Static and Metaprogramming Patterns and Static Frameworks

A Catalog. An Application

Philipp Bachmann
Institute for Medical Informatics and Biostatistics
Clarastrasse 12
CH–4058 Basel, BS
Switzerland
bachlipp@web.de

ABSTRACT

The classic UNIX principle to write code that generates code instead of writing this code yourself [48, Chapters 1,9] is experiencing a revival. Much research was done, the techniques are better understood now, and the generation tools were refined.

This pattern catalog consists of adaptations of the Gang of Four design patterns [27] Abstract Factory, Adapter, Strategy, and Visitor to the metaprogramming level. It shows that replacing runtime polymorphism by static polymorphism helps to lift variation from the code level up to the meta level, where it might more naturally belong to. Some of the patterns proposed are especially useful for facilitating portable code.

The patterns shown can be used to build static Frameworks [50]. A simple example is also presented.

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

Each pattern presentation is accompanied with an example. These examples show sample code in C++. The template metaprogramming capabilities of C++ [2, 17, 65] allow us to express both the program and the meta program in the same programming language.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain–specific architectures, Patterns*; D.3.2 [**Programming Languages**]: Language Classifications—*C++, Multiparadigm languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract Data Types, Frameworks, Polymorphism*; D.4.0 [**Operating Systems**]: General—*Microsoft Windows NT, UNIX*

General Terms

Design

Keywords

Design pattern, generative programming, generic programming, metaprogramming, portability, software product lines, static polymorphism, template, template specialization

A preliminary version of this paper was workshopped at the 13th Pattern Languages of Programs (PLoP) conference 2006.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. Copyright 2006, 2007 is held by the author and the Institute for Medical Informatics and Biostatistics.
PLoP '06, October 21–23, 2006, Portland, OR, USA.
ACM 978-1-60558-151-4/06/10.

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 construction plan in the generation software, and to allow for future adaptations by changing the construction plan. Generative programming [17] 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 generated software into the DSL. This often leads to better optimization opportunities.

The author identified patterns on the metaprogramming level. His goal was to centralize the configuration and to reduce the need for conditional compilation.

Table 1 lists the patterns proposed.

With generative programming at hand the recurring problem of software portability can be solved in an appropriate and more elegant way than without. The patterns Static Adapter and Static Abstract Type Factory especially aim at portability. Usually portability means cross–platform portability. Portability also has a temporal aspect. The availability of a product for many platforms and for long periods of time can provide a significant competitive advantage. So this pattern catalog also assists you in both understanding variability necessary for portable applications and filling the gaps where supportive libraries are not available or cannot be used.

The presentation of each pattern follows the style well known from [13] and [54]. Additionally, each pattern description contains a twofold statement:

Table 1: Pattern–Thumbnails of Static and Metaprogramming Patterns

Section	Title	Intent
3	Static Strategy	Delegate certain tasks an object needs to perform to another object. Allow the delegation to be chosen statically.
4	Static Visitor	Encapsulate operations that should be applied to all elements of an object structure in a separate object. Adding new operations becomes easier then. Different from the Visitor pattern the Static Visitor pattern does not depend on runtime polymorphism at all. It shifts the dependency cycle present in the original Visitor design pattern from the program to the meta program.
5	Static Adapter	Adapt a series of different interfaces to a common interface. The choice of which interface is actually to be adapted can be done at compile time.
6	Static Abstract Type Factory	The Static Abstract Type Factory provides an extensible means to associate expressions in the domain specific language with application data types.
7	Static Framework	Portable code must meet performance requirements on each platform. 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.

1. It presents a pattern on the meta level and
2. it shows how to move variation from the program to the meta program.

The overall pattern descriptions are such that they point to the first statement, whereas contrasting the motivating example with the respective implementation section points you to the second statement.

The code examples are in the C++ programming language, because its generative programming capabilities allow us to use it at both the base and meta levels.

2. METAPROGRAMMING PATTERNS

> What can be said at all can be said clearly.
>
> Ludwig Josef Johann Wittgenstein: Preface of Tractatus logico–philosophicus [71]

The following four Sections 3 to 6 propose the use of generative and template metaprogramming techniques [2, 17, 65] to express classic patterns by the Gang of Four [27] on the metaprogramming level.

After reading these sections the reader will know how to refactor code to move points of variation from the base 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.

The Static Strategy (see Section 3) idea has been published as an implementation option of the Strategy design pattern before in [32]. We repeat it here in more detail because it is a good introduction to static and metaprogramming patterns. Metaprogramming variants of other Gang of Four patterns can be found e.g. in [17, pp 224–234].

3. STATIC STRATEGY

An instance or class based bahavioral pattern

3.1 Also known as

Policy [5, pp 27–51], [66, pp 429–436]
Strategy [32, pp 378–379]

3.2 Intent

Delegate certain tasks an object needs to perform to another object. Allow the delegation to be chosen statically.

3.3 Example

Suppose that you need to implement a stack. 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 [32] design pattern as sketched in Listing 1.

Listing 1: Dynamically injecting an allocator into a stack

```
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:
  explicit IntStack(AllocatorIf &a}
    : allocator_(&a), ... {}
  ...
  void pop() throw() {
    // Calls "allocator_->deallocate()"
    ...
  }
  void push(int data) {
    // Calls "allocator_->allocate()"
    ...
```

```
  }
  int top() const {
    // Returns topmost element, but don't
    // remove it from the stack
    ...
  }
};
```

The details of `IntStack` do not matter here. They are similar to the implementation shown below in Listing 3. An implementation of the `AllocatorIf` interface is injected into `IntStack` on construction. Note that different instances using different allocators do not differ in type.

Listing 2 shows how the parts proposed above work together to instantiate and use a stack.

Listing 2: How to use Strategy

```
int main() {
  NewAllocator alloc;
  IntStack s(alloc);
  for(int i(0);i<42;++i)
    s.push(i);
  for(int i(0);i<42;++i) {
    std::cout<<s.top()<<" ";
    s.pop();
  }
  std::cout<<std::endl;
  return EXIT_SUCCESS;
}
```

Additionally to this simple usage example the (dynamic) Strategy pattern allows for strategies to be generated by a Factory Method [28] or other creational pattern, which returns a pointer to `AllocatorIf` thus hiding its dynamic type.

The Template Method design pattern [31] was another implementation option to factor out implementation details. A disadvantage compared to Strategy was that implementation details factored out into subclasses cannot be reused as Strategies can.

In many cases you know the allocator for a certain stack at compile time. Using dynamic polymorphism may be considered too much of a good thing therefore. So a way is sought to statically bind allocators to the stack instance while still separating allocator from stack code.

3.4 Context

The implementation of certain classes representing e.g. abstract data types consists of different concerns that crosscut each other [37]. These concerns are often bound to the class, not to its instances, and must then be kept immutable for consistency reasons.

3.5 Problem

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

3.6 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.

3.7 Solution

Separate an abstract data type into its essence and an exchangeable class or instance of another class to which it delegates implementation details. 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.

3.7.1 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.

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 1 sketches the participants and their relations to each other.

3.7.2 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.

3.7.3 Rationale

Some configuration issues can be decided early at compile time. In fact, some Abstract Data Types only work correct, 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.

3.8 Resulting Context

Implementation details were factored out of the Abstract Data Type. The AbstractDataType 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.

3.8.1 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.

Table 2: 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	StaticStrategy-Concept
Encapsulates algorithm implementation	

(c) Concrete Strategy

StaticStrategyConcept	
Declares interface to algorithm	

(d) Static Strategy Concept

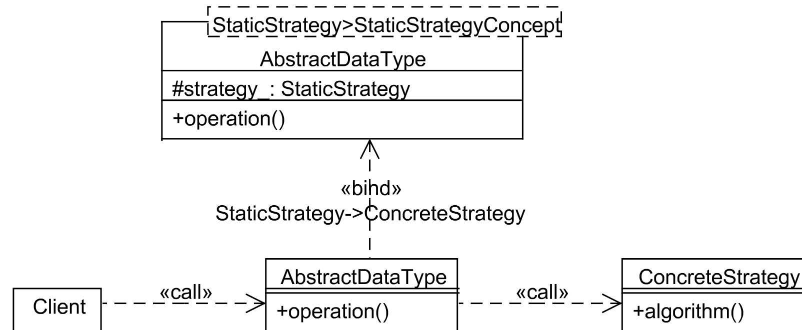

Figure 1: Class diagram illustrating Static Strategy

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 assignment operator member function and copy constructor templates to enable this [63, 64, 42].

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

The implementation technique of the Static Strategy 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 in a future version of the C++ standard, e.g. [18, 33].

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

3.9 Implementation

Once we decided on 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.

3.9.1 Example Resolved

The code shown in Listing 3 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 3: Statically injecting an allocator into a stack

```
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 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.

Listing 4 shows how the parts proposed above work together to instantiate and use a stack.

Listing 4: How to use Static Strategy

```
int main() {
  // Uses "NewAllocator", its default
  // allocator
  IntStack s;
  for(int i(0);i<42;++i)
    s.push(i);
  for(int i(0);i<42;++i) {
    std::cout<<s.top()<<" ";
    s.pop();
  }
  std::cout<<std::endl;
  return EXIT_SUCCESS;
}
```

Different from the (dynamic) Strategy pattern the strategy can't be dynamically created using a creational design pattern—the type of the strategy has to be known at compile time. A static parallel to the creational patterns is proposed in Section 6. Using this technique the strategy class could be hidden behind a `typedef` after all.

3.9.2 Relationship of Example and Participants

The code shown as an example above maps to the participants defined in Section 3.7.1 as shown in Figure 3.

3.10 Variants

Depending on the purpose of the Strategy there are two different implementation options with respect to the granularity of configuration possible. Either the 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 4).

3.11 Known Uses

Examples of Static Strategy can be found in existing software.

3.11.1 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.

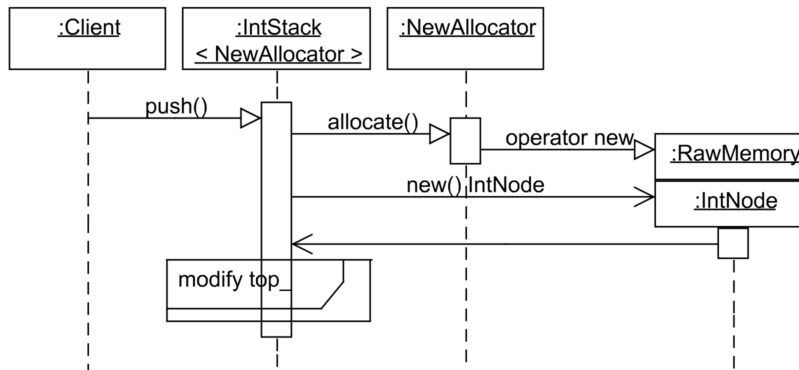

Figure 2: Sequence diagram illustrating Static Strategy

Table 3: Relationship of Code and Participants

Code	Participant
`template<> class IntStack`	AbstractDataType
`int main()`	Client
`struct NewAllocator`	ConcreteStrategy
Implicit. See liability 2 in Section 3.8.1.	StaticStrategyConcept

3.11.2 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.

3.11.3 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 [29] 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<>`.

3.12 Related Patterns

The Static Visitor pattern (see Section 4) 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.

4. STATIC VISITOR

In languages like C++ there is no built–in dynamic double dispatch, i.e. on calling a virtual member function the actual member function called is chosen solely based on the dynamic type of the respective instance, but never additionally dependent on the dynamic type of one of its arguments. The Visitor design pattern [25] uses runtime polymorphism and inversion of control to provide double dispatch. The class diagram corresponding to the original Visitor pattern is shown in Figure 3.

4.1 Intent

Encapsulate operations that should be applied to all elements of an object structure in a separate object. Adding new operations becomes easier then. Different from the Visitor pattern the Static Visitor pattern does not depend on runtime polymorphism at all. It shifts the dependency cycle present in the original Visitor design pattern from the program to the meta program.

4.2 Example

Consider you have a set of classes representing the different entities a file system consists of. One of these classes represents a directory. Directories are Composites [30] that contain instances of classes within the given set including directories. A user might want to traverse the directories recursively and apply a function on the elements encountered. A simple case was to count the elements residing in a certain directory regardless of whether these elements are files or directories. This can be implemented as follows.

Listing 5: Life without visitors

```
struct FileSystemElementIf {
  virtual ~FileSystemElementIf() {}
```

```
  virtual std::size_t count() const =0;
};

class File : public FileSystemElementIf {
  ...
public:
  std::size_t count() const {
    return 1;
  }
  ...
};

class Directory
  : public FileSystemElementIf {
  ...
  typedef
    std::list< FileSystemElementIf * >
    list_type;
public:
  std::size_t count() const {
    list_type ls;
    ...
    // Count oneself
    std::size_t result(1);
    for(list_type::const_iterator
        ci(ls.begin());
      ls.end()!=ci;
      ++ci
    )
      result+=(*ci)->count();
    return result;
  }
  ...
};
```

The details of `File` and `Directory` do not matter here. They are similar to the implementation shown below in Listing 8.

There are two dimensions to extend this class hierarchy. You could add more virtual member functions like `FileSystemElementIf::count()`. And you could add more realizations of `FileSystemElementIf`, i.e. siblings of `File` and `Directory`. The first extension requires you to modify `FileSystemElementIf`, which is impossible if the class hierarchy resides within a library and you don't have access to its source code.

Consider you decided that it's more likely that you will add further virtual member functions like `FileSystemElementIf::count()` than that you will add new classes to the hierarchy. The Visitor design pattern helps then. It works similar to `find -exec` from a UNIX shell.

A traditional implementation looks as shown in Listing 6.

Listing 6: Classic Visitor with double dispatch

```
class File;
class Directory;

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

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

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

class Directory
  : public FileSystemElementIf {
  ...
  typedef
    std::list< FileSystemElementIf * >
    list_type;
public:
  void accept(VisitorIf &v) {
    v.visitDirectory(*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 8.

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

Listing 7: Example for visitors

```
class Count : public VisitorIf {
  std::size_t number_of_elements_;
public:
  Count() : number_of_elements_(0) {}
  void visitFile(File &) {
    ++number_of_elements_;
  }
  void visitDirectory(Directory &) {
    ++number_of_elements_;
  }
  std::size_t getNumber_of_elements()
    const {
    return number_of_elements_;
  }
};
```

The usage of such visitors doesn't differ from Listing 10. However, different from the listing just referred to, the classic Visitor pattern allows for visitors to be generated by a Factory Method [28] or other creational pattern, which returns a pointer to `VisitorIf` thus hiding its dynamic type.

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 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 its realizations if there is no default realization. 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 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.

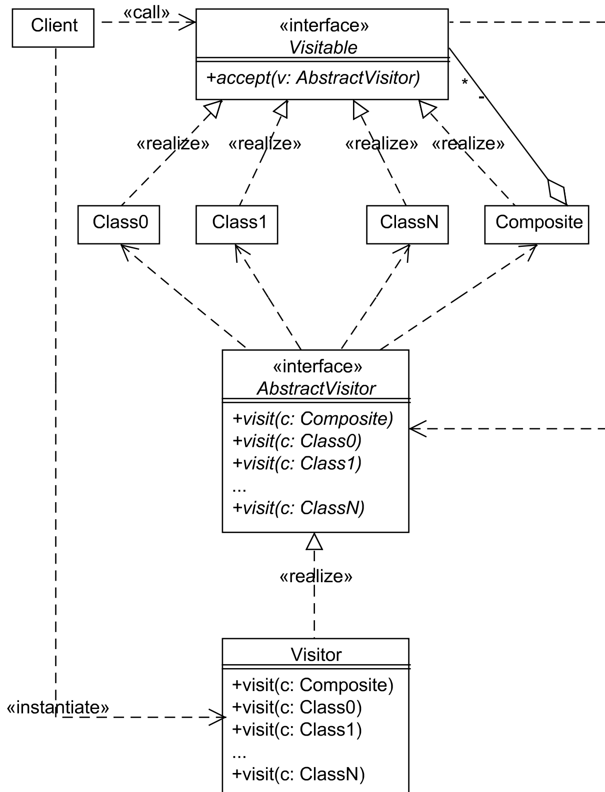

Figure 3: Class diagram illustrating the classic Visitor with double dispatch [25]

It is also worth noting that this implementation uses double dispatch. The meaning of a call to one of the virtual `accept()` member functions both depends on the dynamic type of the instance `accept()` is called on and on the dynamic type of the visitor, the argument to `accept()`.

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

4.3 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.

4.4 Problem

Different algorithms will be applied to the instances arranged in the hierarchy possibly using different traversal strategies. The algorithms are not known at the time the class hierarchy was fixed. So it is not an option to add all algorithms to the set of classes given. But nevertheless the algorithms are known at compile time. 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?

4.5 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.
- Programming towards typesafety means to detect errors early at compile time instead of runtime.
- If the execution of the member functions of a visitor is inexpensive, then the overhead caused by virtual calls—indirection and missing inlining opportunities—becomes a large fraction of the overall traversal time of this visitor.

4.6 Solution

Equip the classes the set consists of once and for all with a member function template accepting an instance of any class that is a model of some visitor concept. This prevents you from the need to repeatedly add functionality to each of these classes. 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 different member functions for each class or by means of overloading.

A first sketch of the solution is shown in Table 4.

4.6.1 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 4 sketches the participants and their relations to each other.

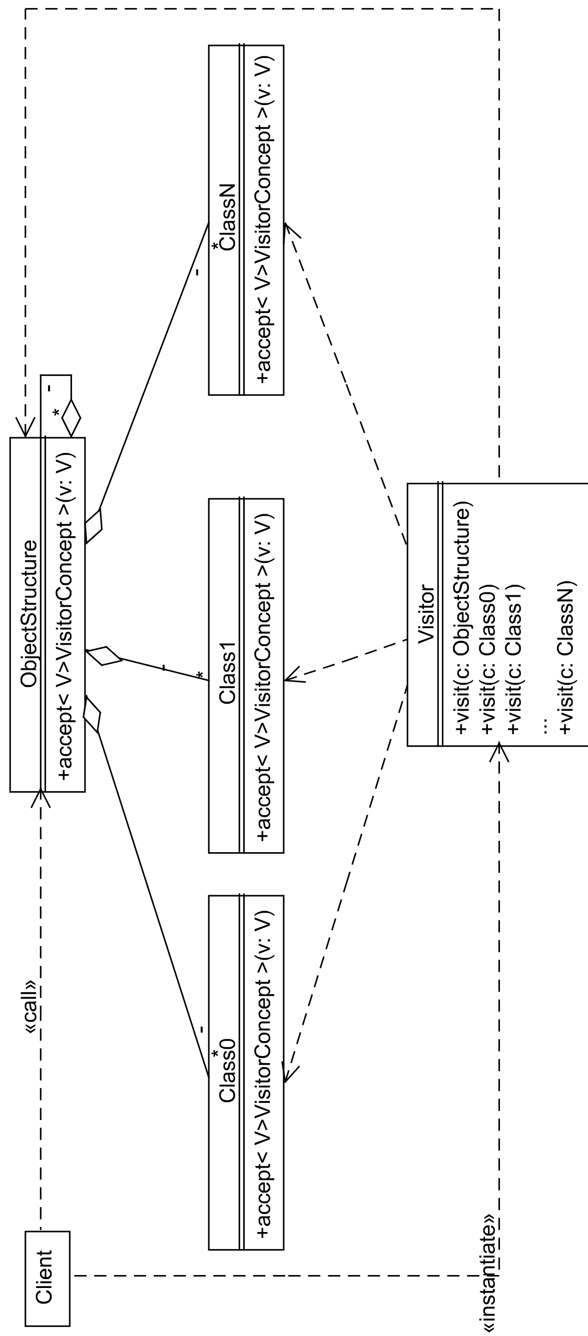

Figure 4: Class diagram illustrating Static Visitor

Table 4: 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

4.6.2 *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 5.

4.6.3 *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. This dependency cycle isn't eliminated with Static Visitor, but it is shifted to the meta program. Because of liability 2 in Section 4.7.1 it cannot be expressed in the code.

The original publication of Visitor uses runtime polymorphism to get double dispatch even in programming languages that don't provide this as a language feature. Here we use static polymorphism and lift the double dispatch to the meta level.

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 as long as the recursion allows.

4.7 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.

4.7.1 *Pros and Cons*

The Static Visitor pattern has the following benefits:

1. *Algorithms can be added.* It's easy to add further algorithms.

2. *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.

3. *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.

The Static Visitor pattern has the following liability:

1. *Extending Object Structure is hard.* The Visitor and Static Visitor patterns trade extensibility regarding new classes in for extensibility regarding further algorithms.

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

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

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 in a future version of the C++ standard, e.g. [18, 33].

4.8 Implementation

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

4.8.1 *Example Resolved*

Listing 8 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

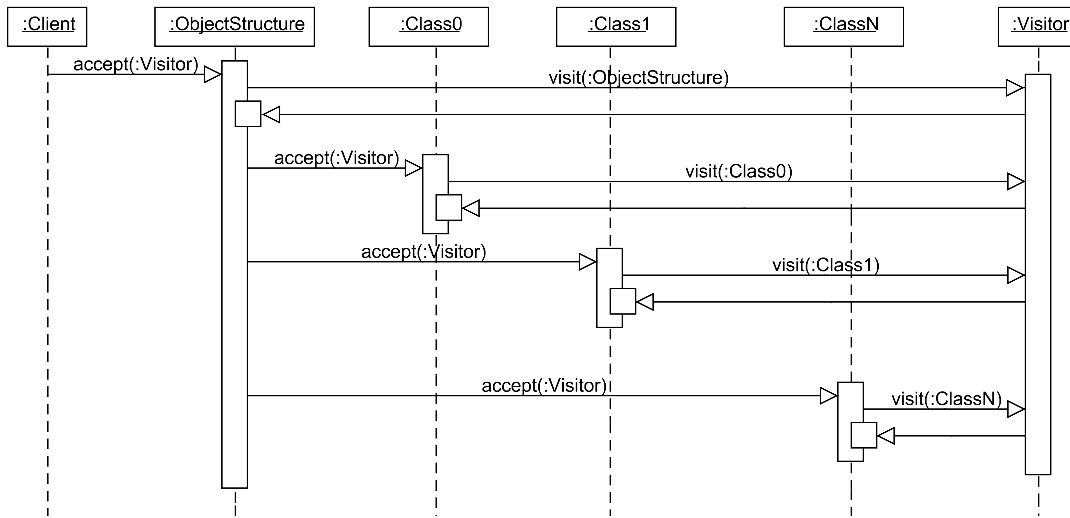

Figure 5: Sequence diagram illustrating Visitor

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. Here we substituted the different `visitFile()` and `visitDirectory()` member functions present in Listing 6 by a series of overloaded `visit()` member functions just to support visitor implementations as simple as `Count` as shown in Listing 9.

Listing 8: 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 9 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 types. 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 9: 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_;
  }
};
```

Listing 10 shows how the parts proposed above work together to count the number of file system elements.

Listing 10: How to use visitors

```
int main() {
  Directory dir("/home/bachlipp");
  Count ctr;
  dir.accept(ctr);
  std::cout<<"\"/home/bachlipp\"␣contains"
    <<"␣"
    <<ctr.getNumber_of_elements()-1
    <<"␣elements."
    <<std::endl;
  return EXIT_SUCCESS;
}
```

Different from the classic Visitor pattern the visitor can't be dynamically created using a creational design pattern—the type of the visitor has to be known at compile time. A static parallel to the creational patterns is proposed in Section 6. Using this technique the visitor class could be hidden behind a `typedef` after all.

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`.

4.8.2 *Relationship of Example and Participants*

The code shown as an example above maps to the participants defined in Section 4.6.1 as shown in Figure 5.

4.9 Variants

A particularly attractive variant is the combination with a variation [68, pp 87–90] of Acyclic Visitor [38], [5, pp 322–328]. It moves the dependency of the declaration 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 6 and Listings 11 and 12 sketch this variant.

Listing 11: Static Visitor enabling the use of Acyclic Visitors

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

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

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

The details of `File` and `Directory` do not matter here. They are similar to the implementation shown above in Listing 8. 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 [7, 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 12: 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();
```

Table 5: Relationship of Code and Participants

Code	Participant
`class Directory`, `class File`	ClassN
`int main()`	Client
`class Directory`	ObjectStructure
`class Count`	Visitor
Implicit. See liability 2 in Section 4.7.1.	VisitorConcept

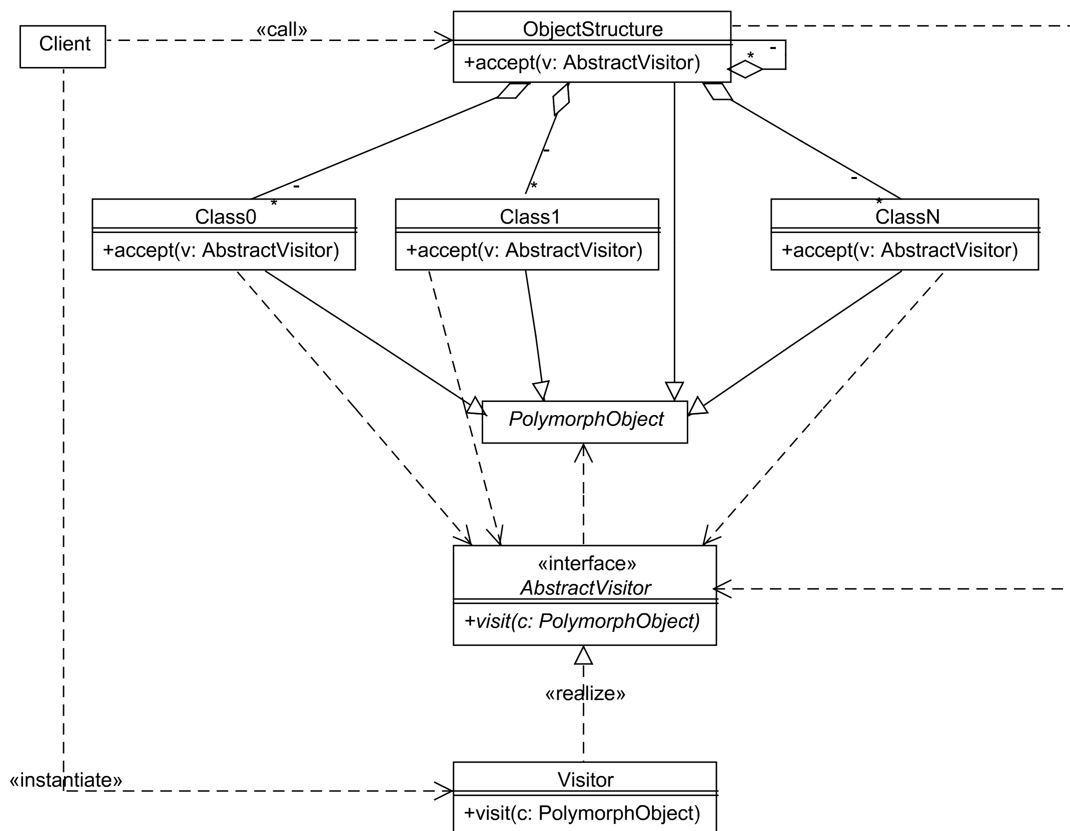

Figure 6: Class diagram illustrating Yet Another Acyclic Visitor

```
    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 static selection 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.

4.10 Known Uses

Examples of Static Visitor can be found in existing software.

4.10.1 Boost.Variant

Boost.Variant [20] 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 `boost::apply_visitor<>()` member function template that fulfills the purpose `accept<>()` fulfills above.

4.10.2 Boost Graph library

The Boost Graph library [57, 59] 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 templates in the description above.

4.11 Related Patterns

The Static Strategy pattern (see Section 3) also inverts control flow.

The Static Visitor is an Internal resp. Passive Iterator [29, pp 339–340,348–352] executing different Static Strategies depending on overloading or on the names of member functions.

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.

5. 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.

5.1 Also known as

Wrapper Facade [56, pp 66–67]

5.2 Intent

Adapt a series of different interfaces to a common interface. The choice of which interface is actually to be adapted can be done at compile time.

5.3 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 [23], 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 13.

Listing 13: 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 cannot assist much in detecting errors, and maintenance likely becomes a nightmare. So the goal is to reduce conditional compilation to a minimum.

5.4 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.

5.5 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 [10].

5.6 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.

5.7 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 6.

5.7.1 Participants

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

PlatformType A low Layer [11], 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.

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 7 sketches the participants and their relations to each other.

5.7.2 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.

5.7.3 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

Table 6: 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

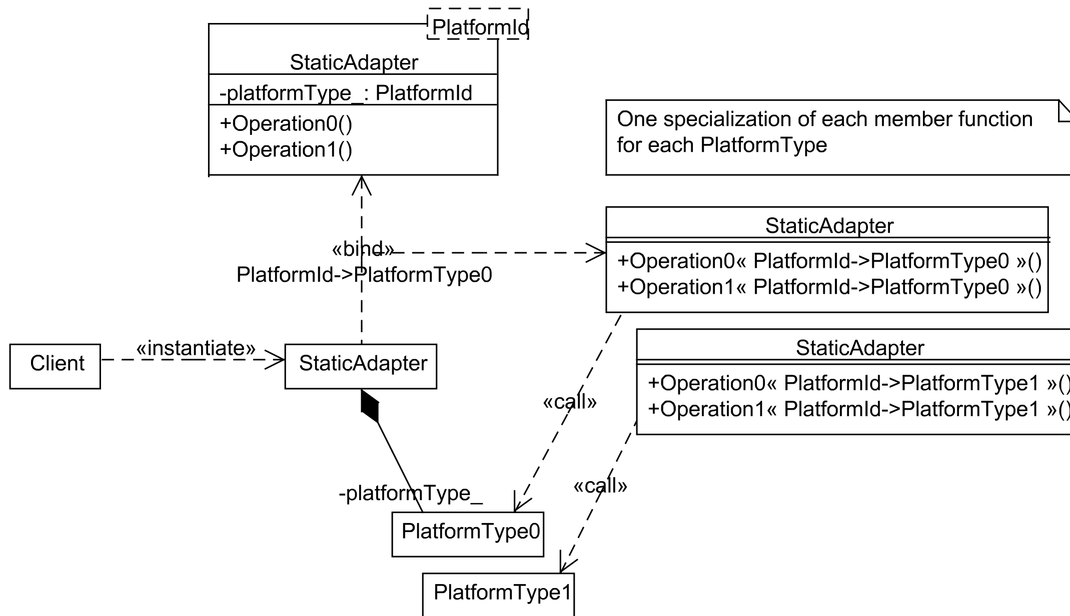

Figure 7: Class diagram illustrating Static Adapter

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 5.10 and 5.11, respectively.

5.8 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.

5.8.1 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 we identified the following implementation specific ones.

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 5.9 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 [47, 73].

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 [6, 14.7.3/5], however.

5.9 Implementation

Static polymorphism is implemented using specialization of member function templates.

5.9.1 Example Resolved

Listing 14 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 14: 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 13. 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 6 as shown in Listing 17. The second purpose is to hide platform specific types from the compilers on all other platforms—otherwise compilation errors are likely.

The Listings 15 and 16 show how the parts proposed above work together to instantiate and use a readers / writer lock.

Listing 15: How to use Static Adapter

```
int main() {
  #if defined(_WIN32)
    typedef ReadersWriter_Mutex<
      CRITICAL_SECTION
```

```
    > rw_mutex;
  #elif defined(UNIX)
    #if defined(RW_LOCKING)
      typedef ReadersWriter_Mutex<
        pthread_rwlock_t
      > rw_mutex;
    #elif defined(NO_RW_LOCKING)
      typedef ReadersWriter_Mutex<
        pthread_mutex_t
      > rw_mutex;
    #else /* defined(RW_LOCKING) */
      #error main(): Fatal error: Missing
        or invalid configuration define.
    #endif /* defined(RW_LOCKING) */
  #else /* defined(_WIN32) */
    #error main(): Fatal error: Platform
      not supported.
  #endif /* defined(_WIN32) */
  rw_mutex lock;
  ...
  lock.writeAcquire();
  ...
  lock.release();
  ...
  return EXIT_SUCCESS;
}
```

The compilation includes the configuration step. Note that conditional compilation fulfills two different purposes here: `NO_RW_LOCKING` and `RW_LOCKING` denote alternatives valid (though not necessarily implemented) on each platform, whereas `_WIN32` and `UNIX` here simply prevent the compiler to fail because of unknown types only defined on some platforms. In Listing 13 these two purposes were interspersed with each other.

Listing 16: How to compile Static Adapter

```
g++ -DRW_LOCKING -DUNIX -c client.cxx
  -o client.o
```

Different from the Adapter pattern the concrete adapter can't be dynamically created using a creational design pattern—the type of the adapter has to be known at compile time. A static parallel to the creational patterns is proposed in Section 6. Using this technique the adapter class could be hidden behind a `typedef` after all.

5.9.2 Relationship of Example and Participants

The code shown as an example above maps to the participants defined in Section 5.7.1 as shown in Figure 7.

5.10 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 [10] suggests the use of Static Strategy (see Section 3) to solve the same problem.

Adaptation to the platform can also happen at the link–editing step. To do so you have to factor out platform specific functionality into static libraries and distribute your application as a collection of static libraries along with an appropriate installation tool. Similar approaches also work for the runtime linker and shared libraries, respectively. The precondition in both cases is that the binary format must not be specific to one platform only.

5.11 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 5.9, nevertheless all of them solve the problem to statically adapt Wrapper Facades to a variety of different platforms.

5.11.1 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.

5.11.2 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.

5.11.3 Boost.Threads

The Boost project contains a set of Wrapper Facades for multithreading [36]. 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.

5.11.4 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 being used instead.

5.11.5 Loki‹library›

The Loki library contains a set of multithreading Wrapper Facades [5, 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 relies on preprocessor constants set differently by the compilers on different platforms or set within platform specific standard header files.

Table 7: Relationship of Code and Participants

Code	Participant
int main()	Client
CRITICAL_SECTION, pthread_mutex_t, pthread_rwlock_t	PlatformType
ReadersWriter_Mutex<>- ::ReadersWriter_Mutex(), ReadersWriter_Mutex<>- ::~ReadersWriter_Mutex(), void ReadersWriter_Mutex<>::readAcquire(), void ReadersWriter_Mutex<>::writeAcquire(), void ReadersWriter_Mutex<>::release()	SpecializationOfMemberFunctions
template<> class ReadersWriter_Mutex	StaticAdapter

5.11.6 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.

5.11.7 Oracle DBMS

During the installation of the Oracle Database Management System on UNIX a so–called linking phase takes place. This is an example for adaptation at link–editing time.

5.11.8 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.

5.12 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.

6. STATIC ABSTRACT TYPE FACTORY

A class based pattern to map types to types.

6.1 Also known as

Generator [17, pp 397–501]
Type Selection [5, pp 65–67]
Type Traits [5, pp 74–83], [67]

6.2 Intent

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

6.3 Example

Different platforms provide different data types for basically the same entity. A POSIX 1003.1c compliant UNIX system represents mutual exclusion locks by the type `pthread_mutex_t`, on MS Win32 `CRITICAL_SECTION` can be taken. Depending on an expression in the domain specific language the correct type should be chosen.

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. One problem with this approach is that the units of configurability are compilation units, a quite coarse entity.

6.4 Context

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

6.5 Problem

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

6.6 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.

6.7 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 8.

6.7.1 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 There's one specialization of Static Abstract Type Factory for each Configuration supported. 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 of defining 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 8 sketches the participants and their relations to each other.

6.7.2 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.

6.7.3 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.

6.8 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.

6.8.1 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 we identified the following implementation specific ones.

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 in a future version of the C++ standard, e.g. [18, 33].

6.9 Implementation

Here the whole class template is going to be specialized. In fact the default class template can be trivial. This Abstract Factory [22] depends on static configuration and creates types.

6.9.1 Example Resolved

Listing 17 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 `CRITICAL_SECTION` 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 17: Portable association of an operating system with certain platform specific types combined with Static Adapter (see Section 5)

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

template< typename OperatingSystem >
struct Multithreading {};
```

Table 8: 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

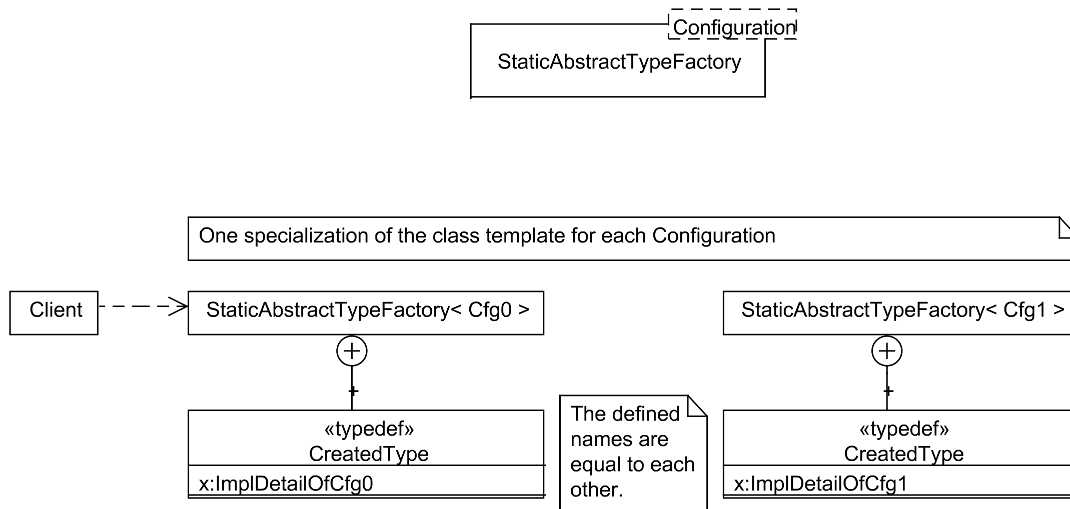

Figure 8: Class diagram illustrating Static Abstract Type Factory

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

template<>
struct Multithreading< MSWin32 > {
  typedef CRITICAL_SECTION 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;
```

6.9.2 Relationship of Example and Participants

The code shown as an example above maps to the participants defined in Section 6.7.1 as shown in Figure 9.

6.10 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 5) are examples for this case; as pointed out in Section 5.7.3 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.

6.11 Known Uses

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

6.11.1 Boost.TypeTraits

Boost.TypeTraits [3] 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.

6.11.2 C++ `std::iterator_traits<>`

The C++ way of Iterators [29] 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.

6.11.3 The Matrix Template Library

The Matrix Template Library [58, 17] 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.

6.12 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.

7. STATIC FRAMEWORK

> 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 [70, p 1355]

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

7.1 Intent

Portable code must meet performance requirements on each platform. 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.

7.2 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 [56] Layer [11] only. Instead experience is made available in terms of Frameworks [50] that use design patterns to allow for adaptation to certain environments. Listing 18 shows a simple class that frees the user from the burden of portable thread handling.

Table 9: Relationship of Code and Participants

Code	Participant
`typedef MSWin32 OS`, `typedef Unix OS`	Configuration
`typedef Multithreading< OS > multithreading_type`	Client
`template<> struct Multithreading< MSWin32 >`, `template<> struct Multithreading< Unix >`	SpecializationOfClassTemplate
`template<> struct Multithreading`	StaticAbstractTypeFactory
Implicit. See liability 2 in Section 6.8.1.	StaticAbstractTypeFactoryConcept

Listing 18: Black–Box Framework

```
// 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 {
  Activation_List act_queue_;
  static impl_type createImpl(size_t);
  explicit Impl(size_t high_water_mark)
    : 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() {
  // Poor men's approach to cancellation
  // - initiate
  impl_->act_queue_.insert(0);
  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 *const impl
    =static_cast< MQ_Scheduler * >(
       arg
     )->impl_;
  for(;;) {
    Method_Request *mr;
    try {
      // Block until the queue is not
      // empty
      impl->act_queue_.remove(&mr);
      // Poor men's approach to
      // cancellation - react
      if(!mr)
        break;
      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 [32] like to the constructor of `Thread_Operation`.

`MQ_Scheduler` is used as an illustration of the Active Object architecture pattern [51, 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 [24] 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 [26]. 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 [28].

More recent versions of the JAWS Adaptive Web System (JAWS) [54, pp 27,47–48], an application closely related to the ADAPTIVE Communication Environment (ACE), are examples for this implementation technique. They use 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 [8, 72]. The base class provides for access to the request processing.

The original `MQ_Scheduler` additionally uses the Template Method design pattern [31] 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 [62, pp 388–393], [61, pp 495–497] as shown in [9].

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.

7.3 Context

A series of applications share implementation similarities not only on a basic Layer, 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.

7.4 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.

7.5 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.

7.6 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 [51]. 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 3) or Static Visitor (see Section 4). Allow for configuration of the code that deals with platform specific interfaces by means of e.g. Static Adapter (see Section 5).

A first sketch of the solution is shown in Table 10.

7.6.1 Participants

Client The Client requests a service from Static Framework. Many clients may request the same service in parallel. Responsiveness or throughput are 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 [50, pp 478–479] of the Static Framework. Conforms to the Static Strategy or the Static Visitor design pattern.

Figure 9 sketches the participants and their relations to each other.

7.6.2 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 10.

7.6.3 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 another implementation, that better conforms to a new platform, for the Static Adapter, 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.

7.7 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.

7.7.1 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 [19] to keep them healthy [46, 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 [17, 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.

Table 10: 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

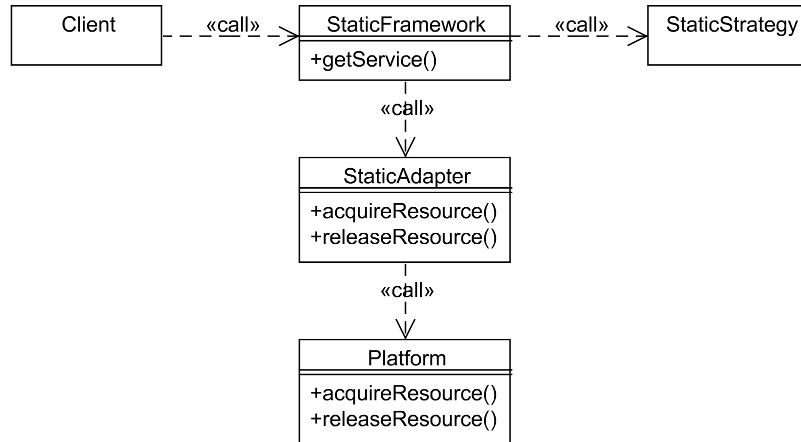

Figure 9: Class diagram illustrating Static Framework

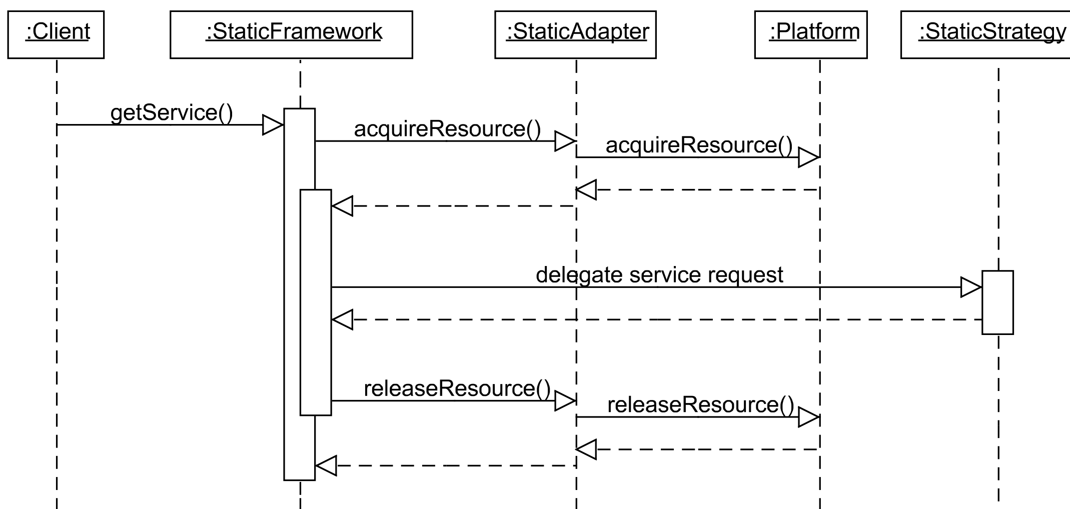

Figure 10: Sequence diagram illustrating Static Framework

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 [50, pp 472–474].

7.8 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.

7.8.1 Example Resolved

Listing 19 shows the class template `MQ_Scheduler<>`. `MQ_Scheduler<>` carries the intention of the example presented in Section 7.2.

Here a static variant of the Command pattern similar to Static Strategy was substituted for the use of the original Command design pattern, and the Adapters used in conjunction with Bridge were replaced by Static Adapters, that determine how to deal with specific Units of Execution [40, pp 217–221].

Listing 19: A Unit of Execution executing a function object

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

class Impl {
  Activation_List act_queue_;
public:
  struct command_adapter {
    virtual ~command_adapter();
    virtual void call() =0;
  };
  template< typename Command >
  class command_proxy
    : public command_adapter {
    Command &command_;
  public:
    explicit command_proxy(
        Command &command
    ) : command_(command) {}
    void call() {
      command_.call();
    }
  };
  explicit Impl(size_t);
  template< typename Command >
  void insert(
    const Command &method_request
  ) {
    act_queue_.insert(new command_proxy<
      Command
    >(method_request));
  }
};

template< typename UE > class MQ_Scheduler
  : public Impl {
  UE thread_;
  void createUE();
  void joinUE();
public:
  explicit MQ_Scheduler(
      size_t high_water_mark
  )
    : Impl(high_water_mark) {
    createUE();
  }
  ~MQ_Scheduler() {
    // Poor men's approach to cancellation
    // - initiate
    act_queue_.insert(0);
    joinUE();
  }
};

// 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 *const impl
    =static_cast< Impl * >(arg);
  for(;;) {
    Impl::command_adapter *mr;
    try {
      // Block until the queue is not
      // empty
      impl->act_queue_.remove(&mr);
      // Poor men's approach to
      // cancellation - react
      if(!mr)
        break;
      mr->call();
    }
    catch(...) {
    }
    delete mr;
  }
  return 0;
}

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

Impl::Impl(size_t high_water_mark)
  : act_queue_(high_water_mark) {}

#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.

7.8.2 Relationship of Example and Participants

The code shown as an example above maps to the participants defined in Section 7.6.1 as shown in Figure 3.

7.9 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 [60, pp 732–736] and One Thread per Client [60, pp 752–753] which do not map well to Active Objects.

7.10 Known Uses

Examples of Static Framework can be found in existing software.

7.10.1 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 [52] or Leader / Followers [53], [60, 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.

7.11 Related Patterns

Though Model–View–Controller [12], Presentation–Abstraction–Control [14], and Separation of Powers [49, pp 24–26] relate to user interfaces, hence another domain than Static Framework, all these patterns separate software into classes with higher likelihood 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 [45], 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++ [62, pp 388–393], [61, pp 495–497], [9, 15, 55] or the Dispose pattern in C# [43] and Java [7, pp 228–230], [1], see further [35, pp 6–7].

8. 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.

The author would like to thank all the participants of PLoP 2006 for their contributions. Special thanks go to the members of the Writers' Workshop [21] "Intimacy Gradient" [4, pp 610–613] the author participated in: Ademar Aguiar, Kanwardeep Singh Ahluwalia, Sachin Bammi, Andrew P. Black, Danny Dig, Brian Foote, Anders Janmyr, June Kim, Ricardo Lopez, Maurice Rabb, Mirko Raner, Errol Thompson, Daniel Vainsencher and last but not least both Robert S. Hanmer and Ward Cunningham, its moderators. Their

Table 11: Relationship of Code and Participants

Code	Participant
Not shown.	Client
MS Windows API (e.g. `uintptr_t _beginthreadex()`), represented by `HANDLE`, and POSIX Threads API (e.g. `int pthread_create()`), represented by `pthread_t`	Platform
`template<> class MQ_Scheduler` without `Impl`, its base class	StaticAdapter
`template<> class MQ_Scheduler`	StaticFramework
Any possible substitute for `Command`	StaticStrategy

feedback had a significant impact on the current version of Static Visitor and Static Abstract Type Factory.

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 in part by the Institute for Medical Informatics and Biostatistics, Basel, Switzerland.

9. REFERENCES

[1] Releasing resources in Java. Retrieved January 9, 2007, from `<http://www.c2.com/cgi/wiki?-ReleasingResourcesInJava>`.

[2] D. Abrahams and A. 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..., Jan. 2005.

[3] Adobe Systems Inc., D. Abrahams, S. Cleary, B. Dawes, A. Gurtovoy, H. Hinnant, J. Jones, M. Marcus, I. Maman, J. Maddock, T. Ottosen, R. Ramey, and J. Siek. Boost.TypeTraits. Retrieved January 9, 2007, from `<http://www.boost.org/doc/-html/boost_typetraits.html>`.

[4] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language. Towns, Buildings, Construction. With* Max Jacobson, Ingrid Fiksdahl–King *and* Shlomo Angel. Oxford University Press, New York, 1977.

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

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

[7] K. Arnold, J. Gosling, and D. 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".

[8] K. Auer. Reusability through self–encapsulation. In Coplien and Schmidt [16], chapter 27, pages 505–516.

[9] P. Bachmann. 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*, Dec. 2005. Retrieved January 9, 2007, from `<http://hillside.net/plop/2005/proceedings/-PLoP2005_pbachmann1_0.pdf>`.

[10] W. Blochinger and W. 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*, Aug. 2003. Retrieved January 9, 2007, from `<http://www-sr.informatik.uni-tuebingen.de/-~bloching/papers/pdcs03.pdf>`.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Layers*, chapter 2: Architekturmuster, pages 32–53. In [13], 1998, 1. korr. nachdruck, 2000. German translation of "Layers".

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Model–View–Controller*, chapter 2: Architekturmuster, pages 124–144. In [13], 1998, 1. korr. nachdruck, 2000. German translation of "Model–View–Controller".

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. 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, 2000. German translation of "Pattern–Oriented Software Architecture. A System of Patterns".

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Presentation–Abstraction–Control*, chapter 2: Architekturmuster, pages 145–169. In [13], 1998, 1. korr. nachdruck, 2000. German translation of "Presentation–Abstraction–Control".

[15] T. Cargill. Localized ownership: Managing dynamic

objects in C++. In Vlissides et al. [69], chapter 1, pages 5–18.

[16] J. O. Coplien and D. 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.

[17] K. Czarnecki and U. 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.

[18] G. Dos Reis and B. 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*, Jan. 2006. Retrieved January 9, 2007, from `<http://www.research.att.com/~bs/popl06.pdf>`.

[19] M. Fowler. *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.

[20] E. Friedman and I. Maman. Boost.Variant. Retrieved January 9, 2007, from `<http://www.boost.org/doc/html/variant.html>`.

[21] R. P. Gabriel. *Writers' Workshops & the Work of Making Things. Patterns, Poetry....* Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 2002.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Abstrakte Fabrik*, chapter 3: Erzeugungsmuster, pages 107–118. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Abstract Factory".

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Adapter*, chapter 4: Strukturmuster, pages 171–185. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Adapter".

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Befehl*, chapter 5: Verhaltensmuster, pages 273–286. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Command".

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Besucher*, chapter 5: Verhaltensmuster, pages 301–318. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Visitor".

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Brücke*, chapter 4: Strukturmuster, pages 186–198. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Bridge".

[27] E. Gamma, R. Helm, R. Johnson, and J. 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, 1996. German translation of "Design Patterns. Elements of Reusable Object–Oriented Software".

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Fabrikmethode*, chapter 3: Erzeugungsmuster, pages 131–143. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Factory Method".

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Iterator*, chapter 5: Verhaltensmuster, pages 335–353. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Iterator".

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Kompositum*, chapter 4: Strukturmuster, pages 239–253. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Composite".

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Schablonenmethode*, chapter 5: Verhaltensmuster, pages 366–372. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Template Method".

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Strategie*, chapter 5: Verhaltensmuster, pages 373–384. In *Professionelle Softwareentwicklung* [27], dritter, unveränderter nachdruck, 1996. German translation of "Strategy".

[33] D. Gregor and B. 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. Retrieved January 9, 2007, from `<http://www.open-std.org/-jtc1/sc22/wg21/docs/papers/2006/n2042.pdf>`.

[34] N. Harrison, B. Foote, and H. 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.

[35] K. Henney. Executing around sequences. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP) 2000, Irsee, Germany*, July 2000. Retrieved January 9, 2007, from `<http://hillside.net/europlop/HillsideEurope/-Papers/ExecutingAroundSequences.pdf>`.

[36] W. E. Kempf. Boost.Threads. Retrieved January 9, 2007, from <http://www.boost.org/doc/html/threads.html>.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. 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.

[38] R. C. Martin. Acyclic visitor. In Martin et al. [39], chapter 7, pages 93–103.

[39] R. C. Martin, D. Riehle, and F. 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.

[40] T. G. Mattson, B. A. Sanders, and B. 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, Sept. 2004.

[41] S. Meyers. *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.

[42] S. Meyers. *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 [41], third edition, 2005.

[43] Microsoft Developer Network (MSDN). .NET framework general reference: Common design patterns. implementing finalize and dispose to clean up unmanaged resources. Retrieved January 9, 2007, from <http://msdn.microsoft.com/library/en-us/-cpgenref/html/cpconfinalizedispose.asp>, 2005.

[44] M. Poppendieck and T. 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.

[45] M. Poppendieck and T. 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 [44], 2003.

[46] M. Poppendieck and T. Poppendieck. *Tool 19: Refactoring*, chapter 6: Build Integrity In, pages 140–145. In *The Agile Software Development Series; ed. by* Alistair Cockburn, Jim Highsmith [44], 2003.

[47] M. Poppendieck and T. 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 [44], 2003.

[48] E. S. Raymond. *The Art of UNIX Programming*. Addison–Wesley Professional Computing Series; ed. by Brian W. Kernighan. Addison Wesley Professional, Boston, Massachusetts. . . , Sept. 2003. Retrieved January 9, 2007, from <http://www.catb.org/~esr/writings/taoup/>.

[49] D. Riehle and H. Züllighofen. A pattern language for tool construction and integration based on the tools and materials metaphor. In Coplien and Schmidt [16], chapter 2, pages 9–42.

[50] D. Roberts and R. Johnson. Patterns for evolving frameworks. In Martin et al. [39], chapter 26, pages 471–486.

[51] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Active Object*, chapter 5: Nebenläufigkeit, pages 411–443. In [54], 2002. German translation of "Active Object".

[52] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Half–Sync / Half–Async*, chapter 5: Nebenläufigkeit, pages 473–497. In [54], 2002. German translation of "Half–Sync / Half–Async".

[53] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Leader / Followers*, chapter 5: Nebenläufigkeit, pages 499–528. In [54], 2002. German translation of "Leader / Followers".

[54] D. Schmidt, M. Stal, H. Rohnert, and F. 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".

[55] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Scoped Locking*, chapter 4: Synchronisation, pages 359–367. In [54], 2002. German translation of "Scoped Locking".

[56] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Wrapper–Facade*, chapter 2: Dienstzugriff und Konfiguration, pages 53–84. In [54], 2002. German translation of "Wrapper Facade".

[57] J. Siek, L.-Q. Lee, and A. Lumsdaine. The Boost Graph library. Retrieved January 9, 2007, from <http://www.boost.org/libs/graph/doc/-table_of_contents.html>.

[58] J. G. Siek. 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, Apr. 1999. Retrieved January 9, 2007, from <http://osl.iu.edu/download/-research/mtl/papers/thesis.pdf>.

[59] J. G. Siek, L.-Q. Lee, and A. 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.

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

[61] B. Stroustrup. *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++”.

[62] B. Stroustrup. *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 edition, 1998. German translation of “The C++ Programming Language, Third Edition”.

[63] H. Sutter. G[uru] o[f] t[he] w[eek] #16. maximally reusable generic containers. Retrieved January 9, 2007, from `<http://www.gotw.ca/gotw/016.htm>`.

[64] H. Sutter. G[uru] o[f] t[he] w[eek] #62. smart pointer members. Retrieved January 9, 2007, from `<http://www.gotw.ca/gotw/062.htm>`.

[65] D. Vandevoorde and N. 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.

[66] D. Vandevoorde and N. M. Josuttis. *Function Objects and Callbacks*, chapter 22, pages 417–474. In [65], June 2003.

[67] D. Vandevoorde and N. M. Josuttis. *Type Classification*, chapter 19, pages 347–364. In [65], June 2003.

[68] J. Vlissides. *Entwurfsmuster anwenden. Deutsche Übersetzung von* Michael Timm. Professionelle Softwareentwicklung. Addison–Wesley–Longman, München · Reading, Massachusetts · Menlo Park, California · New York · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 1999. German translation of “Pattern Hatching. Design Patterns Applied”.

[69] J. M. Vlissides, J. O. Coplien, and N. 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.

[70] N. Wiener. 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.

[71] L. Wittgenstein. *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>`, retrieved January 9, 2007.

[72] B. Woolf. Abstract class. In Harrison et al. [34], chapter 1, pages 5–14.

[73] E. Zaninotto. Keynote “from X programming to the X organisation. 3rd international conference on extreme programming, may, 26–29 alghero”. Retrieved January 9, 2007, from `<http://ciclamino.dibe.unige.it/xp2002/talksinfo/zaninotto.pdf>`, 2002.