Configurable Binding: How to Exploit Mixins and Design Patterns for Resource-Constrained Environments

Sven Apel[1], Helge Sichting[1], and Klemens Böhm[2]

[1] Department of Computer Science
University of Magdeburg, Germany
email: apel@iti.cs.uni-magdeburg.de,
sichting@cs.uni-magdeburg.de
[2] Department of Computer Science
University of Karlsruhe, Germany
email: boehm@ipd.uka.de

Abstract. Realtime and embedded systems are subject to strict requirements on performance and resource consumption. However, modern software-engineering methods do not always allow to meet these technical requirements. More specifically, common approaches for the design of flexible, reusable and customizable software lead to increased memory consumption and reduced performance. This article proposes a configurable binding mechanism that addresses this important problem. It adopts features of design patterns and mixins. Our proposal is not obvious – the difficulty is to arrange the structural elements of design patterns and mixins so that the binding is configurable and copes with those requirements specific to realtime and embedded systems. Measurements show that our approach indeed accomplishes this. At the same time, it provides the known virtues of modern software-engineering methods, in particular configurability and reusability. Finally, we say how to apply our approach to other common design and implementation methodologies, e.g., program families, frameworks, and aspects.

1 Introduction

Today approximately 98% of all computers are deployed as embedded systems [32]. The market is growing rapidly [8]. Future scenarios like ubiquitous computing will push this trend [35]. Currently, software development for embedded systems focuses on technical aspects like performance and resource constraints, but neglects software-engineering requirements, e.g., reusability, customizability, and extensibility. However, these requirements are extremely important for embedded systems as well.

Design patterns are one common method to build configurable, reusable and extensible software [15, 6]. They provide standard solutions for recurring design problems. But with many design patterns, the downside is performance degradation and higher resource consumption [15, 12, 4]. This is due to the use of abstract

classes and methods as well as delegation and the higher degree of indirection. [4] shows that the elimination of unneeded abstract classes reduces memory space consumption by 4.8% and runtime by 23.3% for a specific but representative scenario. Thus there often is a tension between software-engineering requirements and technical requirements.

To overcome this tension, we envision a binding mechanism that the application programmer can configure. The type of the binding is a parameter: If the programmer configures the system such that assembly of the concrete classes from patterns takes place at compile-time (*early binding*), the configuration process will eliminate abstract classes and delegations. This is good with regard to resource consumption and performance. With the other configuration, the implementation is bound at runtime parametrically (*late binding*). In other words, the programmer trades performance etc. for flexibility. The configurations differ only in the point of time when the concrete classes and their methods are bound to the context, but not with regard to the interface. We refer to this as *binding time* in what follows. Furthermore, we perceive the binding as a separate concern. It must be explicit and separated from the implementation of the application logic, to facilitate its configuration. This is in line with the well-known separation of concerns [26, 21].

The seamless integration of such a binding mechanism into known design methodologies with a focus on reusability and configurability, e.g., program families, frameworks, aspects and collaborations, is not obvious. This is because these methodologies use different mechanisms for the (de)composition and encapsulation of modules as well as for their configuration and reuse. Designing a general binding concept which works for all these mechanisms is challenging. E.g., program families and their members typically are composed either at compile time or at runtime. Common implementation techniques do not support both. To give another example, aspect-oriented programming (AOP) focuses on the separation of crosscutting concerns. To our knowledge, research has not addressed the question how to make the binding explicit with AOP. This article introduces a configurable binding mechanism. We say how to combine it with those methodologies without constraining their flexibility or applicability. In short, we model variation points (e.g., abstract classes, template classes) and concrete classes as mixins, introduce separate interfaces for early and late binding and use a dispatcher for delegation. – At the same time, the technical requirements (performance, low memory consumption) prevail. We show that our approach is just as good as the native implementation methodologies in these respects.

Section 2 discusses the prospective impact of the type of binding. Section 3 introduces relevant design and implementation techniques. Section 4 then describes our configurable binding mechanism. Section 5 features an experimental evaluation. Section 6 says how to restructure object-oriented programs in general in order to make the binding configurable and discusses the applicability of our configurable binding to program families, frameworks, aspects, and design patterns. Section 7 reviews related work. Section 8 concludes.

2 The Impact of the Binding Time

The binding type – early or late binding – impacts flexibility and configurability of the software system as well as its memory footprint and performance. This section reviews known binding variants, based on an example. The example is adopted from the *strategy design pattern* (cf. [15]): The objective is to separate an object (*context*) from its behavior. The context uses different strategies, and each strategy encapsulates one type of behavior. For instance, think of a byte buffer (context) which encrypts the data stored. Strategies encapsulate the various encryption algorithms. Replacement of strategies is possible either at compile-time or at runtime (see Fig. 1).

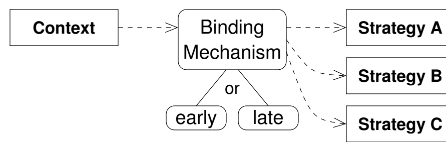

Fig. 1. Binding strategies to the context at different times.

Compile-time – early binding. This variant binds a strategy to the context early. This implies that strategies cannot be exchanged at runtime. The context cannot alter its behavior dynamically. The advantage of this variant is reduced resource consumption. This is because no additional references, method calls, or classes are needed.

Runtime – late binding. This variant binds the strategies late. The programmer uses abstract classes and a level of indirection to implement this binding mechanism. The context can now alter its behavior dynamically. But the use of abstract classes and indirect invocation increase the memory footprint and reduce performance.

In short, both binding mechanisms have their advantages and disadvantages. It depends on the application scenario which one is adequate. Due to the resource-constrainedness and the importance of performance, early binding is indispensable in realtime embedded systems. As mentioned, we see the binding time as a non-functional concern: If one wants to alter the binding time flexibly, he has to make it explicit. To do so, we will propose a binding mechanism that is configurable. An important objective is to avoid the redundant implementation of application logic classes. In other words, the programmer should be able to reuse implementations of concrete classes, irrespective of the binding time.

3 Basic Design and Implementation Techniques

This section classifies relevant implementation techniques by their binding type and assesses the degree of configurability and reusability offered. The implemen-

tation techniques under consideration are the following ones: (1) direct association of context and concrete classes, (2) design patterns which use late binding, and (3) template classes which use early binding. We continue to use the strategy pattern as our running example.

Association. A simple way to separate objects and their behavior is to model the behavior (the strategy) as a separate class and associate it *directly* with the object (the context). We distinguish between two cases, aggregation and association [24]. With aggregation, the strategy is hard-wired with the context. With association, the object owns a reference to a strategy. The first case supports no configuration. The second one allows the replacement of the strategy, but the new strategy has to be type-equivalent. In both cases, reusability of the strategy implementations is low. This is because interface and implementation are not separated (cf. [15]).

Design patterns. The strategy design pattern provides a better solution for the separation of objects and their behavior. Figure 2 shows the corresponding class diagram.[3] The context owns a *reference/pointer* to the interface *AbstractStrategy*. *AbstractStrategy* is a *polymorphic type* [31]. Several concrete strategies implement it using inheritance. Given this polymorphic coherence, the context can invoke the concrete strategies via the same interface. Thus, the programmer can apply the strategies to the context at runtime by replacing the reference of the context. The downside of this implementation technique is the late binding. Virtual function-pointer tables and additional pointers/references increase the memory consumption, and the indirect invocation mechanism based on function pointers decreases the performance. [12] presents a quantitative analysis of this issue for C++. These drawbacks prevent the use of pattern-based software in resource-constrained environments.

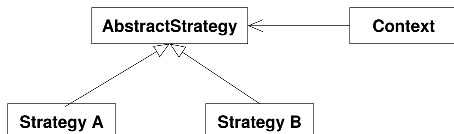

Fig. 2. The strategy design pattern.

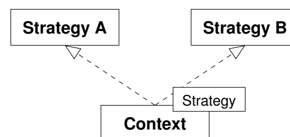

Fig. 3. Exchangeable strategies using mixins.

Template classes and mixins. Template classes allow to implement the separation of context and its strategy as well. The term 'template' is quite overloaded in computer science. We use it similarly to the template construct in C++. Templates allow *generic programming* by the parameterization of classes with types [31]. Templates are typically used to implement

[3] For all diagrams, we use *UML* (see `www.uml.org`).

```
int main() {
    Context<StrategyA> context1; // using strategy A
    Context<StrategyB> context2; // using strategy B
}
```

Fig. 4. Instantiation of the context using different strategies.

data structures which work with various data types. The type is determined at compile-time. Figure 3 shows the template-based implementation of the strategy example. By instantiating the template, the programmer decides which strategy the context refers to (cf. Fig. 4). The context inherits from the strategy chosen. This mechanism is called *mixin-based inheritance*, and the template classes are named *mixins* [5]. In the figure, the dashed arrows indicate that the context might inherit from the base class. In contrast to the pattern-based variant, no resources are wasted, and there is no performance overhead. Moreover, modern compilers are able to *inline* calls to templates classes, because the type is known at compile-time [31]. With abstract classes in turn, this is not the case. On the other hand, the programmer must choose the strategy at compile-time.

All three variants are efficient solutions for specific application requirements. However, due to the inadequate support of configuration and reuse, we do not consider the direct association approach in what follows. The template variant provides compile-time configurability only, but exhibits good performance and has a small memory footprint. The use of abstract classes, adopted by design patterns, is flexible, but results in weaker performance and higher resource consumption. To combine the advantages of these approaches, we will integrate them in a new, more abstract concept. Furthermore, classes of the application logic shall be reusable, regardless of the binding variant in use. This is not possible with a combination of templates and patterns, due to the different interfaces of the concrete strategy classes. The pattern-based approach requires an interface that declares all methods of the strategies as virtual/abstract. This is different from the template approach. In other words, templates and patterns cannot be exchanged mutually. This is because the application logic classes are not reusable.

This article shows how to combine the advantages of both approaches without loosing on applicability. We make the binding mechanism explicit and therefore configurable. Our mechanism allows to reuse functional concerns (the application logic), irrespective of the binding.

4 The Configurable Binding Mechanism

This section introduces the new notion of *configurable binding*. We first provide a general description, followed by an illustration using the strategy pattern.

4.1 General Concept

Starting point of our description are the differences of mixin layers and design patterns in terms of flexibility and configurability as well as performance and memory consumption. Concepts such as abstract classes, indirect invocation, and delegation yield high flexibility and runtime configurability. On the other hand, they incur performance penalties and higher memory consumption. Our approach uses these concepts, but allows for their easy elimination in case runtime configurability is not needed. Compared to existing approaches, it is now the programmer who makes this configuration decision. Figure 5 depicts the structural components that allow for runtime configurability. (So far, the situation depicted is identical to the pattern approach.) The goal is to make the binding configurable if the programmer wants to have runtime configurability, or, alternatively, use mixin techniques to make the classes exchangeable at compile-time. To ease the presentation, we describe our approach as a sequence of four steps.

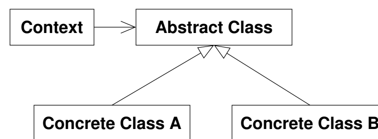

Fig. 5. Using abstract classes to gain flexibility and configurability.

1. In the first step, the context is implemented as a mixin. It inherits from its template parameter (cf. Fig. 6). The dashed arrows stand for inheritance relations that are configurable. The instantiation of the mixins specifies which relation is actually established (by mixin-based inheritance, cf. Sec. 3). Doing so, the programmer can instantiate the context with concrete classes. Thus, the context has full access to the attributes and methods of the class. After this first step, the application programmer can exchange concrete class implementations at compile-time by substituting the template parameter.

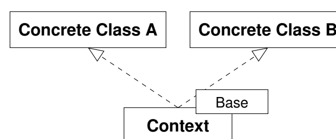

Fig. 6. Implementing the context as mixin.

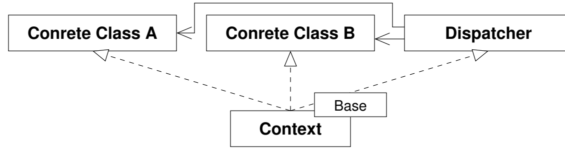

Fig. 7. Introducing the dispatcher for runtime configurability.

2. The second step introduces a dispatcher (cf. Fig. 7). It provides runtime configurability. It has the same interface as the concrete classes. So, the programmer can use the dispatcher instead of concrete classes. Moreover, the dispatcher owns a reference to a concrete class. The dispatcher delegates incoming calls to the concrete classes, using the reference.

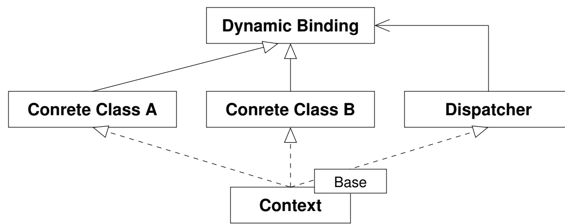

Fig. 8. Introducing a unique interface for all concrete classes.

3. To provide a unique interface for all concrete classes, this step introduces an interface (*Dynamic Binding*) (cf. Fig. 8). It declares all methods of the concrete classes as virtual/abstract. The reference of the dispatcher points to this interface.

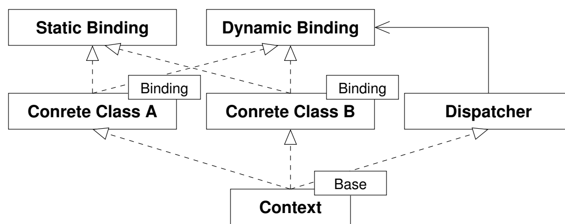

Fig. 9. Implementing the concrete classes as mixins.

4. Because the interface is only needed for dynamic configurability, the concrete classes inherit from the interface only in this case. To accomplish this, the concrete classes are implemented as mixins (cf. Fig. 9). In the dynamic case, they are instantiated using the interface *Dynamic Binding*. In the static case, they inherit from the empty interface *Static Binding*.

Figure 10 shows the resulting class hierarchy of late binding. We call it the *dynamic case* in the following. The figure shows that the dispatcher delegates calls from the context to the concrete class. The programmer can exchange concrete classes at runtime by resetting the reference of the dispatcher. The *Dynamic Binding* interface facilitates this. Figure 11 depicts the class hierarchy in the case of early binding. In what follows, we refer to it as the *static case*. The programmer instantiates the context directly using the concrete classes. In this case, no additional interface is needed. The concrete classes inherit from the empty interface *Static Binding*.

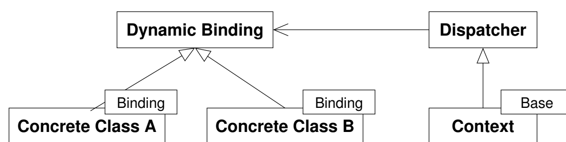

Fig. 10. The dynamic case – late binding of concrete classes.

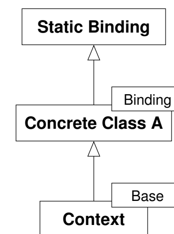

Fig. 11. The static case – early binding of concrete classes.

4.2 Application to the Strategy Pattern

To illustrate our binding mechanism, we apply it to the strategy pattern. We will refer to the result as *extended strategy pattern*. To provide even more insight, we use a concrete scenario: A communication client sends and receives data. The client uses different strategies to do so. It supports two concrete strategies: *SyncSending* sends and receives data in a synchronous way. *AsyncSending* does the same in an asynchronous way (cf. Fig. 12). The code snippets are in C++. This is realistic, because C++ is the dominant language for realtime and embedded systems. But every language which supports inheritance, abstract classes, and templates could be used, e.g., Eiffel, Ada, as well as generic extensions to Java and C#. Based on this scenario, we explain how to integrate the binding mechanism into the extended strategy pattern.

According to the first step, the client is implemented as a mixin. It inherits from *SyncSending* or *AsyncSending*. This depends on the instantiation of the

```
1   class SyncSending {
2   public:
3       int send(char *buf) {/* synchronous sending */}
4       int recv(char *buf) {/* synchronous receiving */}
5   };
6
7   class AsyncSending {
8   public:
9       int send(char *buf) {/* asynchronous sending */}
10      int recv(char *buf) {/* asynchronous receiving */}
11  };
```

Fig. 12. Two concrete sending strategies.

```
1   template <class StrategyType>
2   class Client : public StrategyType {
3   ...
4   };
5   int main() {
6       Client<SyncSending> c1;  // early binding of synchronous sending
7       c1.send("Hello World");  // synchronous sending
8       Client<AsyncSending> c2; // early binding of asynchronous sending
9       c2.send("Hello World");  // asynchronous sending
10  }
```

Fig. 13. The static case – early binding of strategies.

template parameter (cf. Fig. 13, Lines 6, 8). Figure 13 gives the code of the client mixin and the two instances. Client *c1* uses a synchronous sending mechanism (Lines 6–7), *c2* an asynchronous one (Lines 8–9).

As a second step, we introduce a dispatcher. It has the same interface as the common strategies. It owns a pointer to a strategy object and a method to set the strategy at runtime (cf. Fig. 14). If the programmer wants to exchange strategies at runtime, he uses the class *Dispatcher* instead of concrete strategies to instantiate the client. The client calls the *send*/*recv* method, as in the other case. The dispatcher delegates calls to the strategy object currently registered. By invoking *setStrategy* on the dispatcher, the programmer can register and exchange strategy implementations at runtime.

In the third step we introduce an interface for the dynamic case. It denotes all methods of the strategies as virtual (see Fig. 15, Lines 1–5). In the static case, this is not necessary and not desired. To reuse strategy implementations in the static as well as in the dynamic case, the interface for the dynamic case must be

```
1   class Dispatcher {
2   protected:
3       DynamicBinding *m_strategy;
4   public:
5       int send(char *buf) {return m_strategy->send(buf);}
6       int recv(char *buf) {return m_strategy->recv(buf);}
7       void setStrategy(DynamicBinding *s) {m_strategy = s;}
8   };
```

Fig. 14. The dispatcher allows runtime exchangeability of strategies.

```
class DynamicBinding {
public:
    virtual int send(char *buf) = 0;
    virtual int recv(char *buf) = 0;
};

class StaticBinding {
};
```

Fig. 15. The interface for dynamically exchangeable strategies.

```
template <class Binding>
class SyncSending : public Binding {
public:
    int send(char *buf) {/* synchronous sending */}
    int recv(char *buf) {/* synchronous receiving */}
};

template <class Binding>
class AsyncSending : public Binding {
public:
    int send(char *buf) {/* asynchronous sending */}
    int recv(char *buf) {/* asynchronous receiving */}
};
```

Fig. 16. Concrete strategies implement a binding interface by mixin-based inheritance.

different from the one for the static case: For the dynamic case, the strategies have to implement the interface *DynamicBinding*. This is not necessary in the static case and therefore their interface remains unchanged.

The fourth step overcomes the problem of declaring an interface for the dynamic and another for the static case. For that, we implement strategies as mixins (cf. Fig. 16). Each strategy inherits from the template parameter *Binding*. In the dynamic case, the programmer replaces this parameter by the interface *DynamicBinding* (cf. Fig. 15, Lines 1–5). In the static case, the programmer uses the empty interface *StaticBinding* (see Fig. 15, Line 7). Thus, virtual methods are used, and indirect invocations (by invoking the dispatcher) occur only in the dynamic case. The static case avoids this overhead. This is because the client is instantiated directly with the concrete strategy implementation. Moreover, there are no virtual methods. Figure 17 shows the class diagram of the extended strategy pattern.

The following paragraphs say how to use our pattern at code level:

Static case. Figure 18 (Lines 1–4) shows how to use our pattern to implement and (re)use strategies using early binding. This code snippet is similar to Figure 13. We instantiated two client objects using two concrete strategies (Lines 1 and 3). We instantiated the strategies in turn using the empty interface *StaticBinding*. Thus, there is no runtime overhead or increased resource consumption, but concrete strategies cannot be reset at runtime. Figure 19 depicts the class hierarchy of the static case using early binding.

Dynamic case. Figure 18 (Line 6–12) depicts the use of our pattern for dynamic reconfiguration of strategies. The dispatcher instantiates the client

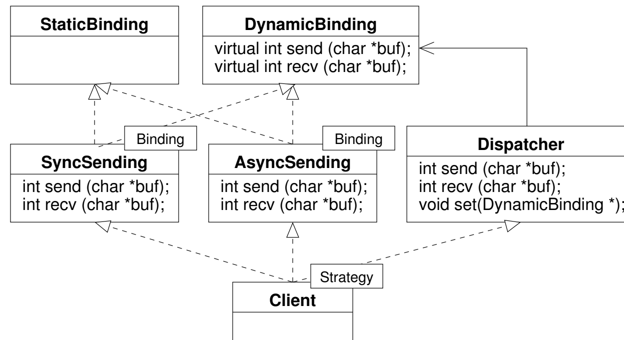

Fig. 17. The implementation of communication client using the extended pattern.

```
1   Client<SyncSending<StaticBinding> > c1;   // synchronous sending
2   c1.send("Hello World");
3   Client<AsyncSending<StaticBinding> > c2;  // asynchronous sending
4   c2.send("Hello World");
5
6   Client<Dispatcher> c1;                // instantiation using the Dispatcher
7   SyncSending<DynamicBinding> sync;     // creating an exchangeable strategy
8   AsyncSending<DynamicBinding> async;   // creating an exchangeable strategy
9   c1.setStrategy(&sync);                // use synchronous sending
10  c1.send("Hello World");
11  c1.setStrategy(&async);               // use asynchronous sending
12  c1.send("Hello World");
```

Fig. 18. Instantiation of strategies for sending using early and late binding.

(Line 6). By invoking the method *setStrategy()*, the programmer can exchange the strategy at runtime. This method is inherited from the dispatcher and therefore only available in the dynamic case. In Lines 7 and 8, two different strategy objects are allocated. They inherit from the *DynamicBinding*-interface by template instantiation. Thus their methods are virtual. The effect is that strategies can be easily exchanged at runtime (Lines 9, 11). Figure 20 depicts the class hierarchy of the dynamic case using late binding.

5 Measurements

The previous section has shown that our extended strategy pattern provides a configurable binding mechanism. This section compares our extended pattern to the native variants, the mixin-based and standard pattern approach. A standard mixin-based implementation is the natural reference point to compare our early-binding configuration to experimentally. When using late binding, we compare our extended pattern to a standard pattern-based solution.

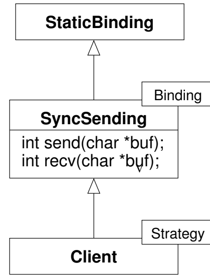

Fig. 19. The static case – early binding of sending strategies.

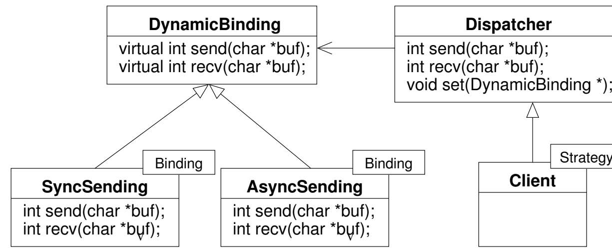

Fig. 20. The dynamic case – late binding of sending strategies.

5.1 Memory Footprint

We have implemented the traditional variants and our extended pattern in C++. The code was compiled on a linux system (Athlon XP 2600+, Kernel 2.4) using the compiler g++ (GCC) 3.3.1. Each variant was compiled three times – (1) without optimization (`-c`), (2) with highest optimization level (`-O3 -c`), and (3) with optimization of the code size (`-Os -c`). We have created object files instead of executable binaries to avoid the overhead of wrapper and startup code. After the compilation we have stripped the symbol table using the *strip* command and determined the size of the code (using *size* on the compiled object files).

Table 1 contains our measurements of the standard strategy pattern and of the standard mixin variant (cf. Sec. 3). The results in the table are expected. The template variant has a significantly smaller memory footprint than the pattern variant. The results refer to one context and two strategies. Except for the pointers, methods, and template-parameters required, all implementations are empty. The context was instantiated only once. We used only one pattern or template for the measurements, respectively. Naturally, the absolute difference between template-based and pattern-based implementations becomes larger if templates and patterns are used several times.

	-c	-O3 -c	-Os -c
mixins	168	7	7
pattern	459	203	204

Table 1. Code sizes of the traditional implementations in byte.

	-c	-O3 -c	-Os -c
static case	178	12	7
dynamic case	616	225	267

Table 2. Code sizes of the extended pattern-variants in byte.

Table 2 shows the code size of our extended strategy pattern. We calculated the footprint of the static and of the dynamic case (cf. Sec. 4). The table tells us that memory consumption in the static case is much less, compared to the dynamic variant. In both settings, resource consumption is similar to the one of the native counterparts. The small overhead is due to the additional use of templates. – This result is a success which we have not expected. We rather had assumed a similar difference between the dynamic and the static variants, compared to the native implementations. But we have not thought that the footprint will not vary significantly.

5.2 Performance Measurements

Early binding has a better performance compared to late binding (approximately 20% [12, 4]). We have conducted performance measurements, in order to reveal the effect of our configurable binding mechanism on performance in quantitative terms. More specifically, we wonder if performance of our implementations is comparable to the native variants, if the better performance of early binding is preserved, and if the overhead of late binding is acceptable.

For the performance measurements we have kept the settings of the footprint measurements, except for the following modifications: To determine the performance of the different implementation variants, we used functions of the GNU C library (declared in `sys/time.h`). To obtain comparable results we created one message for each variant and called the send and receive methods 1.000.000 times (using a *for*-loop). We need the iterations to overcome the limited resolution of the timer. Moreover, real-world applications execute pattern code repeatedly, e.g., container iterators or sending strategies for sockets. The methods of our strategies were empty except for one pointer assignment. To minimize the impact of side effects and inaccuracies, we calculated the total execution time (for 1.000.000 iterations) of each variant ten times. Table 3 shows the average execution times of the native implementations. As expected, the mixin variant is 20% faster than the standard pattern variant on average. Table 4 shows the average execution time of our extended pattern. The static variant is approximately 25% faster than the dynamic variant. These results are in line with the observations in [12, 4]. An important new result is that the performance of the variants of our extended pattern is comparable to their native-implementation counterparts. The additional delegation from the dispatcher to the concrete strategies causes the negligible overhead of the dynamic case compared to the standard pattern. The static variant of our extended pattern preserves the advantages of early binding. The overhead of the dynamic variant is acceptable compared to the native dynamic variant. As with the memory footprint, the impact on performance becomes more significant with several patterns.

Next to these experiments, we have investigated which parameters affect the performance of our extended pattern compared to the native implementations. Table 5 and 6 show the execution times of the native implementations and of our extended pattern against the number of arguments of the send and receive methods. The numbers in braces are the percentage values of the execution time

against the implementation with no arguments. The experimental setting was unmodified except for the number of arguments. As expected, this number affects the execution time. The key observation is that the increase of the execution time of the static variants (mixin and static case of the extended pattern) compared to the dynamic variants (standard pattern and dynamic variant of the extended pattern) is similar. We found no indication that our approach is not beneficial for different numbers of arguments. The small overhead of the dynamic case is caused by the additional delegation.

Further experiments examined which other parameters affect the performance of our approach compared to the native ones. We hypothesized that the following ones might be relevant: number of virtual methods of the strategies, number of concrete strategies, and depth of the inheritance tree of the strategies. The experiments (numbers are almost identical to Tables 3 and 4) told us that none of these parameters affects performance, neither of the native implementations nor of our extended pattern.

5.3 Discussion

Our experiments have shown that the implementation of our extended strategy pattern has a footprint and performance comparable to the one of the common variants. The binding time is configurable. It affects performance and footprint as well as flexibility and configurability. Moreover, programmers can reuse implementations in all variants. This is in line with the well-known *separation of concerns* [26, 21]. In addition to the separation of an object and its behavior, our approach also provides a separation of the binding mechanism (i.e., a separation of the concerns performance, memory footprint, configurability and flexibility) from the application logic. All this does not conflict with technical requirements. The benefits of our approach are most significant for resource-constrained and realtime systems.

6 Application to other Approaches

The configurable binding mechanism is applicable to other patterns as well as to other design and implementation methodologies. This section discusses the most promising ones.

variant	mixin	pattern
time	131.7	160.5

Table 3. Average execution time of the native implementations (in milliseconds for 1.000.000 iterations).

variant	static case	dynamic case
time	132.8	178.9

Table 4. Average execution time of the extended pattern (in milliseconds for 1.000.000 iterations).

arguments	0	5	10
mixin	131.7	165.6 (123.9%)	184.5 (140.9%)
std. pattern	160.5	183.4 (114.2%)	210.2 (130.9%)

Table 5. Average execution time of the native implementations against the number of arguments (in milliseconds for 1.000.000 iterations).

arguments	0	5	10
static case	132.8	164.6 (123.9%)	180.8 (136.1%)
dynamic case	178.9	224.7 (125.6%)	259.9 (145.3%)

Table 6. Average execution time of the extended pattern against the number of arguments (in milliseconds for 1.000.000 iterations).

Design Patterns. Next to the strategy pattern, we implemented and restructured several other patterns according to our approach, namely *decorator*, *composite*, *state*, *factory method*, and *builder* (cf. [15]). In their extended form, one can use them in resource-constrained environments as well. The restructuring of patterns with only one abstract class (e.g., state) is analogous to the one of the strategy pattern and is straightforward. Experiments yield results similar to the extended strategy pattern. Patterns with several abstract classes, e.g., factory method, are more interesting. Our approach can eliminate several abstract classes. We illustrate this using the factory method (cf. Fig. 21). The factory method defines an interface (*Creator*) for creating an object. It lets subclasses decide which class (derived form the interface *Product*) to instantiate, by overriding method *factoryMethod*. The factory-method pattern lets a class defer instantiation to subclasses. This pattern includes two abstract classes, *Creator* and *Product*. In the following we discuss two restructured versions of the pattern. The first one eliminates the interface *Creator*, and the second one eliminates both interfaces, *Creator* and *Product*.

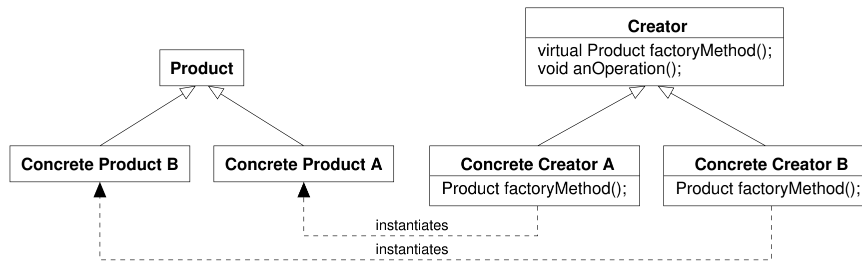

Fig. 21. The factory method pattern.

In the first version, we implemented the concrete creators as mixins. We introduced a dispatcher for the dynamic case and an interface for the dynamic binding. For the static case we need only an empty interface. The configuration is analogous to the strategy pattern. In the second implementation of the extended factory method pattern we eliminated the abstract product as well. As with the creators and the strategies, we implemented the concrete products as mixins, added a dispatcher and introduced binding interfaces for the dynamic and the static case. A programmer can configure two binding mechanisms independently of each other, one for the creators, the other one for the products. Experiments with the extended factory method pattern are positive: The restructuring affects the footprint and the performance as expected. Consequently, the second version is faster and smaller than the first one.

Program Families, Collaborations, and Mixin-Layers. The rationale behind program families is to maximize the reuse of software components and to tailor software to the requirements of an application scenario [26]. Thus, they are particularly interesting for embedded or realtime systems. A program family consists of a bottom-up layer of components. Collaboration-based design [2] and mixin-layers [29] map program families to the object-oriented world. A collaboration of objects implements each layer. A programmer refines layers by subclassing their objects.

We have already applied our approach to a program family of middleware platforms [1]. We have used our extended strategy pattern to implement a flexible synchronization mechanism (synchronous vs. asynchronous sending of data) and different connection types (sending datagrams vs. streams) of the internal socket model. Figure 22 and 23 depict connections implemented as mixin layers in top-down order. We have embedded the connection (context) in a basic layer. The connection owns a reference to a socket (strategy) object. The socket layers are arranged below the connection. If a programmer requires late binding, he puts the dispatcher between connection and sockets (cf. Fig. 22). The connection then refers to the dispatcher which delegates a call to the concrete socket implementation. With early binding, one socket is directly bound to the connection (cf. Fig. 23). It must provide the interface which the connection expects. – Integrating this approach into the program family makes the program family more and easier configurable. It preserves the reusability of the implementations of the concrete synchronization mechanisms and connection types. This high degree of configurability was one main objective [1, 28].

Frameworks. Frameworks provide domain-specific solutions for a problem domain in form of a set of classes [13, 11, 19]. Example domains are network- or GUI-programming. The programmer can customize the functionality of the framework by refining a set of abstract classes. They form the variation points of the framework. The drawbacks of an excessive use of abstract classes are well-known, namely performance penalties and a higher memory consumption. To map our ideas to frameworks, the variation points do not have to be implemented as abstract classes or interfaces, but as tem-

plate parameters (we argue). A programmer can then instantiate a variation point (framework mixin-class) using a dispatcher (dynamic case) or concrete classes (static case).

Aspects. Aspect-Oriented Programming aims to encapsulate and separate crosscutting concerns [21]. This prevents code-tangling and scattering. Many aspect-oriented languages and frameworks provide aspect-inheritance, e.g., *AspectJ* [20], *AspectC++* [30], *PROSE* [27], *AspectWerkz* [34]. Design patterns are important for this programming paradigm as well. They are general solutions to recurring problems. The aspect languages mentioned implement aspects as classes. Thus, one could combine abstract aspects, inheritance and template-based mechanisms [23, 16, 22] to implement a similar configurable binding mechanism for aspects. This is advantageous for dynamic aspects, which are instantiated at runtime.

Summing up, our approach is not only applicable to specific design patterns, but to any software which must be customizable or variable. In particular, software for embedded systems requires a binding that is configurable. Our binding mechanism is applicable to restructure existing software. Typically points where to insert are abstract classes, templates and template-based inheritance.

7 Related Work

Research on design patterns, mixins and aspect-oriented programming is related to our approach, as we will explain.

The topic of design patterns and pattern-oriented software development has a long tradition: Gamma et al. [15] and Buschmann et al. [6] give a comprehensive overview on patterns and their use. Beuche et al. [4] argue that common object-oriented methods such as design patterns fail in the domain of deeply embedded systems. The excessive use of delegation and abstract classes/methods leads to performance penalties and to an unacceptable memory consumption. They propose the combination of feature modeling, aspect-oriented programming and design patterns to build reusable, configurable and lightweight software for deeply embedded systems [3, 14]. In contrast to our approach, they have to use

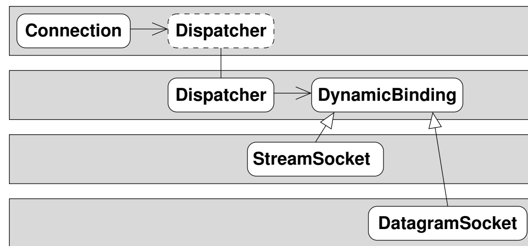

Fig. 22. Late binding of connections and sockets.

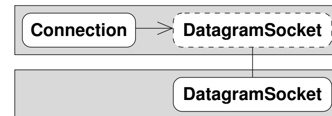

Fig. 23. Early binding of connections and sockets.

several tools like code analysis and transformation systems as well as definitions of requirements. Our approach is less complex but also less flexible.

Bracha et al. [5] were first to propose *mixin-based inheritance* using templates. Batory et al. [29] have shown how to exploit this concept to achieve a high degree of configurability. They propose *mixin layers* as large-scale components. VanHilst et al. [33] and Czarnecki et al. [10] use similar mechanisms to increase the level of configurability and reusability. Contrary to our approach, these approaches focus on compile-time configurability. They do not consider runtime configuration. Further, a programmer cannot reuse implementations for runtime configuration.

Cardone et al. [7] have proposed a new type of design pattern, the *sibling pattern*, to solve the *extensibility problem* [9]. To implement the sibling pattern, mixin layers are used. Ostermann et al. [25] have proposed *delegation layers* to achieve runtime configurability. Delegation layers have characteristics similar to mixin layers, but are composable at runtime. Moreover, they solve the extensibility problem in a way similar to the sibling pattern. But currently there is no programming language with the features required to implement delegation layers that is usable in practice. The high degree of indirection causes a significant runtime overhead.

Hanneman et al. [18] combined design patterns and aspect-oriented programming to improve software reuse. Haneberg et al. [17] increase configurability and reusability using *parametric introductions*. Loughran et al. [23] have proposed *framed aspects* which combine templates and aspects to increase configurability. These approaches focus on aspect-oriented programming only and cannot be easily applied to other programming paradigms.

8 Conclusions

This article has proposed a new implementation scheme for software that is reusable, a configurable binding mechanism. It allows to choose between late and early binding and preserves reusability of application-specific classes. Our approach is particularly useful for resource-contrained environments, e.g., real-time and embedded systems, because of its nice performance characteristics and scalable memory consumption.

The approach combines the virtues of mixins and common design patterns. The application programmer can configure software to be either runtime-configurable or compile-time-configurable. This is important, since both variants have their advantages. We have shown that an extended strategy pattern and its two instances (static and dynamic) are comparable to common design patterns and mixin implementations in terms of performance and resource consumption. Another big advantage is reusability of concrete strategy implementations. We have explained that our results are applicable to other patterns as well as to other design and implementation methodologies, notably program families, frameworks, and aspects. Thus, configurable binding is a general solution to build config-

urable, reusable and extensible software without performance penalties or excessive memory consumption.

References

1. S. Apel and K. Böhm. Using Mixins to Build a Flexible Lightweight Middleware for Ubiquitous Computing. In *Proceedings of the 4th IEEE Workshop on Software Engineering and Middleware (SEM'04) in conjunction with the 19th IEEE Conference on Automated Software Engineering (ASE'04)*, Linz, Austria, September 2004.
2. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), 1992.
3. D. Beuche, R. Meyer, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Streamlined PURE Systems. In *Proceedings of the 3rd ECOOP Workshop on Object-Orientation in Operating Systems (ECOOP-OOOSWS 2000)*, France, 2000.
4. D. Beuche, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Streamlining Object-Oriented Software for Deeply Embedded Applications. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*. IEEE Computer Society, 2000.
5. G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'90) and Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'90)*, 1990.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, New York, 1996.
7. R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using Mixins to Build Flexible Widgets. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*. ACM Press, 2002.
8. Business Communications Company. Future of Embedded Systems Technology, 2000. BCC Press release on market study RG-229.
9. W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Foundations of Object-Oriented Languages*, June 1990.
10. K. Czarnecki and U. Eisenecker. Synthesizing Objects. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, Lisbon, Portugal, 1999.
11. L. P. Deutsch. *Design Reuse and Frameworks in the Smalltalk-80 System*. Addison-Wesley, 1989.
12. K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, 1996.
13. M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors. *Building Application Frameworks*. Wiley Computer Publishing, 1999.
14. A. Gal, W. Schröder-Preikschat, and O. Spinczyk. On Minimal Overhead Operating Systems and Aspect-Oriented Programming. In *Proceedings of the ECOOP Workshop on Object Orientation and Operating Systems (ECOOP-OOOSWS'01)*, Hungary, 2001.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

16. P. Greenwood, L. Blair, N. Loughran, and A. Rashid. Dynamic Framed Aspects for Dynamic Software Evolution. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (held with ECOOP'04)*, Oslo, Norway, 2004.
17. S. Hanenberg and R. Unland. Parametric Introductions. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, 2003.
18. J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, 2002.
19. R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), 1988.
20. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, 2001.
21. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.
22. D. Lohmann, G. Blaschke, and O. Spinczyk. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, Vancouver, Canada, October 2004.
23. N. Loughran and A. Rashid. Framed Aspects : Supporting Variability and Configurability for AOP. In *International Conference on Software Reuse (ICSR-8)*, Madrid, Spain, 2004.
24. Object Management Group (OMG). *Unified Modeling Language (UML)*, 1.5 edition. http://www.omg.org/technology/documents/formal/uml.htm.
25. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*. Springer-Verlag, 2002.
26. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions On Software Engineering*, SE-5(2), March 1979.
27. A. Popovici, G. Alonso, and T. Gross. Just-in-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, 2003.
28. H. Sichting. A Middleware-Architecture for Mobile Information Systems. Master's thesis, Otto-von-Guericke University, Magdeburg, Germany, 2004. in german.
29. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 11(2), 2002.
30. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.
31. B. Stroustrup. *The C++ Programming Language (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1991.
32. J. Turley. The Two Percent Solution. *Embedded Systems Programming*, 2002. http://www.embedded.com/story/OEG20021217S0039.
33. M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *2nd JSSST International Symposium on Object Technologies for Advanced Software*, 1996.

34. A. Vasseur. Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address it? In *Proceedings of the Dynamic AOP Workshop in conjunction with 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, March 2004.
35. M. Weiser. Hot Topics: Ubiquitous Computing. *IEEE Computer*, 26(10), 1993.