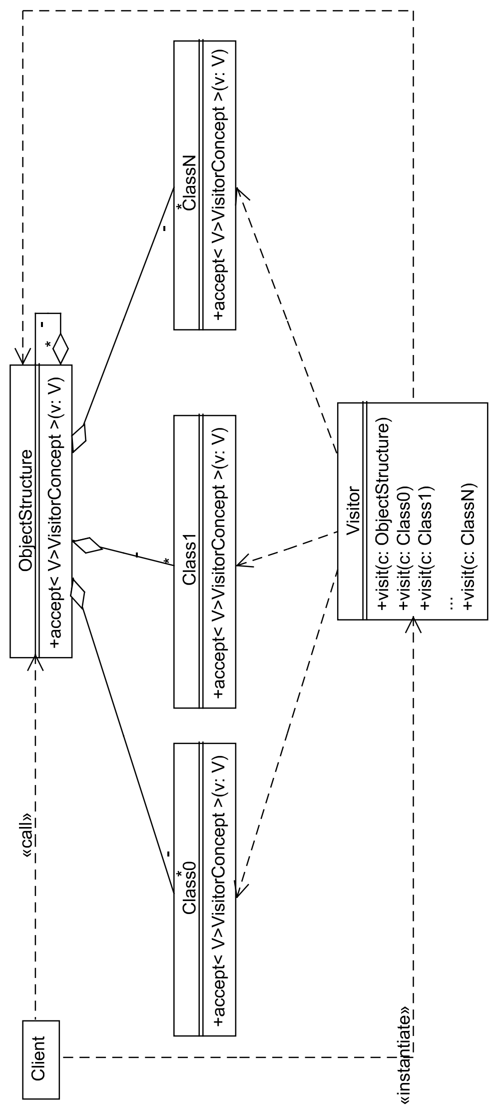

Figure 2.4: Class diagram illustrating Static Visitor

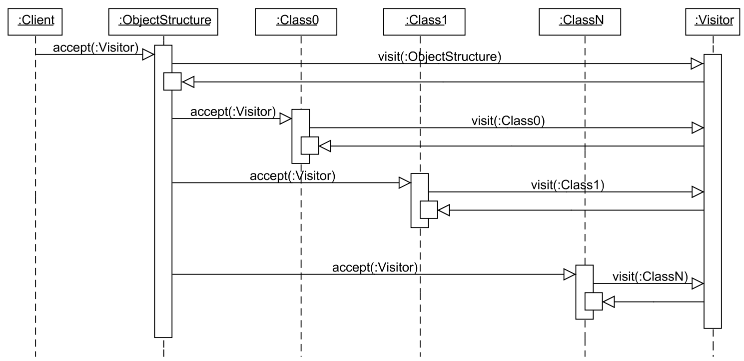

Figure 2.5: Sequence diagram illustrating Visitor

2. *Accept does not depend on Visitor* The accept member functions do not depend on a Visitor interface anymore. Instead they depend on a Visitor Concept.

Additionally to these general pros and cons the following implementation specific ones were identified.

The implementation technique of the Static Visitor pattern shown has the following liabilities:

1. *No double dispatch possible.* As member function templates cannot be declared `virtual`, the Static Visitor does not implement double dispatch. As long as Object Structure is not part of a Framework [RJ98] while placing the responsibility over Class N and Visitor to the user of the Framework this issue is a minor disadvantage only because the benefit of double dispatch is undermined by the fact that for the Visitor pattern the set of classes Object Structure can contain has to be known *a priori* anyway. With Frameworks, however, Object Structure must not depend on any Class N for extensibility reasons, so double dispatch is necessary. Static Frameworks (see Section 3) may point at a way out.

2. *No concept of Visitor.* The Visitors have to be models of the same concept Visitor Concept: They all have to provide member function overloads of the same name. Such concepts cannot currently be expressed in C++. There are matured proposals to overcome this issue, e.g. [DRS06, GS06].

2.2.8 Implementation

This section shows the implementation of the pure Static Visitor design pattern. It is not combined with other variations of the same pattern.

Example Resolved

Listing 2.4 shows the two classes `Directory` and `File`. For simplicity reasons it is assumed that file systems consist of instances of these classes only. In the real UNIX world you would additionally expect classes like `SymbolicLink`, `Device`, and `Process`. `Directory` is a container that can hold an arbitrary number of `Directory` and `File` instances. Instances of both `Directory` and `File` can be asked to tell their size, a property with very different meaning among the different file system objects.

Extensibility is given by the fact that instances of both `File` and `Directory` can be visited by arbitrary visitors.

Listing 2.4: Static Visitor

```
class File {
  const std::string name_;
public:
  explicit File(const char name[]) : name_(name) {}
  template< typename Visitor > void accept(Visitor &v) {
    v.visit(*this);
  }
  std::size_t fileSize() const {
    stat s;
    stat(name_,&s);
    return s.st_size;
  }
};

class Directory {
  const std::string name_;
  // Separate list types - thus double dispatch is not necessary
  typedef std::list< Directory > dir_list;
  typedef std::list< File > file_list;
public:
  explicit Directory(const char name[]) : name_(name) {}
  template< typename Visitor > void accept(Visitor &v) {
    v.visit(*this);
    dir_list dir;
    file_list file;
    DIR *const dirp=opendir(name_);
    dirent *direntp=0;
    while((direntp=readdir(dirp))) {
      stat s;
      stat(direntp->d_name,&s);
      if(S_ISDIR(s.st_mode))
        dir.push_back(static_cast< Directory >(direntp->d_name));
      else if(S_ISREG(s.st_mode))
        file.push_back(static_cast< File >(direntp->d_name));
    }
    closedir(dirp);
    for(dir_list::iterator in(dir.begin());dir.end()!=in;++in)
      in->accept(v);
    for(file_list::iterator in(file.begin());file.end()!=in;++in)
      in->accept(v);
  }
  std::size_t size() const {
```

```
    stat s;
    stat(name_,&s);
    return s.st_size;
  }
};
```

Listing 2.5 shows two visitors. Both do not modify the instances visited. `AccumulateSize` sums up the sizes of all nodes encountered. `Count` simply counts all nodes regardless of their type. Both visitors carry some state. This state can only be fed back to the client because the `accept<>()` member function templates in the listing before pass the visitor by reference. Another implementation option of the `accept<>()` member function templates was to return the visitor taken by value as the algorithms of the C++ Standard Library do.

Listing 2.5: Two examples for visitors

```
class AccumulateSize {
  std::size_t size_;
public:
  AccumulateSize() : size_(0) {}
  void visit(const File &f) {
    size_+=f.fileSize();
  }
  void visit(const Directory &d) {
    size_+=d.size();
  }
  std::size_t size() const {
    return size_;
  }
};

class Count {
  std::size_t number_of_elements_;
public:
  Count() : number_of_elements_(0) {}
  template< typename FileSystemElement >
  void visit(const FileSystemElement &) {
    ++number_of_elements_;
  }
  std::size_t getNumber_of_elements() const {
    return number_of_elements_;
  }
};
```

Implementing `visit()` as a template member function as with `Count` additionally breaks the dependency of the visitor class from the classes of the elements visited. However, this only works if the visitor does not really access the elements as in the example or if the element classes all model the same concept which is not the case in the example, because the member functions returning a size have different names in `File` and `Directory`.

2.2.9 Variants

A particularly attractive variant is the combination with a variation of Acyclic Visitor [Mar98], [Ale03, pp 322–328]. It moves the dependency of the declaration

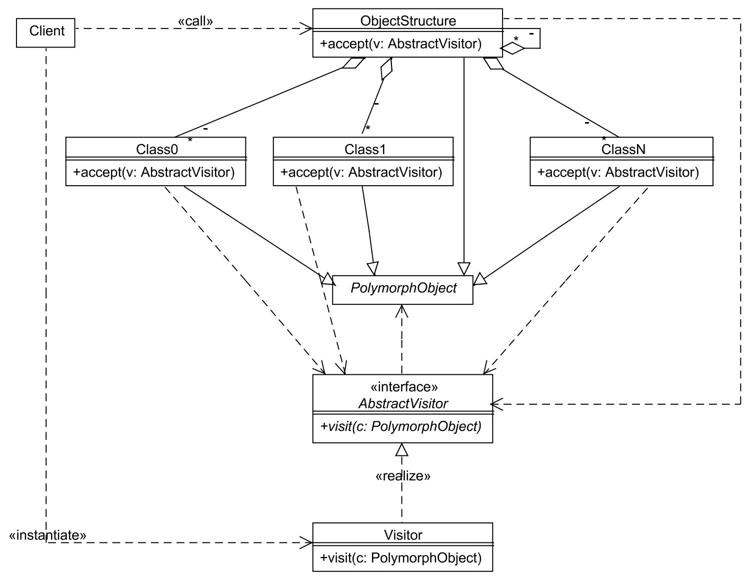

Figure 2.6: Class diagram illustrating Yet Another Acyclic Visitor

of Visitor from the class hierarchy to the definition of Visitor. To accomplish this the Visitor uses `dynamic_cast<>()` to convert a reference to a common superclass to a reference to one of the classes the hierarchy consists of. Combining Static Visitor with this variant of Acyclic Visitor can further reduce the dependencies between the interfaces of the visitor classes and the classes visited.

Figure 2.6 and Listings 2.6 and 2.7 sketch this variant.

Listing 2.6: Static Visitor enabling the use of Acyclic Visitors

```
struct PolymorphObject {
  virtual ~PolymorphObject() =0 {}
};

class File : public PolymorphObject {
  ...
public:
  template< typename Visitor > void accept(Visitor &v);
  ...
};

class Directory : public PolymorphObject {
  ...
public:
  template< typename Visitor > void accept(Visitor &v);
  ...
};
```

The details of `File` and `Directory` do not matter here. They are similar to the implementation shown above in Listing 2.4. The difference is that both specialize the nearly trivial class `PolymorphObject` now. This is done for the sole purpose of enabling polymorphism, as in C++ there is no standard root class or interface of all classes like e.g. `java.lang.Object` in Java [AGH01, pp 47,110–112].

The modified visitor class example exploits this property to move the dependencies from concrete file system element classes from its header file to its implementation file only.

Listing 2.7: An example for an Acyclic Visitor

```
// Header file
class AccumulateSize {
  std::size_t size_;
public:
  AccumulateSize();
  void visit(const PolymorphObject &);
  std::size_t size() const;
};

// Implementation file
AccumulateSize::AccumulateSize() : size_(0) {}

void AccumulateSize::visit(const PolymorphObject &o) {
  if(const File *const f=dynamic_cast< const File * >(&o)) {
    size_+=f->fileSize();
    return;
  }
  if(const Directory *const d
        =dynamic_cast< const Directory * >(&o)) {
    size_+=d->size();
    return;
  }
  // Ignore instance of unknown file system element type
}

std::size_t  AccumulateSize::size() const {
  return size_;
}
```

Without establishing the relation between the classes representing file system elements and the common base class with at least one virtual member function the visitor classes could not benefit from `dynamic_cast<>()`.

Compared to the original Visitor design pattern the virtual `visit()` member functions and the interface `VisitorIf`, which declares them, became replaced by static polymorphism in terms of the template member functions `accept<>()`, that now take any visitor that is a model of a visitor concept. The overloading of the `visit()` member functions became replaced by dynamic polymorphism in terms of `dynamic_cast<>()`. So the original pattern is nearly turned upside down—runtime polymorphism becomes static polymorphism and vice versa—leading to a vast reduction of bidirectional dependencies between the visitor interface and the class hierarchy visited.

2.2.10 Known Uses

Examples of Static Visitor can be found in existing software.

Boost.Variant

Boost.Variant [FM] represents a C++ container that holds exactly one value of arbitrary type. This kind of classes is often used when interfacing a strongly typed language like C++ with a scripting language or with a remoting library like MS COM. Boost.Variant provides both a runtime type checked access and a compile time type checked access to the value stored. The latter uses the Static Visitor pattern by means of the `apply_visitor<>()` member function template that fulfills the purpose `accept<>()` fulfills above.

Boost Graph library

The Boost Graph library [SLL, SLL02] defines several Visitor Concepts. There is no need for a common Visitor base class for each concept, because the Static Visitor pattern is used. For example the template function `boost::depth_first_search<>()` accepts all Visitor classes that are models of the DFSVisitor concept and plays the role of the `accept<>()` member function template in the description above.

2.2.11 Related Patterns

The Static Strategy pattern (see Section 2.1) also inverts control flow.

The Static Visitor is an Internal resp. Passive Iterator [GHJV96h, pp 339–340,348–352] executing different Static Strategies depending on overloading.

The Visitor design pattern uses (runtime) polymorphism to allow for substitution of a concrete visitor by another implementation. The Static Visitor pattern is its static counterpart.

2.3 Static Adapter

A class based pattern to map types to behavior. The Static Adapter pattern helps decouple an application from a single platform. It ensures that all adapters reliably model the same concept.

2.3.1 Also known as

Wrapper Facade [SSRB02f, pp 66–67]

2.3.2 Intent

Adapt a series of different interfaces to a common interface. The decision which interface is actually to be adapted can be done at compile time. It has to be ensured that regardless of which interface is going to be adapted each adaptation results in the same interface not to break client code.

2.3.3 Example

Consider that a library will be built to abstract from different concurrency control primitives on different platforms. For example there will be a class `ReadersWriter_Mutex` providing the member functions `readAcquire()`, `writeAcquire()`, and `release()`. The implementation of the class translates platform specific interfaces—most likely imperative and not object oriented ones—into an object oriented interface common to a variety of platforms. The constructor will perform initialization of the platform specific primitive if necessary, and the destructor will free resources again if required.

Traditionally such code would either use the Adapter design pattern [GHJV96b], or the original Wrapper Facade pattern is used with conditional compilation, i.e. the interface and especially the implementation is interspersed with preprocessor instructions as shown in Listing 2.8.

Listing 2.8: Using conditional compilation to adapt platform specific readers / writer locks to a uniform interface

```
class ReadersWriter_Mutex {
  #if defined(_WIN32)
    CRITICAL_SECTION lock_;
  #elif defined(UNIX)
    pthread_rwlock_t lock_;
  #else /* defined(_WIN32) */
    #error ReadersWriter_Mutex: Fatal error: Platform not supported.
  #endif /* defined(_WIN32) */
  // No copy allowed, therefore private and declared only
  ReadersWriter_Mutex(const ReadersWriter_Mutex &);
  // No assignment allowed, therefore private and declared only
  ReadersWriter_Mutex &operator=(const ReadersWriter_Mutex &);
public:
  ReadersWriter_Mutex() {
    #if defined(_WIN32)
      InitializeCriticalSection(&lock_);
    #elif defined(UNIX)
      if(pthread_rwlock_init(&lock_,NULL))
        throw std::runtime_error(
          "Call␣to␣\"pthread_rwlock_init()\"␣failed."
        );
    #endif /* defined(_WIN32) */
  }
  ~ReadersWriter_Mutex() {
    #if defined(_WIN32)
      DeleteCriticalSection(&lock_);
    #elif defined(UNIX)
      assert(!pthread_rwlock_destroy(&lock_));
    #endif /* defined(_WIN32) */
  }
  void readAcquire() {
    #if defined(_WIN32)
      EnterCriticalSection(&lock_);
```

```
    #elif defined(UNIX)
      if(pthread_rwlock_rdlock(&lock_))
        throw std::runtime_error(
          "Call␣to␣\"pthread_rwlock_rdlock()\"␣failed."
        );
    #endif /* defined(_WIN32) */
  }
  void writeAcquire() {
    #if defined(_WIN32)
      EnterCriticalSection(&lock_);
    #elif defined(UNIX)
      if(pthread_rwlock_wrlock(&lock_))
        throw std::runtime_error(
          "Call␣to␣\"pthread_rwlock_wrlock()\"␣failed."
        );
    #endif /* defined(_WIN32) */
  }
  void release() {
    #if defined(_WIN32)
      LeaveCriticalSection(&lock_);
    #elif defined(UNIX)
      if(pthread_rwlock_unlock(&lock_))
        throw std::runtime_error(
          "Call␣to␣\"pthread_rwlock_unlock()\"␣failed."
        );
    #endif /* defined(_WIN32) */
  }
};
```

For every member function and for the attribute conditional compilation is used here.

Preprocessor instructions are somewhat outside of the programming language used, however. This solution is not very elegant, the compiler can not assist much in detecting errors, and maintenance likely becomes a nightmare. So the goal is to reduce conditional compilation to a minimum.

2.3.4 Context

Different platforms potentially adhere different standards. A mapping was defined to provide a common programming interface, sometimes referred to as a portable runtime or a Wrapper Facade.

2.3.5 Problem

How can the compiler(s) guarantee, that all implementations for different platforms model the same concept (e.g. provide the member functions `readAcquire()`, `writeAcquire()`, and `release()`)? The Wrapper Facade pattern suggests a way to provide a common abstraction of platform specific interfaces to user code, but does not discuss in detail how to adapt this abstraction to more than one platform [BK03].

Table 2.3: Class–Responsibility–Collaboration Cards

Client	
Instantiates template	PlatformType, Static Adapter, Specialization of Member Functions

(a) Client

PlatformType	
Binds template argument of	Static Adapter
Fixed for a single	Client
Provides interface to	Specialization of Member Functions

(b) Platform Type

SpecializationOfMemberFunctions	
Adapts	PlatformType
Statically implements	Static Adapter

(c) Specialization of Member Functions

StaticAdapter	
Provides platform agnostic interface to	Client

(d) Static Adapter

2.3.6 Forces

- The more platforms to be supported and the more degrees of freedom static configuration by means of the domain specific language available, ensuring that each variant compiles and works becomes a nightmare without processes and tools that help.
- Explicit representation of (static) configurability makes the code more understandable.
- Dynamic configuration by means of the Adapter design pattern is not an option for code that would benefit from early binding and inlining.

2.3.7 Solution

Static polymorphism can be used to statically configure the Wrapper Facade to choose the correct, platform specific implementation. The configuration has to be restricted to the member functions and not to the whole class to ensure that the interface remains identical on all platforms.

A first sketch of the solution is shown in Table 2.3.

Participants

Client Client code instantiates the Static Adapter template for a Platform Type.

PlatformType A low Layer [BMR+00a], likely with an imperative interface. The interfaces handling different Platform Types might differ significantly. Platform Types often represent entities that can be acquired and then released again. Such entities are referred to as resources. A Platform Type remains fixed during runtime of the application and most likely for even much longer periods.

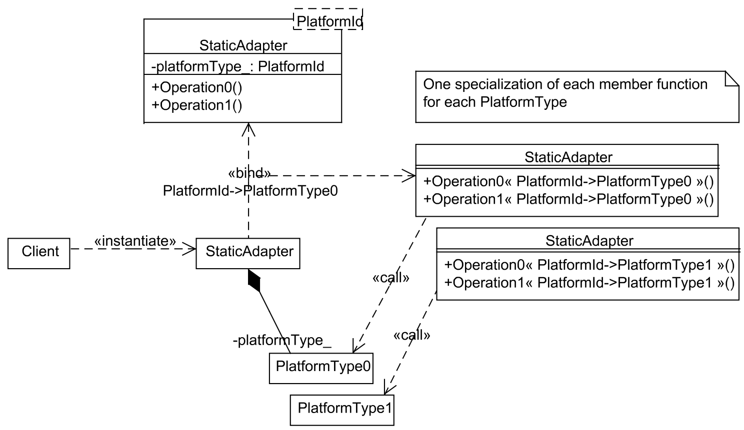

Figure 2.7: Class diagram illustrating Static Adapter

SpecializationOfMemberFunctions For each Platform Type all member functions of the Static Adapter are specialized and defined.

StaticAdapter A class template declaring the platform independent static interface of the Wrapper Facade. The member functions are declared, but not defined. Therefore the template parameter is not restricted to a certain concept.

Figure 2.7 sketches the participants and their relations to each other.

Dynamics

The Client binds the template parameter of Static Adapter to an appropriate Platform Type. Most often it does so by a `typedef`. The resulting class will be instantiated then. Within the same translation unit there are declarations of Specializations of Member Functions. During binding of template parameters the compiler records the respective symbols to the object code, and during link editing the linker will take the appropriate definitions of Specialization Of MemberFunctions.

Rationale

If runtime efficiency is critical dynamic configuration would lead to systems with virtual calls and less opportunities for inlining. Given that there is no need to let the configuration depend on e.g. user input or configuration files, then static configuration can solve these efficiency problems. Static polymorphism is used to let take Static Adapter possibly totally different implementations from one Platform Type to another.

This pattern uses explicit specialization of member functions, not of the whole class template. This ensures that the interface of the Wrapper Facade is the same for each Platform Type. By some sense Static Adapter is a concept the Client can trust in, and the instantiated template which binds a Platform Type to the template argument of Static Adapter and implements the member functions by means of their Specializations is a model of the concept.

This design pattern is not restricted to the implementation technique promoted. Other techniques are elaborated on in Sections 2.3.10 and 2.3.11, respectively.

2.3.8 Resulting Context

For each Platform Type there is a Wrapper Facade. The compiler guarantees that these Wrapper Facades do not differ regarding to their interfaces.

Pros and Cons

The Static Adapter pattern has the following benefits:

1. *Runtime efficiency.* As with Wrapper Facade this pattern tries to keep the platform abstraction Layer as thin as possible.

2. *Cross platform contract.* Static Adapter provides a cross platform contract. The Client can trust in the concept defined by the Static Adapter.

The Static Adapter pattern has the following liability:

1. *Static configuration itself must be portable.* The Static Adapter pattern presumes a portable technique for static configuration. The more platforms have to be supported the more restrictions this requirement will impose.

Additionally to these general pros and cons the following implementation specific ones were identified.

The implementation technique of the Static Adapter pattern shown has the following benefits:

3. *More than one specialization per platform.* This special technique allows for more than a single specialization for specific platforms, while on other platforms there might be only a single specialization. See Section 2.3.9 for an example. Providing a toolset from which a tool can be chosen by means of a simple `typedef` supports delaying irreversible decisions, an Agile and lean principle [PP03d, Zan02].

4. *One language only.* An implementation based on metaprogramming techniques of the programming language used anyway means that all can be done within a single environment. There is no need to use another tool to perform static configuration.

The implementation technique of the Static Adapter pattern shown has the following liability:

2. *For some compilers everything must be inlined.* Then this technique reveals implementation details in header files. This might not be appropriate. There are notable exceptions among the compilers where in the case of explicit specializations the definitions do not have to be inlined anymore and can go into separate implementation files because of increasing support of [Ame03, 14.7.3/5], however.

2.3.9 Implementation

Static polymorphism is implemented using specialization of member function templates.

Example Resolved

Listing 2.9 shows two different implementations of a Wrapper Facade. The implementation for `CRITICAL_SECTION`s works on MS Win32 and does not make a difference between reading and writing. The implementation for `pthread_rwlock_t` works on POSIX 1003.1c compliant systems and treats reading and writing differently.

Listing 2.9: Statically adapting platform specific readers / writer locks to a uniform interface

```
// Header file
template< typename Lock > class ReadersWriter_Mutex {
  Lock lock_;
  // No copy allowed, therefore private and declared only
  ReadersWriter_Mutex(const ReadersWriter_Mutex &);
  // No assignment allowed, therefore private and declared only
  ReadersWriter_Mutex &operator=(const ReadersWriter_Mutex &);
public:
  ReadersWriter_Mutex();
  ~ReadersWriter_Mutex();
  void readAcquire();
  void writeAcquire();
  void release();
};

// Specializations of member functions
#ifdef _WIN32
template<>
ReadersWriter_Mutex< CRITICAL_SECTION >
  ::ReadersWriter_Mutex();

template<>
ReadersWriter_Mutex< CRITICAL_SECTION >
  ::~ReadersWriter_Mutex();

template<>
void ReadersWriter_Mutex< CRITICAL_SECTION >::readAcquire();

template<>
void ReadersWriter_Mutex< CRITICAL_SECTION >::writeAcquire();
```

```
template<>
void ReadersWriter_Mutex< CRITICAL_SECTION >::release();
#endif /* defined(_WIN32) */

#ifdef UNIX
template<>
ReadersWriter_Mutex< pthread_rwlock_t >
  ::ReadersWriter_Mutex();

template<>
ReadersWriter_Mutex< pthread_rwlock_t >
  ::~ReadersWriter_Mutex();

template<>
void ReadersWriter_Mutex< pthread_rwlock_t >::readAcquire();

template<>
void ReadersWriter_Mutex< pthread_rwlock_t >::writeAcquire();

template<>
void ReadersWriter_Mutex< pthread_rwlock_t >::release();
#endif /* defined(UNIX) */

// Implementation file
#ifdef _WIN32
ReadersWriter_Mutex< CRITICAL_SECTION >
  ::ReadersWriter_Mutex() {
  InitializeCriticalSection(&lock_);
}

ReadersWriter_Mutex< CRITICAL_SECTION >
  ::~ReadersWriter_Mutex() {
  DeleteCriticalSection(&lock_);
}

void ReadersWriter_Mutex< CRITICAL_SECTION >::readAcquire() {
  EnterCriticalSection(&lock_);
}

void ReadersWriter_Mutex< CRITICAL_SECTION >::writeAcquire() {
  EnterCriticalSection(&lock_);
}

void ReadersWriter_Mutex< CRITICAL_SECTION >::release() {
  LeaveCriticalSection(&lock_);
}
#endif /* defined(_WIN32) */

#ifdef UNIX
ReadersWriter_Mutex< pthread_rwlock_t >
  ::ReadersWriter_Mutex() {
  if(pthread_rwlock_init(&lock_,NULL))
    throw std::runtime_error(
      "Call␣to␣\"pthread_rwlock_init()\"␣failed."
    );
}
```

```
ReadersWriter_Mutex< pthread_rwlock_t >
  ::~ReadersWriter_Mutex() {
  assert(!pthread_rwlock_destroy(&lock_));
}

void ReadersWriter_Mutex< pthread_rwlock_t >::readAcquire() {
  if(pthread_rwlock_rdlock(&lock_))
    throw std::runtime_error(
      "Call␣to␣\"pthread_rwlock_rdlock()\"␣failed."
    );
}

void ReadersWriter_Mutex< pthread_rwlock_t >::writeAcquire() {
  if(pthread_rwlock_wrlock(&lock_))
    throw std::runtime_error(
      "Call␣to␣\"pthread_rwlock_wrlock()\"␣failed."
    );
}

void ReadersWriter_Mutex< pthread_rwlock_t >::release() {
  if(pthread_rwlock_unlock(&lock_))
    throw std::runtime_error(
      "Call␣to␣\"pthread_rwlock_unlock()\"␣failed."
    );
}
#endif /* defined(UNIX) */
```

With this code in place the configuration consists of a simple `typedef ReadersWriter_Mutex< platformLock > rw_mutex_t;` where `platformLock` is one of the locks the template is specialized for as the class template lacks a default implementation. As you can see the MS Windows implementation does not use readers / writer locking in its implementation; with the above approach it is also possible to add another specialization for the UNIX platform family for `pthread_mutex_t` which does not use readers / writer locking in its implementation, too. Providing multiple specializations for a single platform can be beneficial in cases where special implementations have side effects not appropriate in certain situations. An example for this was an MS Windows emulation for real readers / writer locks that allocates handles. Each use of such locks in fields of unknown size must be avoided not to run out of handles, so in this case you are better off using `CRITICAL_SECTION`s.

This technique results in a great reduction of preprocessor instructions compared to Listing 2.8. The remaining conditional compilation code serves for two purposes: First, the correct `typedef` has to be selected. This could alternatively be done by the Static Abstract Type Factory pattern proposed in Section 2.4 as shown in Listing 2.10. The second purpose is to hide platform specific types from the compilers on all other platforms—otherwise compilation errors are likely.

2.3.10 Variants

In languages which have the distinction between header and source files one header and as many source files can be defined as platforms have to be supported. The build mechanism, e.g. Make, then determines which of the source files to compile. If more than one implementation exists for a Platform Type, then the decision of which one to take can be deferred until link–edit time.

A macro processor like M4 can be used to generate platform specific code.

Instead of an Adapter style implementation [BK03] suggests the use of Static Strategy (see Section 2.1) to solve the same problem.

2.3.11 Known Uses

Examples of Static Adapter can be found in existing software. Though none of the following libraries uses the implementation technique presented in Section 2.3.9, nevertheless all of them solve the problem to statically adapt Wrapper Facades to a variety of different platforms.

ACE

The ADAPTIVE Communication Environment (ACE) consists of multiple Layers. Wrapper Facades build the lowest Layer. ACE supports many platforms and is written in C++. The Wrapper Facades are organized as one header and one implementation file each. The platform differences are implemented using conditional compilation within the bodies of the member functions. Configuration is done by preprocessor constants defined in a central header file included by all files. A header file appropriate for the platform given has either to be manually declared as the central header file by the user before ACE is going to be compiled or can be generated using GNU Autoconf.

APR

The Apache Portable Runtime (APR) consists of Wrapper Facades. It supports BeOS, Novell Netware, IBM OS/2, UNIXes, and MS Windows and is written in C. Each Wrapper Facade is declared in one header file. For each platform supported there is a corresponding implementation file. Which implementation file to compile and link is chosen by means of the Python script `gen_build.py` called from `buildconf`. After this static configuration step GNU Autoconf configures remaining degrees of freedom. Then APR can be compiled, linked, and installed using Make.

Boost.Threads

The Boost project contains a set of Wrapper Facades for multithreading [Kem]. It supports POSIX, Apple OS X and MS Win32 and is written in C++. Platform independence is gained by conditional compilation within the bodies of member functions. The static configuration is done by Perforce Jam files, which force appropriate preprocessor constants to be set.

GTK+ GLib

The GTK+ library forms the basic layer of Gimp and Gnome. Its GLib base Layer is a counterexample for Static Adapter, as the Adapter pattern is used instead.

Loki<library>

The Loki library contains a set of multithreading Wrapper Facades [Ale03, pp 391–402]. It supports POSIX and MS Win32 and is written in C++. The code is completely inlined within a single header file. Conditional compilation determines which implementation to take. The configuration relys on preprocessor constants set differently by the compilers on different platforms or set within platform specific standard header files.

NSPR

The Netscape Portable Runtime consists of Wrapper Facades. It supports POSIX and many other flavours of UNIX, Apple Mac and MS Win32 and is written in C. NSPR is implemented using a mixed approach: First, for each Wrapper Facade there are one header file and many implementation files for different platforms. Second, further static configuration is established by means of conditional compilation within an implementation file appropriate to the platform. A GNU Autoconf script both sets preprocessor constants for conditional compilation and Make variables to compile and link the correct implementation file.

SAL

Open Office System Abstraction Layer (SAL) consists of Wrapper Facades. It supports both UNIX systems which adhere to the POSIX standards and MS Windows. Each Wrapper Facade is splitted into two halfs. The lower level C Layer consists of one header and two implementation files each. A Perl build mechanism determines which of the implementation files is compiled and linked. On top of this a thin and completely inlined C++ Layer establishs object oriented abstractions.

2.3.12 Related Patterns

The Wrapper Facade pattern proposes a way to abstract from a specific platform by defining an interface common to all platforms. The implementation translates imperative application programming interfaces into an object oriented representation and unifies return values and the signalization of error conditions. The description of Wrapper Facade states the need for such an abstraction layer, but it does not discuss ways to ensure that exactly the same interface is implemented for each platform.

The Adapter design pattern uses (runtime) polymorphism to allow for changes of concrete adapters. The compiler guarantees that each adapter implements the same interface. The Static Adapter pattern is its static counterpart.

2.4 Static Abstract Type Factory

A class based pattern to map types to types.

2.4.1 Also known as

Generator [CE00, pp 397–501]
Type Selection [Ale03, pp 65–67]
Type Traits [Ale03, pp 74–83], [VJ03c]

2.4.2 Intent

The Static Abstract Type Factory provides an extensible means to associate expressions on the level of the domain specific language with application data types.

2.4.3 Example

Different platforms provide different data types for basically the same entity. Mutual exclusion locks are represented by the type `pthread_mutex_t` on a POSIX 1003.1c compliant UNIX, on MS Win32 `CriticalSection` can be taken. Depending on an expression in the domain specific language the correct type should be chosen while at the same time hiding it from application code behind a unified type name.

A traditional approach was to implement one header file for each platform, each defining the same type names. Either before compilation one of these header files has to be renamed to a predefined file name that is used in the `include` preprocessor directives, or conditional compilation is used to include the appropriate header file into the application code. This approach has the disadvantage that it only works if there is no overlap between the platforms, as only a single header file can be included at the same time to avoid multiple definitions of the same type name. Otherwise it results in duplication of large parts of these header files.

2.4.4 Context

A domain specific language is given. The application to be build for a special static configuration will consist of types, data, and behavior.

Table 2.4: Class–Responsibility–Collaboration Cards

Configuration	
Determines a static configuration	

(a) Configuration

Client	
Instantiates template	StaticAbstractTypeFactory, SpecializationOfClassTemplate, Configuration

(b) Client

SpecializationOfClassTemplate	
Is model of	StaticAbstractTypeFactoryConcept
Specialized for	Configuration
Specializes	StaticAbstractTypeFactory
Defines unified name for type	

(c) Specialization of Class Template

StaticAbstractTypeFactory	
Exists just to enable specializations	

(d) Static Abstract Type Factory

StaticAbstractTypeFactoryConcept	
Declares interface to	Client

(e) Static Abstract Type Factory Concept

2.4.5 Problem

How to associate application data types to the different static configurations? How to encapsulate variation in types?

2.4.6 Forces

- The association of a certain static configuration to application properties is unidirectional.
- The domain specific language should be agnostic about these associations.
- The association mechanism should be extensible.

2.4.7 Solution

Static polymorphism can be used to statically configure `typedef` members of a class template. For this to happen specializations of the class template are defined representing the associations resulting from different static configurations.

A first sketch of the solution is shown in Table 2.4.

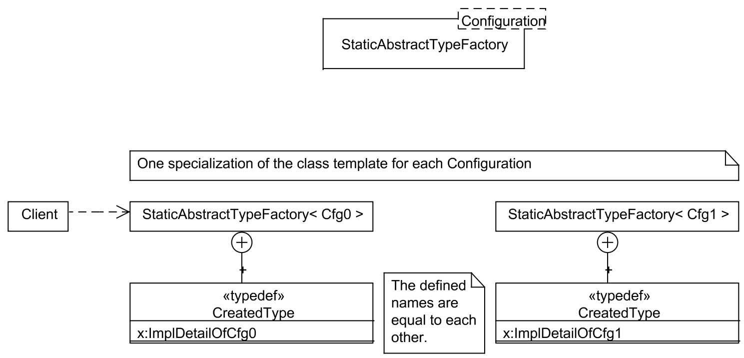

Figure 2.8: Class diagram illustrating Static Abstract Type Factory

Participants

Configuration An expression in the domain specific language to represent a special static configuration.

Client Client code instantiates the Static Abstract Type Factory template for a Configuration. It then uses one of its member types or type definitions.

SpecializationOfClassTemplate For each Configuration supported one specialization of Static Abstract Type Factory exists. As a model of Static Abstract Type Factory Concept it defines member types and provides them under a unified type name to the Client.

StaticAbstractTypeFactory A class template just for the sake to define specializations.

StaticAbstractTypeFactoryConcept Every Specialization of Class Template must define the same type names given by this concept to offer a consistent interface to the Client.

Figure 2.8 sketches the participants and their relations to each other.

Dynamics

The Client binds the template parameter of Static Abstract Type Factory to an appropriate Configuration. Most often it does so by a `typedef`. Within the same translation unit there are Specializations of Class Template. During binding the compiler takes the appropriate specialization instead of the more general Static Abstract Type Factory template. The Client then uses the member types defined within Specialization of Class Template to instantiate them.

Rationale

As all Specializations of Class Template provide the same member type name for potentially different types which depend on Configuration, the implications of a certain configuration can be hidden from the Client. Static Abstract Type Factory associates a static configuration to a configuration specific type. This association is extensible in two ways: First, further specializations can be added to support more configurations. Second, this pattern allows to associate any number of configuration dependend types with a static configuration by adding either another StaticAbstractTypeFactory and appropriate specializations or another member type or type name to all existing specializations of a StaticAbstractTypeFactory.

2.4.8 Resulting Context

The Client can ask the Static Abstract Type Factory for a type passing an expression in the domain specific language and does not need to care about the details. The Static Abstract Type Factory maps these configuration expressions to appropriate types.

Pros and Cons

The Static Abstract Type Factory pattern has the following benefits:

1. *Arbitrarily complex mappings at compile time.* This pattern allows to perform arbitrarily complex mappings from a representation of a static configuration to types at compile time.

2. *Extensibility.* It is easy to add new specializations for new static configurations.

3. *Parallel usage possible.* It is possible to use multiple specializations for different configurations in parallel in the same file.

Additionally to these general pros and cons the following implementation specific ones were identified.

The implementation technique of the Static Abstract Type Factory pattern shown has the following liabilities:

1. *Inheritance relations among Configuration not considered.* Say you organize your domain specific classes in a hierarchy. A `Linux` and a `SunSolaris` class may inherit from a `Unix` class. If a template specialization exists for `Unix`, but not for `Linux`, then the lookup of template specializations for configuration `Linux` will result in the non specialized class template, not in the specialization for `Unix`. The need to also specialize the class template for `Linux` and `SunSolaris` will probably result in double work.

2. *No concept of Specialization of Class Template.* The specializations of StaticAbstractTypeFactory have to be models of the same concept: They

all have to provide the same member types or type names. Such concepts cannot currently be expressed in C++. There are matured proposals to overcome this issue, e.g. [DRS06, GS06].

2.4.9 Implementation

Here the whole class template is going to be specialized. In fact the default class template can be trivial. This Abstract Factory [GHJV96a] depends on static configuration and creates types.

Example Resolved

Listing 2.10 proposes the class template `Multithreading<>` that can be instantiated for either `MSWin32` or `Unix`. Depending on the template instantiation the member type `Multithreading<>::rw_lock` is another name for either `CriticalSection` or `pthread_rwlock_t`. `Multithreading<>` could be extended to also hold type definitions for other types of the multithreading domain, e.g. condition variables, thread identifiers, semaphores, and keys identifying thread local storage.

Listing 2.10: Portable association of an operating system with certain platform specific types combined with Static Adapter (see Section 2.3)

```
// DSL
struct MSWin32 {};
struct Unix {};

template< typename OperatingSystem > struct Multithreading {
};

// Specializations of class template
#ifdef _WIN32
typedef MSWin32 OS;

template<> struct Multithreading< MSWin32 > {
  typedef CriticalSection rw_lock;
  // Other types...
};
#endif /* defined(_WIN32) */

#ifdef UNIX
typedef Unix OS;

template<> struct Multithreading< Unix > {
  typedef pthread_rwlock_t rw_lock;
  // Other types...
};
#endif /* defined(UNIX) */

...

typedef Multithreading< OS > multithreading_type;

```

```
...

// Apply Static Adapter on a static type,
// which real name is hidden from code
ReadersWriter_Mutex< multithreading_type::rw_lock > rw_mutex;
```

2.4.10 Variants

The technique of specialization of the class template can also be used to let a class template define different values to a member `enum` for its specializations and thus map types to integer constants. Often a standard value will then be defined by the class template, which will be overridden for certain template arguments by means of specializations. This is the most popular meaning of a Trait. The technique can similarly be modified to map types to behavior; `std::numeric_limits<>` from the C++ Standard Library and Static Adapter (see Section 2.3) are examples for this case; as pointed out in Section 2.3.7 the mapping of types to behavior already lets you represent concepts in C++, which is not the case with mappings to types or numbers.

The injection of members into class templates and its specializations can also be performed by `public` inheritance instead of explicit definition.

Templates can also be defined with integral template parameters instead of type parameters. Using specializations on certain values integers can be mapped to types, numbers, or behavior, respectively.

2.4.11 Known Uses

Examples of Static Abstract Type Factory can be found in existing software.

Boost.TypeTraits

Boost.TypeTraits [AAC+] provide both class templates to get meta information on types and class templates to transform types. The first kind of templates works with explicit specialization and returns integral constants, while the second kind works with partial specialization and contains member type definitions.

C++ `std::iterator_traits<>`

The C++ way of Iterators [GHJV96h] provides a mechanism to statically gather information on e.g. the type an Iterator points to by means of the class template `std::iterator_traits<>`. For most Iterator types the default implementation of this class template will fit. If not, `std::iterator_traits<>` can be explicitly or partially specialized on the type of the uncommon Iterator. The C++ standard provides such a partial specialization for pointers to arbitrary types.

The Matrix Template Library

The Matrix Template Library [Sie99], [CE00] uses Type Generators to provide tools for linear algebra. The matrix types are the result of static configuration with many degrees of freedom. The client can request e.g. full or sparse matrix types to be generated at compile time.

2.4.12 Related Patterns

The Abstract Factory design pattern uses runtime polymorphism to allow for the substitution of a concrete instance factory by another one. The Static Abstract Type Factory pattern uses compile time polymorphism to allow for the substitution of a type factory by another one.

It may well be that in principle we cannot make any machine the elements of whose behavior we cannot comprehend sooner or later. This does not mean in any way that we shall be able to comprehend these elements in substantially less time than the time required for operation of the machine, or even within any given number of years or generations.

NORBERT WIENER [Wie60, p 1355]

Chapter 3

Static Framework

Ready–made software artifact designed reusable with help of static patterns

3.1 Intent

The purpose of writing portable code is not only to provide an application for a variety of platforms, but to do so in a way that on each platform certain requirements concerning performance are met. Static Frameworks assist you in writing code that can be adapted more easily to multiple platforms while making sure that on each platform the application can fulfill its original purpose.

3.2 Example

Server design involves decisions on how to deal with concurrent service requests issued by clients. This decision depends on the target platform. Some platforms are good at multiprocessing, some perform better if multithreading is used instead, and other platforms might show their full potential with event based designs. Therefore it does not suffice to treat platform dependencies on a low level Wrapper Facade [SSRB02f] Layer only. Instead experience is made available in terms of Frameworks [RJ98] that use design patterns to allow for adaptation to certain environments. Listing 3.1 shows a simple class that frees the user from the burden of portable thread handling.

Listing 3.1: Black–Box Framework [RJ98]

```
// Header file
extern "C" {
  void *svc_run(void *);
}

```

```
struct Method_Request {
  virtual ~Method_Request();
  virtual void call() =0;
};

class MQ_Scheduler {
  friend void *svc_run(void *);
  struct Impl;
  typedef std::auto_ptr< Impl > impl_type;
  impl_type impl_;
public:
  explicit MQ_Scheduler(size_t);
  ~MQ_Scheduler();
  // Transfers ownership
  void insert(Method_Request *);
};

// Implementation file
struct MQ_Scheduler::Impl {
  volatile bool isActive_;
  Activation_List act_queue_;
  static impl_type createImpl(size_t);
  explicit Impl(size_t high_water_mark)
    : isActive_(true), act_queue_(high_water_mark) {}
  virtual ~Impl() {}
  virtual void createUE(MQ_Scheduler &) =0;
  virtual void joinUE() =0;
};

MQ_Scheduler::MQ_Scheduler(size_t high_water_mark)
  : impl_(Impl::createImpl(high_water_mark)) {
  impl_->createUE(*this);
}

MQ_Scheduler::~MQ_Scheduler() {
  impl_->isActive_(false);
  impl_->joinUE();
}

void MQ_Scheduler::insert(Method_Request *method_request) {
  impl_->act_queue_.insert(method_request);
}

void *svc_run(void *arg) {
  assert(arg);
  MQ_Scheduler::impl_type *impl
    =static_cast< MQ_Scheduler * >(arg)->impl_;
  while(impl->isActive_) {
    Method_Request *mr;
    // Block until the queue is not empty
    impl->act_queue_.remove(&mr);
    try {
      mr->call();
    }
    catch(...) {
    }
    delete mr;
```

```
  }
  return 0;
}

#if defined(_WIN32)

class Win32Impl : public MQ_Scheduler::Impl {
  HANDLE thread_;
  // No copy allowed, therefore private and declared only
  Win32Impl(const Win32Impl &);
  // No assignment allowed, therefore private and declared only
  Win32Impl &operator=(const Win32Impl &);
public:
  Win32Impl(size_t high_water_mark)
    : MQ_Scheduler::Impl(high_water_mark), thread_(0) {}
  void createUE(MQ_Scheduler &sched) {
    thread_=reinterpret_cast< HANDLE >(
              _beginthreadex(0,0,svc_run,&sched,0,0)
            );
    if(!thread_)
      throw std::runtime_error(
        "Call␣to␣\"_beginthreadex()\"␣failed."
      );
  }
  void joinUE() {
    if(thread_)
      if(WAIT_FAILED==WaitForSingleObject(thread_,INFINITE)) {
        throw std::runtime_error(
          "Call␣to␣\"WaitForSingleObject()\"␣failed."
        );
        thread_=0;
      }
  }
};

MQ_Scheduler::impl_type
MQ_Scheduler::Impl::createImpl(size_t high_water_mark) {
  return static_cast< MQ_Scheduler::impl_type >(
    new Win32Impl(high_water_mark)
  );
}

#elif defined(UNIX)

class UNIXImpl : public MQ_Scheduler::Impl {
  pthread_t thread_;
  // No copy allowed, therefore private and declared only
  UNIXImpl(const UNIXImpl &);
  // No assignment allowed, therefore private and declared only
  UNIXImpl &operator=(const UNIXImpl &);
public:
  UNIXImpl(size_t high_water_mark)
    : MQ_Scheduler::Impl(high_water_mark) {}
  void createUE(MQ_Scheduler &sched) {
    if(pthread_create(&thread,0,svc_run,&sched))
      throw std::runtime_error(
        "Call␣to␣\"pthread_create()\"␣failed."
```

```
      );
  }
  void joinUE() {
    if(pthread_join(thread_,0))
      throw std::runtime_error(
        "Call␣to␣\"pthread_join()\"␣failed."
      );
  }
};

MQ_Scheduler::impl_type
MQ_Scheduler::Impl::createImpl(size_t high_water_mark) {
  return static_cast< MQ_Scheduler::impl_type >(
    new UNIXImpl(high_water_mark)
  );
}

#endif /* defined(_WIN32) */
```

The thread function and an opaque argument structure are passed Strategy [GHJV96k] like to the constructor of `Thread_Operation`.

`MQ_Scheduler` is used as an illustration of the Active Object architecture pattern [SSRB02a, p 425]. The client hands ownership over instances of `Method_Request` over to the Active Object, i.e. it passes a pointer to a Command [GHJV96c] to an instance of `MQ_Scheduler`. The scheduler asynchronously executes the Command and deletes it afterwards.

The portability is gained using the Bridge design pattern [GHJV96e]. Even the constructors of `MQ_Scheduler` do not have to know concrete implementation classes, because it delegates the creation of an appropriate implementation to a Factory Method [GHJV96g].

More recent versions of the JAWS Adaptive Web System (JAWS) [SSRB02d, pp 27,47–48], an application closely related to the ADAPTIVE Communication Environment (ACE), is an example for this implementation technique. It uses the Active Object design pattern combined with Bridge. The worker thread design is prescribed by a Strategy. All possible Strategies are derived from a single Abstract Class [Aue95, Woo00]. The base class provides for access to the request processing.

The original `MQ_Scheduler` additionally uses the Template Method design pattern [GHJV96j] to make the loop executed by the worker thread adaptable. In this case starting and stopping threads from within the bodies of the constructor and the destructor of the scheduler can lead to bad surprises that can be solved using a helper class implementing Resource Acquisition is Initialization [Str98, pp 388–393], [Str94, pp 495–497] as shown in [Bac05].

Neither the operating system nor the thread function will change during the life time of `MQ_Scheduler`. In fact, especially the operating system will remain constant during the whole time the application is installed on the particular computer. So there is an option to move the configurability up to the meta level.

3.3 Context

A series of applications share implementation similarities not only on a basic Layer [BMR+00a], but also regarding the interaction of objects. An example of this are TCP/IP servers for different protocols, that likely have similar solutions to the problem how to react upon incoming connections.

3.4 Problem

From analysis through architecture and design to the implementation of the initial system ideas central to the design might have been lost in the final code; these ideas are the reason why the code is how it is, but they might not explicitly be represented within the code. This can make reuse of code hard, if it has to be adapted to a different environment.

3.5 Forces

- Code duplication has to be avoided.
- Sometimes higher Layers must be adaptable.
- Future adaptations might be requested by a customer.
- The code base needs to remain maintainable.
- Some configuration remains fixed during a period often much longer than the runtime of an application.
- Experience should be transformed into ready–made software artifacts, if reuse is likely.

3.6 Solution

Cast the real intent of a software construct into a code representation. Make the abstractions explicit. Raise the level of abstraction from a pure series of commands to a function or a function object, potentially an Active Object [SSRB02a]. Develop a Framework that is configurable in two ways: Enable static configurability of user code supplied as a function or function object by means of a Static Strategy (see Section 2.1) or Static Visitor (see Section 2.2). Allow for configuration of the code that deals with platform specific interfaces by means of e.g. Static Adapter (see Section 2.3).

A first sketch of the solution is shown in Table 3.1.

Table 3.1: Class–Responsibility–Collaboration Cards

Client	
Requests service	Static Framework

(a) Client

Platform	
Grants access to resources to	StaticAdapter

(b) Platform

StaticAdapter	
Adapts implementation of	StaticFramework
to	Platform

(c) Static Adapter

StaticFramework	
Prepares environment	StaticAdapter
Passes service request from	Client
to	StaticAdapter

(d) Static Framework

StaticStrategy	
Finally processes service request on behalf of	StaticFramework

(e) Static Strategy

3.6.1 Participants

Client The Client requests a service from Static Framework. Many clients may request the same service in parallel. Response time is important for each Client.

Platform An interface to a Layer the Static Adapter communicates with. The interfaces of different Platforms might differ significantly. Platforms often provide access to entities that can be acquired and then released again. Such entities are referred to as resources. A Platform remains fixed during runtime of the application and most likely for even much longer periods.

StaticAdapter Mediates between Static Framework and Platform to allow for easier reuse of Static Framework on many Platforms.

StaticFramework A Framework like representation of the idea of the implementation of the server reacting upon service requsts from Clients. Resource usage should be minimized. Static Framework delegates implementation details specific to a certain Platform to Static Adapter and the implementation of a specific service to Static Strategy.

StaticStrategy A user supplied function pointer or function object which plugs into a Hot Spot [RJ98, pp 478–479] of the Static Framework. Conforms to the Static Strategy or the Static Visitor design pattern.

Figure 3.1 sketches the participants and their relations to each other.

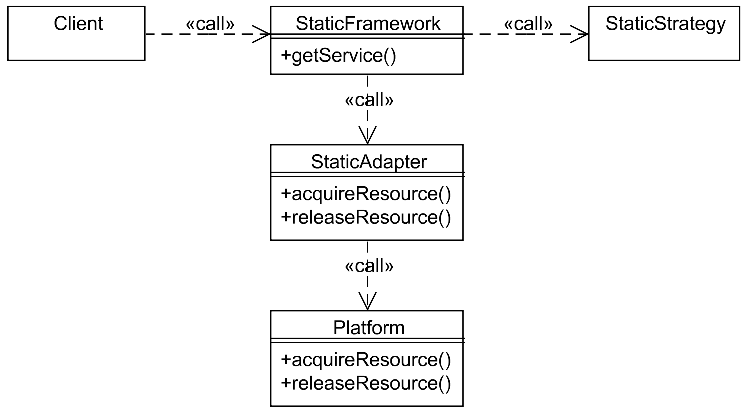

Figure 3.1: Class diagram illustrating Static Framework

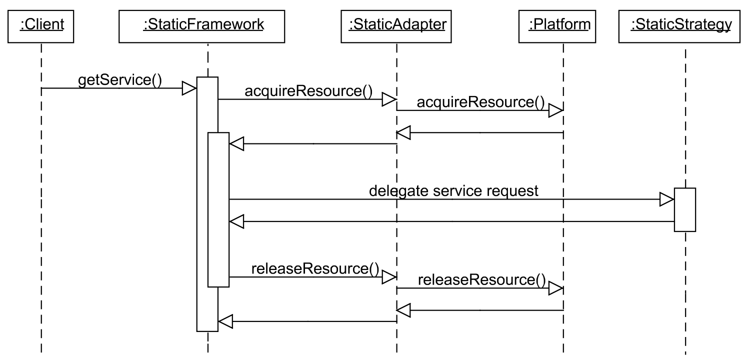

Figure 3.2: Sequence diagram illustrating Static Framework

3.6.2 Dynamics

Instead of interweaving user code with Framework code this pattern advocates the introduction of Static Framework. The Client directs a service request to Static Framework. With help of Static Adapter and indirectly of Platform the latter prepares an environment necessary to fulfill the request. From within this environment it delegates work to the Static Strategy. The dynamics is shown in Figure 3.2.

3.6.3 Rationale

Increasing the level of abstraction and explicitly representing the intent of implementations means to generalize the code. Separating Framework code from user code helps to substitute the Static Adapter by another implementation that better conforms to a new platform, i.e. instead of the code the intent will be the starting point of porting this application. Otherwise adaptation means three steps at once:

1. The original intent must be reconstructed from the implementation which is mixed up of Static Framework, Platform specific code and the Static Strategy, if the intent did not have been clearly documented.

2. An analysis of the target environment results in a new implementation of this intent.

3. The new implementation has to be merged with the Static Strategy.

This potentially has to be repeated for every new situation. Because this is hard work, most of the time a short cut will be taken: The original code will be ported one by one, even if the result is an incorrect application.

Because different service implementations can be injected into the Static Framework in terms of a Static Strategy, it can be reused in a lot of different situations, that share the same orchestration of objects with each other.

3.7 Resulting Context

The level of abstraction represented in the code became increased. The implementation is split into Platform specific code, and code that does not depend on a specific Platform. The Platform dependent code is organized such that it can be replaced easily by another implementation for another Platform. For this to work only the Static Adapter has to be replaced. There are at least as many Static Adapters available as there are supported Platforms. The Platform independent code is splitted up in a Static Strategy and the Static Framework. The latter orchestrates the interplay of the other participants.

3.7.1 Pros and Cons

The Static Framework pattern has the following benefits:

1. *Design reveals essence of problem.* Splitting an application into several components often contributes to a better understanding of the overall business problem. In theory this understanding was the result of the analysis phase. Understanding requirements more often will be an iterative process, and trying to find key components necessary to fulfill these requirements yields better systems.

2. *Keeping the architecture healthy.* Introducing Static Frameworks contributes to an important intent of Agile approaches. Architectures need continuous Refactoring [Fow99] to keep them healthy [PP03c, pp 141–142].

3. *Reusability.* Frameworks designed this way facilitate reusability regarding both different Platforms and different services represented by the user supplied Static Strategies.

4. *Shifting the point of variation upwards.* The point of variation got shifted, thus the system is adaptable to a wider range of Platforms. Large parts of the orchestration of objects performed by Static Framework can be made configurable. This increases adaptability even further and leads to a larger Static Adapter and a thinner Static Framework or advocates the additional use of Template Method Based on Parameterized Inheritance [CE00, pp 231–234].

5. *Preserves performance as if optimized for a single Platform only.* Each Static Adapter can carfully be optimized for its Platforms. Because the configuration is performed statically, the final application is assembled by the compiler, and there will be no overhead induced by virtual calls or missing opportunities for inlining.

The Static Framework pattern has the following liability:

1. *Building Frameworks is hard.* It requires much experience to decide what has to go into Static Framework and what into Static Strategy. Some of the difficulties result from the fact that both Static Framework and Static Strategy reify behavior—there are no real world entities that parallel the object oriented abstractions. As with Frameworks in general finding a good balance might require Three Examples [RJ98, pp 472–474].

3.8 Implementation

As this is a very general design pattern, there can hardly be a detailed suggestion for an implementation that fits all cases. Probably the most difficult step during implementation is to decide how to split the code into Static Framework, Static Strategy, and Static Adapter. As a rule of thumb code that depends on Platform more likely belongs to Static Adapter than to Static Framework and vice versa. Service specific code that is likely to change between different instantiations of Static Framework should go into Static Strategy.

3.8.1 Example Resolved

Listing 3.2 shows the class template `MQ_Scheduler<>`. `MQ_Scheduler<>` carries the intention of the example presented in Section 3.2.

Here the use of the Command design pattern was substituted by a static variant similar to Static Strategy, and the Adapters used in conjunction with Bridge were replaced by Static Adapters, that determine how to deal with specific Units of Execution [MSM04, pp 217–221].

Listing 3.2: A Unit of Execution executing a function object

```
// Header file
extern "C" {
  void *svc_run(void *);
}
```

```

class Impl {
  Activation_List act_queue_;
  volatile bool isActive_;
public:
  struct command_adapter {
    virtual ~command_adapter();
    virtual void call() =0;
  };
  explicit Impl(size_t);
};

template< typename UE > class MQ_Scheduler : public Impl {
  typedef UE unit_of_execution_type;
  template< typename Command > class command_proxy
    : public command_adapter {
    Command &command_;
  public:
    explicit command_proxy(Command &command) : command_(command) {}
    void call() {
      command_.call();
    }
  };
  unit_of_execution_type thread_;
  void createUE();
  void joinUE();
public:
  explicit MQ_Scheduler(size_t high_water_mark)
    : Impl(high_water_mark) {
    createUE();
  }
  ~MQ_Scheduler() {
    isActive_=false;
    joinUE();
  }
  template< typename Command >
  void insert(const Command &method_request) {
    act_queue_.insert(new command_proxy< Command >(method_request));
  }
};

// Specializations of member functions
#ifdef _WIN32
template<> void MQ_Scheduler< HANDLE >::createUE();

template<> void MQ_Scheduler< HANDLE >::joinUE();
#endif /* defined(_WIN32) */

#ifdef UNIX
template<> void MQ_Scheduler< pthread_t >::createUE();

template<> void MQ_Scheduler< pthread_t >::joinUE();
#endif /* defined(UNIX) */

// Implementation file
void *svc_run(void *arg) {
  assert(arg);
```

```
  Impl *impl=static_cast< Impl * >(arg);
  while(impl->isActive_) {
    Impl::command_adapter *mr;
    // Block until the queue is not empty
    impl->act_queue_.remove(&mr);
    try {
      mr->call();
    }
    catch(...) {
    }
    delete mr;
  }
  return 0;
}

Impl::command_adapter::~command_adapter() {
}

Impl::Impl(size_t high_water_mark)
  : act_queue_(high_water_mark), isActive_(true) {}

#ifdef _WIN32
void MQ_Scheduler< HANDLE >::createUE() {
  thread_=reinterpret_cast< HANDLE >(
            _beginthreadex(0,0,svc_run,this,0,0)
          ));
  if(!thread_)
    throw std::runtime_error(
      "Call␣to␣\"_beginthreadex()\"␣failed."
    );
}

void MQ_Scheduler< HANDLE >::joinUE() {
  if(WAIT_FAILED==WaitForSingleObject(thread_,INFINITE)) {
    throw std::runtime_error(
      "Call␣to␣\"WaitForSingleObject()\"␣failed."
    );
}
#endif /* defined(_WIN32) */

#ifdef UNIX
void MQ_Scheduler< pthread_t >::createUE() {
  if(pthread_create(&thread_,NULL,svc_run,this))
    throw std::runtime_error(
      "Call␣to␣\"pthread_create()\"␣failed."
    );
}

void MQ_Scheduler< pthread_t >::joinUE() {
  if(pthread_join(thread_,0))
    throw std::runtime_error(
      "Call␣to␣\"pthread_join()\"␣failed."
    );
}
#endif /* defined(UNIX) */
```

`MQ_Scheduler<>` can be instantiated using either `HANDLE` or `pthread_t`. Not all template instantiations are possible on all platforms. It was also possible to add an explicit specialization e.g. for `pid_t` on UNIX platforms—doing so would offer the possibility to switch between threads and processes to the user.

Of course Commands with statically bound types here look somewhat artificial, because they are converted into Commands with dynamically bound types by means of `Impl::command_adapter`. The latter is a technical implementation detail, however, as the operating system does not deal with user defined types, but with opaque pointers instead.

3.9 Variants

Template Method Based on Parameterized Inheritance can further increase the adaptability of Static Framework. That way `MQ_Scheduler<>` could be extended to support designs like One Child per Client [Ste98, pp 732–736] and One Thread per Client [Ste98, pp 752–753] which do not map well to Active Objects.

3.10 Known Uses

Examples of Static Framework can be found in existing software.

3.10.1 Apache httpd 2.x

For a long time Apache httpd is one of the most popular webservers. It is available for a big variety of different hardware architectures and operating systems. With Apache 2.0 multiprocessing modules (MPMs) were introduced. The code was divided into an aspect concerned with the management of Units of Execution and another aspect responsible for request processing. The first aspect was factored out into an MPM with a general interface thus allowing for exchanging a concrete MPM with another implementation. Each MPM potentially daemonizes the webserver and then starts Units of Execution, distributes and balances work among them, adapts the number of Units to the load, listens to asynchronous requests to terminate the webserver, and then shuts the Units down again. The currently available MPMs are grouped into platformspecific sets. The interface is general enough to allow for both threads and processes as Units of Execution. For the UNIX family of operating systems there exist multiprocessed modules similar to the Apache 1.3 design, but also multithreaded and hybrid ones implementing either the Half–Sync / Half–Async [SSRB02b] or Leader / Followers [SSRB02c], [Ste98, pp 754–756] design pattern. Each Apache webserver runs with exactly one MPM. The configuration is done statically before compilation by means of an appropriate command line option on calling the GNU `configure` script. The request processing code is called from the Units of Execution spawned in the MPM configured.

Though Model–View–Controller [BMR+00b], Presentation–Abstraction–Control [BMR+00d], and Separation of Powers [RZ95, pp 24–26] relate to user interfaces, hence another domain than Static Framework, all these patterns separate software into classes with higher likelyhood to change and into classes that likely remain stable. User interfaces change because of both technology changes and because Perceived Integrity is a competitive advantage on the market [PP03b], whereas the Static Framework allows for adaptation to multiple platforms. By some sense it is a user interface, too.

As Platforms often give access to resources, Static Framework will be implemented using techniques like Resource Acquisition is Initialization in languages like C++ [Str98, pp 388–393], [Str94, pp 495–497], [Bac05, Car96, SSRB02e] or the Dispose pattern in C# [Mic05] and Java [AGH01, pp 228–230], [rel], see further [Hen00, pp 6–7].

Acknowledgements

Without the invaluable feedback of Peter Sommerlad, who was the PLoP shepherd of this work, this paper would not have been the way it is now.

I thank Frank Buschmann and Douglas C. Schmidt for their openness to discuss the Wrapper Facade pattern back in spring 2002. The respective email correspondance motivated me to write down the Static Adapter pattern.

Last but not least my thanks and love go to Cornelia Kneser, my wife, for her constant support throughout the writing of the paper.

This work was supported by the Institute for Medical Informatics and Biostatistics, Basel, Switzerland.

Bibliography

[AAC+] Adobe Systems Inc., David Abrahams, Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, Itay Maman, John Maddock, Thorsten Ottosen, Robert Ramey, and Jeremy Siek: *Boost.typetraits*. <http://www.boost.org/doc/html/-boost_typetraits.html>.

[AG05] Abrahams, David and Aleksey Gurtovoy: *C++ Template Metaprogramming. Concepts, Tools, and Techniques from Boost and Beyond*. C++ In–Depth Series; ed. by Bjarne Stroustrup. Addison Wesley Professional, Boston, Massachusetts..., January 2005.

[AGH01] Arnold, Ken, James Gosling und David Holmes: *Die Programmiersprache Java. Deutsche Übersetzung von RederTranslations,* Dorothea Reder *und* Gabi Zöttl. Programmer's Choice. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 2001. German translation of "The Java Programming Language. Third Edition".

[Ale03] Alexandrescu, Andrei: *Modernes C++ Design. Generische Programmierung und Entwurfsmuster angewendet, Übersetzung aus dem Amerikanischen von* Jobst Giesecke. mitp, Bonn, 1. Auflage, 2003. German translation of "Modern C++ Design: Generic Programming and Design Patterns Applied".

[Ame03] American National Standards Institute, New York, NY: *ISO / IEC 14882:2003(E). Programming languages—C++. Langages de programmation—C++, International Standard*, second edition, October 2003.

[Aue95] Auer, Ken: *Reusability through self–encapsulation.* In Coplien, James O. and Douglas C. Schmidt [CS95], chapter 27, pages 505–516.

[Bac05] Bachmann, Philipp: *Change of authority and thread safe interface goes synchronized*. In *Proceedings of the 12th Pattern Languages of Programs (PLoP) conference 2005, Allerton Park, Monticello, IL, USA*, December 2005. <http://hillside.net/plop/-2005/proceedings/PLoP2005_pbachmann1_0.pdf>.

[BK03] Blochinger, Wolfgang and Wolfgang Küchlin: *Cross–platform development of high performance applications using generic programming*. In *Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems (PDCS) 2003, Reno, NV, USA*, August 2003. <http://www-sr.informatik.uni-tuebingen.de/~bloching/-papers/pdcs03.pdf>.

[BMR+00a] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Layers*, Kapitel 2: Architekturmuster, Seiten 32–53. In: [BMR+00c], 1998, 1. korr. Nachdruck Auflage, 2000. German translation of "Layers".

[BMR+00b] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Model–View–Controller*, Kapitel 2: Architekturmuster, Seiten 124–144. In: [BMR+00c], 1998, 1. korr. Nachdruck Auflage, 2000. German translation of "Model–View–Controller".

[BMR+00c] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern–orientierte Softwarearchitektur. Ein Pattern–System, deutsche Übersetzung von* Christiane Löckenhoff. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 1998, 1. korr. Nachdruck Auflage, 2000. German translation of "Pattern–Oriented Software Architecture. A System of Patterns".

[BMR+00d] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Presentation–Abstraction–Control*, Kapitel 2: Architekturmuster, Seiten 145–169. In: [BMR+00c], 1998, 1. korr. Nachdruck Auflage, 2000. German translation of "Presentation–Abstraction–Control".

[Car96] Cargill, Tom: *Localized ownership: Managing dynamic objects in C++*. In Vlissides, John M. et al. [VCK96], chapter 1, pages 5–18.

[CE00] Czarnecki, Krzysztof and Ulrich W. Eisenecker: *Generative Programming. Methods, Tools, and Applications*. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal ·

London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, first printing, may 2000 edition, 2000.

[CS95] Coplien, James O. and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*, volume 1 of *The Software Patterns Series; ed. by* John Vlissides. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 1995.

[DRS06] Dos Reis, Gabriel and Bjarne Stroustrup: *Specifying C++ concepts.* In *Proceedings of the 33rd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) January 11–13, 2006, Charleston, SC, USA*, January 2006. `<http://www.research.att.com/~bs/popl06.pdf>`.

[FM] Friedman, Eric and Itay Maman: *Boost.variant.* `<http://www.boost.org/doc/html/variant.html>`.

[Fow99] Fowler, Martin: *Refactoring. Improving the Design of Existing Code, with contributions by* Kent Beck*,* John Brant*,* William Opdyke*, and* Don Roberts. The Addison–Wesley Object Technology Series; Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, 1999.

[GHJV96a] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Abstrakte Fabrik*, Kapitel 3: Erzeugungsmuster, Seiten 107–118. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Abstract Factory".

[GHJV96b] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Adapter*, Kapitel 4: Strukturmuster, Seiten 171–185. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Adapter".

[GHJV96c] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Befehl*, Kapitel 5: Verhaltensmuster, Seiten 273–286. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Command".

[GHJV96d] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Besucher*, Kapitel 5: Verhaltensmuster, Seiten 301–318. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Visitor".

[GHJV96e] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Brücke*, Kapitel 4: Strukturmuster, Seiten 186–198. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Bridge".

[GHJV96f] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, deutsche Übersetzung von* Dirk Riehle. Professionelle Softwareentwicklung. Addison–Wesley–Longman, Bonn · Reading, Massachusetts · Menlo Park, California · New York · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Design Patterns. Elements of Reusable Object–Oriented Software".

[GHJV96g] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Fabrikmethode*, Kapitel 3: Erzeugungsmuster, Seiten 131–143. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Factory Method".

[GHJV96h] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Iterator*, Kapitel 5: Verhaltensmuster, Seiten 335–353. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Iterator".

[GHJV96i] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Kompositum*, Kapitel 4: Strukturmuster, Seiten 239–253. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Composite".

[GHJV96j] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Schablonenmethode*, Kapitel 5: Verhaltensmuster, Seiten 366–372. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Template Method".

[GHJV96k] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Strategie*, Kapitel 5: Verhaltensmuster, Seiten 373–384. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Strategy".

[GS06] GREGOR, DOUGLAS and BJARNE STROUSTRUP: *Concepts*. Technical Report WG21/N2042 = J16/06-0112, ISO IEC JTC1 / SC22 / WG21—The C++ Standards Committee, Herb Sutter · Microsoft Corp. · 1 Microsoft Way · Redmond WA USA 98052-6399, June 2006. <http://www.open-std.org/jtc1/-sc22/wg21/docs/papers/2006/n2042.pdf>.

[Hen00] HENNEY, KEVLIN: *Executing around sequences*. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP) 2000, Irsee, Germany*, July 2000. <http://hillside.net/europlop/HillsideEurope/-Papers/ExecutingAroundSequences.pdf>.

[HFR00] HARRISON, NEIL, BRIAN FOOTE, and HANS ROHNERT (editors): *Pattern Languages of Program Design*, volume 4 of *The Software Patterns Series; ed. by* JOHN M. VLISSIDES. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, 2000.

[Kem] KEMPF, WILLIAM E.: *Boost.threads*. <http://www.boost.org/-doc/html/threads.html>.

[KLM+97] KICZALES, GREGOR, JOHN LAMPING, ANURAG MENDHEKAR, CHRIS MAEDA, CRISTINA VIDEIRA LOPES, JEAN-MARC LOINGTIER, and JOHN IRWIN: *Aspect–oriented programming*. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP), Finland*, number 1241 in *Lecture Notes in Computer Science*. Springer, June 1997.

[Mar98] MARTIN, ROBERT C.: *Acyclic visitor*. In MARTIN, ROBERT C. et al. [MRB98], chapter 7, pages 93–103.

[Mey05a] MEYERS, SCOTT: *Effektive C++. 55 Specific Ways to Improve Your Programs and Designs*. Addison–Wesley Professional Computing Series; ed. by BRIAN W. KERNIGHAN. Addison–Wesley, Upper Saddle River, NJ · Boston · Indianapolis · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, third edition, 2005.

[Mey05b] MEYERS, SCOTT: *Item 45: Use member function templates to accept "all compatible types"*, chapter 7: Templates and Generic Programming, pages 218–222. In *Addison–Wesley Professional Computing Series; ed. by* BRIAN W. KERNIGHAN [Mey05a], third edition, 2005.

[Mic05] MICROSOFT DEVELOPER NETWORK (MSDN): *.NET framework general reference: Common design patterns. implementing finalize and dispose to clean up unmanaged resources.* `<http://msdn.microsoft.com/library/en-us/cpgenref/-html/cpconfinalizedispose.asp>`, 2005.

[MRB98] MARTIN, ROBERT C., DIRK RIEHLE, and FRANK BUSCHMANN (editors): *Pattern Languages of Program Design*, volume 3 of *The Software Patterns Series; ed. by* JOHN VLISSIDES. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, 1998.

[MSM04] MATTSON, TIMOTHY G., BEVERLY A. SANDERS, and BERNA L. MASSINGILL: *Patterns for Parallel Programming.* The Software Patterns Series; ed. by JOHN VLISSIDES. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, September 2004.

[PP03a] POPPENDIECK, MARY and TOM POPPENDIECK: *Lean Software Development. An Agile Toolkit.* The Agile Software Development Series; ed. by ALISTAIR COCKBURN, JIM HIGHSMITH. Addison Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 2003.

[PP03b] POPPENDIECK, MARY and TOM POPPENDIECK: *Tool 17: Perceived Integrity*, chapter 6: Build Integrity In, pages 129–135. In *The Agile Software Development Series; ed. by* ALISTAIR COCKBURN, JIM HIGHSMITH [PP03a], 2003.

[PP03c] POPPENDIECK, MARY and TOM POPPENDIECK: *Tool 19: Refactoring*, chapter 6: Build Integrity In, pages 140–145. In *The Agile Software Development Series; ed. by* ALISTAIR COCKBURN, JIM HIGHSMITH [PP03a], 2003.

[PP03d] POPPENDIECK, MARY and TOM POPPENDIECK: *Tool 7: Options Thinking*, chapter 3: Decide as Late as Possible, pages 52–57. In *The Agile Software Development Series; ed. by* ALISTAIR COCKBURN, JIM HIGHSMITH [PP03a], 2003.

[Ray03] RAYMOND, ERIC S[TEVEN]: *The Art of UNIX Programming.* Addison–Wesley Professional Computing Series; ed. by BRIAN W. KERNIGHAN. Addison Wesley Professional, Boston, Massachusetts..., September 2003. `<http://www.catb.org/~esr/-writings/taoup/>`.

[rel] *Releasing resources in Java.* `<http://www.c2.com/cgi/wiki?-ReleasingResourcesInJava>`.

[RJ98] ROBERTS, DON and RALPH JOHNSON: *Patterns for evolving frameworks.* In MARTIN, ROBERT C. et al. [MRB98], chapter 26, pages 471–486.

[RZ95] RIEHLE, DIRK and HEINZ ZÜLLIGHOFEN: *A pattern language for tool construction and integration based on the tools and materials metaphor.* In COPLIEN, JAMES O. and DOUGLAS C. SCHMIDT [CS95], chapter 2, pages 9–42.

[Sie99] SIEK, JEREMY G.: *A modern framework for portable high performance numerical linear algebra.* Master's thesis, Graduate School of the University of Notre Dame, Department of Computer Science and Engineering, Notre Dame, Indiana, April 1999. `<http://osl.iu.edu/download/research/mtl/papers/-thesis.pdf>`.

[SLL] SIEK, JEREMY, LIE-QUAN LEE, and ANDREW LUMSDAINE: *The boost graph library.* `<http://www.boost.org/libs/graph/doc/-table_of_contents.html>`.

[SLL02] SIEK, JEREMY G., LIE-QUAN LEE, and ANDREW LUMSDAINE: *The Boost Graph Library. User Guide and Reference Manual. Foreword by* ALEXANDER STEPANOV. C++ In–Depth Series; ed. by BJARNE STROUSTRUP. Addison Wesley Professional, Boston, Massachusetts…, 2002.

[SSRB02a] SCHMIDT, DOUGLAS, MICHAEL STAL, HANS ROHNERT und FRANK BUSCHMANN: *Active Object*, Kapitel 5: Nebenläufigkeit, Seiten 411–443. In: [SSRB02d], 2002. German translation of "Active Object".

[SSRB02b] SCHMIDT, DOUGLAS, MICHAEL STAL, HANS ROHNERT und FRANK BUSCHMANN: *Half–Sync / Half–Async*, Kapitel 5: Nebenläufigkeit, Seiten 473–497. In: [SSRB02d], 2002. German translation of "Half–Sync / Half–Async".

[SSRB02c] SCHMIDT, DOUGLAS, MICHAEL STAL, HANS ROHNERT und FRANK BUSCHMANN: *Leader / Followers*, Kapitel 5: Nebenläufigkeit, Seiten 499–528. In: [SSRB02d], 2002. German translation of "Leader / Followers".

[SSRB02d] SCHMIDT, DOUGLAS, MICHAEL STAL, HANS ROHNERT und FRANK BUSCHMANN: *Pattern–orientierte Software-Architektur. Muster für nebenläufige und vernetzte Objekte, übersetzt aus dem Amerikanischen von* MARTINA BUSCHMANN. dpunkt.verlag, Heidelberg, 2002. German translation of "Pattern–Oriented Software

Architecture. Volume 2: Patterns for Concurrent and Networked Objects".

[SSRB02e] SCHMIDT, DOUGLAS, MICHAEL STAL, HANS ROHNERT und FRANK BUSCHMANN: *Scoped Locking*, Kapitel 4: Synchronisation, Seiten 359–367. In: [SSRB02d], 2002. German translation of "Scoped Locking".

[SSRB02f] SCHMIDT, DOUGLAS, MICHAEL STAL, HANS ROHNERT und FRANK BUSCHMANN: *Wrapper–Facade*, Kapitel 2: Dienstzugriff und Konfiguration, Seiten 53–84. In: [SSRB02d], 2002. German translation of "Wrapper Facade".

[Ste98] STEVENS, W. RICHARD: *UNIX Network Programming*, volume 1. Networking APIs: Sockets and XTI. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1998.

[Str94] STROUSTRUP, BJARNE: *Design und Entwicklung von C++*. Addison–Wesley, Bonn · Paris · Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario · Workingham, England · Amsterdam · Milan · Sydney · Tokyo Singapore · Madrid · San Juan · Seoul · Mexico City · Taipei, Taiwan, 1994. German translation of "The Design and Evolution of C++".

[Str98] STROUSTRUP, BJARNE: *Die C++–Programmiersprache. Deutsche Übersetzung von* NICOLAI JOSUTTIS *und* ACHIM LÖRKE. Addison–Wesley–Longman, Bonn · Reading, Massachusetts · Menlo Park, California · New York · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, dritte, aktualisierte und erweiterte Auflage, 1998. German translation of "The C++ Programming Language, Third Edition".

[Suta] SUTTER, HERB: *G[uru] o[f] t[he] w[eek] #16. maximally reusable generic containers.* `<http://www.gotw.ca/gotw/016.htm>`.

[Sutb] SUTTER, HERB: *G[uru] o[f] t[he] w[eek] #62. smart pointer members.* `<http://www.gotw.ca/gotw/062.htm>`.

[VCK96] VLISSIDES, JOHN M., JAMES O. COPLIEN, and NORMAN L. KERTH (editors): *Pattern Languages of Program Design*, volume 2. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, 1996.

[VJ03a] VANDEVOORDE, DAVID and NICOLAI M. JOSUTTIS: *C++ Templates. The Complete Guide.* Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, June 2003.

[VJ03b] Vandevoorde, David and Nicolai M. Josuttis: *Function Objects and Callbacks*, chapter 22, pages 417–474. In [VJ03a], June 2003.

[VJ03c] Vandevoorde, David and Nicolai M. Josuttis: *Type Classification*, chapter 19, pages 347–364. In [VJ03a], June 2003.

[Wie60] Wiener, Norbert: *Some moral and technical consequences of automation. as machines learn they may develop unforeseen strategies at rates that baffle their programmers.* Science. American Association for the Advancement of Science, 131(3410):1355–1358, May 1960.

[Wit60] Wittgenstein, Ludwig: *Tractatus logico–philosophicus · Tagebücher · Philosophische Untersuchungen.* Suhrkamp Verlag, Frankfurt am Main, 1960. Translation: Tractatus logico–philosophicus. German text with an English translation *en regard* by C[harles] K[ay] Ogden, with an introduction by Bertrand Russell, London, UK: Routledge and Kegan Paul 1922, see `<http://www.kfs.org/~jonathan/witt/tlph.html>`.

[Woo00] Woolf, Bobby: *Abstract class.* In Harrison, Neil et al. [HFR00], chapter 1, pages 5–14.

[Zan02] Zaninotto, Enrico: *Keynote "from X programming to the X organisation. 3rd international conference on extreme programming, may, 26–29 alghero".* `<http://ciclamino.dibe.unige.it/-xp2002/talksinfo/zaninotto.pdf>`, 2002.