Design Patterns for Independent Building Blocks

Ullrich Köthe
Fraunhofer-Institute for Computer Graphics, Rostock, Germany
Joachim-Jungius-Str. 11, D-18059 Rostock, Germany
Email: koethe@egd.igd.fhg.de

Abstract: The pattern language presented in this paper aims at helping designers to develop flexible building blocks that can be plugged together as needed by the application to be built. The patterns try to identify essential properties of flexible and reusable software. In particular, we show that extensive standardization is not a necessary prerequisite of flexibility as long as interfaces are designed in a way that supports building block adaptation. We hope that the presented design approach will be a small step towards the long envisioned "software factory".

Introduction

Over the decades, many different ways of designing and developing software have been invented. Deciding which way to take in a given situation has become difficult not only for novices, but even for experts. Many design approaches are competing with each other, and the followers of one method usually claim that theirs is superior to all others. Nevertheless, almost any approach can successfully be used to produce elegant software. However, no approach does consistently produce *flexible software components* that are easy to change, maintain, and reuse. These qualities still seem to be mostly the result of the designer's insight and intuition rather than the outcome of the application of a set of well-defined rules. This paper presents a pattern language that describes the current results of an ongoing effort to identify essential properties of flexible software, independently of any specific design method. Although most underlying ideas have been described by other authors before, often they are covered only briefly, or scattered within a huge body of other material. Presenting them as a pattern language has been of great help to me in better understanding why certain methods did or did not work in a given situation.

When we develop software artifacts, we have to deal with two fundamental problems: complexity and change. In terms of the number of possible internal states, software is among the most complex of human creations. And an ability to change is indeed the defining characteristic of software as a "soft" entity. Two basic techniques are used to manage complexity and change: abstraction and reuse. *Abstraction* describes our ability to concentrate on certain aspects of a problem and leave out other properties. Using abstraction, we can decompose a complicated problem step-by-step into simpler ones, until each sub-problem can be solved. A good decomposition also ensures that only a small part of the system is affected by

changing requirements. *Reuse* allows us to concentrate creativity and effort on a few sub-problems, while existing solutions are applied to the others. These two basic techniques also characterize successful software design.

Our discussion of flexible software will be based on the concept of a *building block.* A building block is a software artifact that provides some piece of functionality and establishes a boundary between this functionality and the rest of the world. Thus there is an inside (the implementation) and an outside (the environment or context). Building blocks exist at all scale levels. The smallest ones are just the built-in types and procedures (including functions and methods) of the programming language. On top of these we built classes and objects. At the next level we find modules (a.k.a. packages, units, components). These are used to built application programs, which are finally grouped into application systems. Obviously, these types of building blocks differ widely in terms of their scope and technical realization.

Nevertheless, there are a number of interesting and fundamental commonalties which we try to explore in the patterns below. Essential characteristics of good, flexible building block collections are largely independent of their scale level and the methods to design them. Thus, many methods that are commonly regarded as incompatible and competing actually solve the same problem in different contexts and at different scales, thus complementing rather than contradicting each other.

The Patterns

In the sequel, the patterns will be presented in the style invented by Christopher Alexander [ALEXANDER+77]. In my opinion, this less formalized style is most suitable for describing ideas which don't easily fit into a predefined set of subsections. In fact, tasks such as identifying the scale of the problem and decomposing it into good abstractions, have been and will be as much an art as a science. When reading the patterns, it is also important to keep in mind that they don't stand for themselves, but need to be applied in concert to live up to their full potential. The following table gives an overview over the patterns presented.

Pattern Name	Category	Problem	Solution
ONE SECRET PER BUILDING BLOCK	building block definition	How should we assign tasks to building blocks?	Make sure that only one well-defined task is assigned to each. Delegate other tasks explicitly.
ABSTRACTION OF DEPENDENCIES	definition	Explicit delegation does not ensure flexibility.	Describe services abstractly as to pose as little restrictions on suitable servers as possible.

OFFERED INTERFACE	building block specification	The inner workings of a building block must be hidden.	Specify a building block's capabilities, invocation syntax, and post-conditions in an offered (exported) interface.
REQUIRED INTERFACE	specification	Clients tend to depend on the interfaces of specific servers.	Specify a building block's requirements in minimal required (imported) interfaces.
DESCRIPTIVE INTERFACE	specification	A building block is hard to use if it is not properly described.	Provide machine readable descriptive interfaces and organize them into meta building blocks.
FLEXIBLE INTEGRATION	building block integration	Inadequate integration technology destroys flexibility and performance.	Prefer generative technologies at lower and combination at higher levels.
INTERFACE STANDARDIZATION	integration	The number of independent interfaces explodes.	Unify related interfaces into abstract, standardized interfaces.
INTERFACE TRANSLATION	integration	Despite standardization, mismatch between interfaces inevitably arises.	Take a proactive attitude towards interface translation and prefer interface definitions that use explicit connectors.
SELF-CONFIGURATION	building block usability	It is error prone and time consuming to set up all collaborations manually.	Standardize meta-information so that building blocks be automatically configured and adapted.
FLEXIBILITY ON DEMAND	usability	Humans are overwhelmed by extensive configuration opportunities.	Provide defaults and wrappers to hide flexibility until it is really needed.

Table 1: Overview over the patterns described in this paper

Building Block Definition

Most problems cannot be solved directly. They have to be decomposed recursively into simpler problems, until each of the problems can be solved. Defining the subproblems and their corresponding building blocks is the purpose of analysis and design. Clearly, giving a complete account of these activities is beyond the scope of this paper. So, I present only those patterns which I think are most significant for the definition of *independent* building blocks.

ONE SECRET PER BUILDING BLOCK

How should we assign tasks to building blocks?

Separation of concerns has long been considered a key technique in handling complexity and change. But exactly how much functionality each building block should provide has been less clear. On the one hand, it seems natural to keep the number of different building blocks to a minimum to avoid losing the overall perspective of the system. On the other hand, smaller building blocks are easier to handle individually.

During the last years, we have observed a tendency towards smaller building blocks. This general tendency is nicely exemplified by the evolution of multi-tier systems. Originally, business applications were build as monolithic programs containing the business logic, the user interface, and persistent storage. Later it was realized that reusability was improved by delegating the persistent storage to a database server. In such a client-server or two-tier, system the database could serve many different client applications, and the same client application could be run on top of various databases. With the invention of middleware technologies (such as CORBA [OMG95]) an even finer decomposition became practical: Now, the user interface (including the necessary visualization and dialog functionality) was also separated, while the application logic remained in the middle tier of the thus obtained three-tier architecture. Meanwhile, multi-tier systems are being built to allow for an even higher degree of flexibility and reusability.

Experience has shown, that it is best to restrict a building block to only *one* responsibility. We can interpret an exchangeable building block as a point on a particular axis of change, with all compatible building blocks being different points on that same axis, whereas incompatible building blocks refer to other dimensions of change. Thus, the idea of one task per building block is to make all dimensions of change orthogonal and uncorrelated, so that building blocks can indeed be freely combined, and there is no cross-talk (i.e. a cascade of consequential changes) if we modify just one axis.

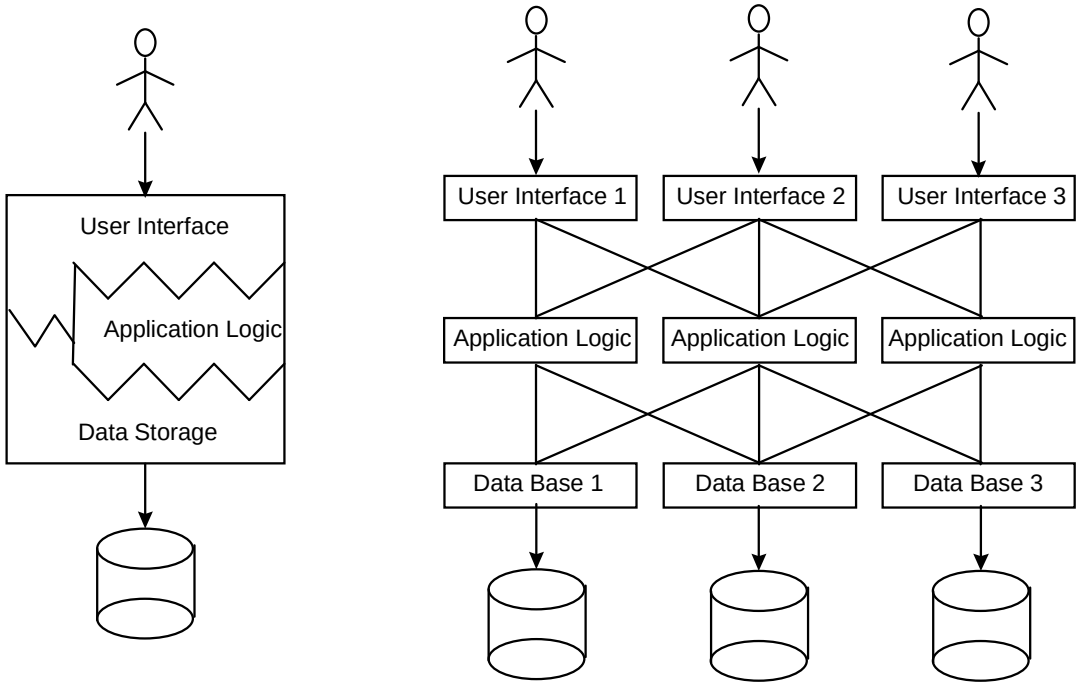

Figure 1: Left: Monolithic design (large building blocks with several tasks). right: Three-tier architecture: one task per building block

Of course, the requirement of one task per building block is not a strictly defined concept. There is some room for deciding what exactly belongs to one specific task. Experience shows, however, that programmers tend to put to much functionality into each building block. For example, typical secondary tasks, such as data access, user interfaces functionality, memory management, concurrency control, and error handling, are often interleaved with the primary task of a building block, although they should rather be delegated. Also, programmers often forget that the coordination of other building blocks is a task in itself, which should reside in a building block on its own rather than being merged with functional code.

Therefore:

When you decompose a system into building blocks, make sure that only one well-defined task is assigned to each. Delegate other tasks to collaborating building blocks.

ABSTRACTION OF DEPENDENCIES

Explicit delegation alone does not lead to flexibility as long as dependencies are concrete and hard-wired.

Look at the following example: In C++, we can generalize the expression `a+b` to any type we desire by using operator overloading. The standard idiom to implement this 'adder building block' reduces addition to a copy construction and an add-assign operation +=. For complex numbers, this looks as follows:

```
complex operator+(complex const & a, complex const & b)
{
    complex result(a);    // copy construct result from left argument
    result += b;          // add-assign right argument to result
    return result;
}
```

This (extremely simple) building block delegates most of its work to the `complex` class. However, this dependency is hard-wired, so the new building block is not reusable for other types. By using templates, we can describe the dependency more abstractly and generalize addition to any type which supports add-assignment:

```
template <class T>
T operator+(T const & a, T const & b)
{
    T result(a);          // copy construct result from left argument
    result += b;          // add-assign right argument to result
    return result;
}
```

The formal type parameter `T` takes the place of the concrete type `complex`. However, this implementation only covers the case when left and right argument of the addition have the same type. The case of adding a `complex<float>` to a `complex<double>` is not covered. Therefore, we extend the operator implementation to take two template arguments. However, there is an additional problem that must be solved: the correct return type of the expression should be determined automatically. I.e., the adder should know that the addition of a `complex<double>` to a `complex<float>` yields a `complex<double>`, much like adding an `int` to a `float` yields a `float`. This problem is solved by the introduction of a meta-class `PromoteTraits`, which encapsulates type promotion rules applicable under these circumstances:

```
template <class T1, class T2>
PromoteTraits<T1, T2>::Promote   // return promote type of T1 and T2
operator+(T1 const & a, T2 const & b)
{
   PromoteTraits<T1, T2>::Promote result(a), tmp(b); // promote args
   result += tmp;    // add-assign promoted right argument to result
   return result;
}
```

The important observation about this example is how generalization of the addition operator is achieved. The first version knows exactly, that it depends upon a type `complex` with known properties. The second version of the operator accepts any type which supports copying and add-assignment. But we still require right and left arguments to be of identical types. In the third version, this assumption is also dropped, which leads to a new abstract dependency, namely the presence of a meta-class `PromoteTraits`[1] defined for the two operand types.

I call this process of gradual generalization ABSTRACTION OF DEPENDENCIES: instead of saying specifically, which collaborators it needs, the building block describes only abstract properties of its collaborators. It thus becomes more and more invariant with respect to changing collaborators.

Ideally, the degree of abstraction of the intention is directly reflected in the implementation. This is the case in the above example: although very abstract, the code resulting from implementation variant 3 can be readily compiled, provided the requirements (`PromoteTraits::Promote` is defined and supports add-assignment, T1 and T2 are convertible to their promote type) are met. This is very nice, because it removes the gap between abstraction and implementation that often defeats flexibility in practice.

It is, however, not always possible to formulate the code as abstractly as the intention. This is especially true if the operations of collaborating building blocks need to be tightly entangled to fulfill their purpose, as is the case in many temporal collaborations such as synchronization (lock and unlock must be inserted at the right places) or optimization (operations must be rearranged to run faster). Solutions for these cases are currently an active area of research.

When one abstracts dependencies, it is extremely important to pay attention to the scales of software design. If abstraction is done with the wrong granularity or with an unsuitable mechanism, the abstraction overhead may become so large as to render the abstraction useless. We will come back to this problem in the pattern FLEXIBLE INTEGRATION.

Therefore:

A building block that depends on services provided by other building blocks should describe these services abstractly as to pose as little restrictions on suitable servers as possible. Depending on the building block's scale level, the right kind of abstraction must be determined.

---

[1] The `PromoteTraits` meta-class is a template specialized for all pairs of types we want to mix in an expression. It normally returns the 'higher' of its two argument types like this:

```
struct PromoteTraits<int, float> { typedef float Promote; };
```

Building Block Specification

To be usable, building blocks must exactly specify their interfaces. Interfaces describe the assumptions a client can make about a building block's behavior, as well as the assumptions the building block itself makes about its collaborators. ABSTRACTION OF DEPENDENCIES requires that these two kinds of assumptions be treated differently. Therefore, we distinguish offered and required interfaces. In addition, descriptive interfaces will provide meta-information that enables automatic configuration of collaborations.

OFFERED INTERFACE

(also known as : EXPORTED INTERFACE)

The inner workings of a building block must be hidden.

The traditional way of looking at interfaces corresponds mostly to what I call OFFERED INTERFACE. The OFFERED INTERFACE specifies which services a building block exports, how these services can be invoked, and which post-conditions hold after the services' completion. Although this sounds easy enough, in practice it is very difficult to decide exactly what should go into the offered interface of a given building block – should it be minimal or extensive?

A partial answer to this question is given by the pattern ONE SECRET PER BUILDING BLOCK: Every building block should contain just one well-defined piece of functionality. This directly translates into the interface being narrowly focused – all offers must describe a facet of the basic functionality. But still, this leaves us with a considerable degree of freedom since there are always many different ways to present the same functionality.

Many designers advocate minimal offered interfaces. A minimal interface must ensure at least *controllability* and *observability* [WEIDE+96]. This means that the interface's functions must be powerful enough to (1) reach every point in the building block's state space from every other point, and (2) tell exactly whether two instances of a building block refer to the same point in the state space or not. On the other hand, *minimality* requires that there is exactly one way to invoke every facet of the functionality.

Consider. for example, a stack: in principle, four functions are sufficient: `createStack`, `isEmpty`, `push`, and `pop`. We can transform any state into any other by pushing and popping appropriate elements. We can find out whether two stacks are equal by popping and comparing elements until a difference is found (the original state can be restored if the popped elements are put in a temporary stack).

There are, however, severe practical problems with this minimal interface. First, it leads to performance problems. For example, if we want to find out the size of the

stack, we must count how many elements we have to pop until `isEmpty` yields true. Clearly, this is very inefficient. Second, minimal interfaces don't support system evolution: if an offered interface has to be changed, the change is preferably implemented as an extension, so that existing clients do not break. Of course, extensions destroy minimality.

Third, in many cases minimality is not even a well-defined concept because it may depend on very subtle variations in what clients want to do. Consider a client that periodically monitors whether certain elements are in the stack. This functionality is most naturally implemented by adding an iterator to the stack's interface. In contrast to the pop-and-push-back method required by the minimal interface, iterators would be both more efficient and non-destructive, but is the interface still minimal? These questions get even more difficult when building blocks become more complicated – what is, for example, the minimal interface of a spreadsheet component?

So, in practice the idea of a minimal offered interface does not work very well. A server is more reusable if it is more powerful. This leads us to consider extensive offered interfaces. A prototypical example for this approach is the PostScript language: PostScript is clearly more powerful than strictly required to control printers, but is extremely versatile: pixel graphics, line graphics and text are all handled easily regardless of paper size and printer resolution.

However, extensive offered interfaces create their own bag of problems. First, the pattern ONE SECRET PER BUILDING BLOCK must not be violated. Also, the more functions a building block offers the more difficult it is to maintain. This means that functions should not be added to an offered interface just because some client *might* want to use them. Only real requirements of real clients justify interface extension.

Another important consideration is the *interface segregation principle* [MARTIN96]: clients should not be forced to depend upon interfaces that they do not use. At best, any change in the offered interface causes a recompile of the depending building blocks. In practice, the situation is even worse: client programmers tend to use everything offered, so a potential dependency is turned into a real one. Avoiding this problem has traditionally created extremely difficult engineering trade-offs: which functions should be offered to clients, and which should be hidden? Wrong decisions here will be very expensive to correct when detected to late.

There is only one good solution to this problem: we must ensure that clients do not *directly* depend on any offered interface. This is the purpose of the REQUIRED INTERFACE pattern to be described next. When clients don't know directly which functionality a server actually provides, additional server capabilities do not generate new dependencies. Thus, there is no longer a need to hide functionality a server could in principle provide. An extensive offered interface becomes feasible.

Therefore:

Specify a building block's capabilities, invocation syntax, and post-conditions in an offered interface. Optimize usability by making the offered interface powerful enough to allow for many different ways to use the services provided.

REQUIRED INTERFACE

(also known as : IMPORTED INTERFACE)

Clients tend to become dependent on the interfaces of their servers. This must be prevented to design truly independent building blocks.

Consider an operating system that provides a service "Printing" to application programs. Usually, the operating systems doesn't itself implement this service (one task per building block), neither does it directly delegate the work to a specific printer (abstraction of dependencies). Instead, it publishes a device driver specification which describes the necessary capabilities of possible print servers. Any server that conforms to this required interface (i.e. implements a device driver with the required capabilities and syntax) can be used as a printing device. Thus, the operating system becomes more reusable because it can now cooperate with many different printers adaptable to the required interface.

Thus, a REQUIRED INTERFACE formally describes assumptions a building block makes about its environment. It specifies which kinds of other building blocks it is going to import, which pre-conditions they must fulfill, and how their services will be invoked. Required interfaces are naturally created in the process of ABSTRACTION OF DEPENDENCIES. Consider again the generalization of the expression `a+b` discussed in that pattern: the requirements which were derived in the curse of abstraction are exactly the required interface of the final implementation of the adder building block.

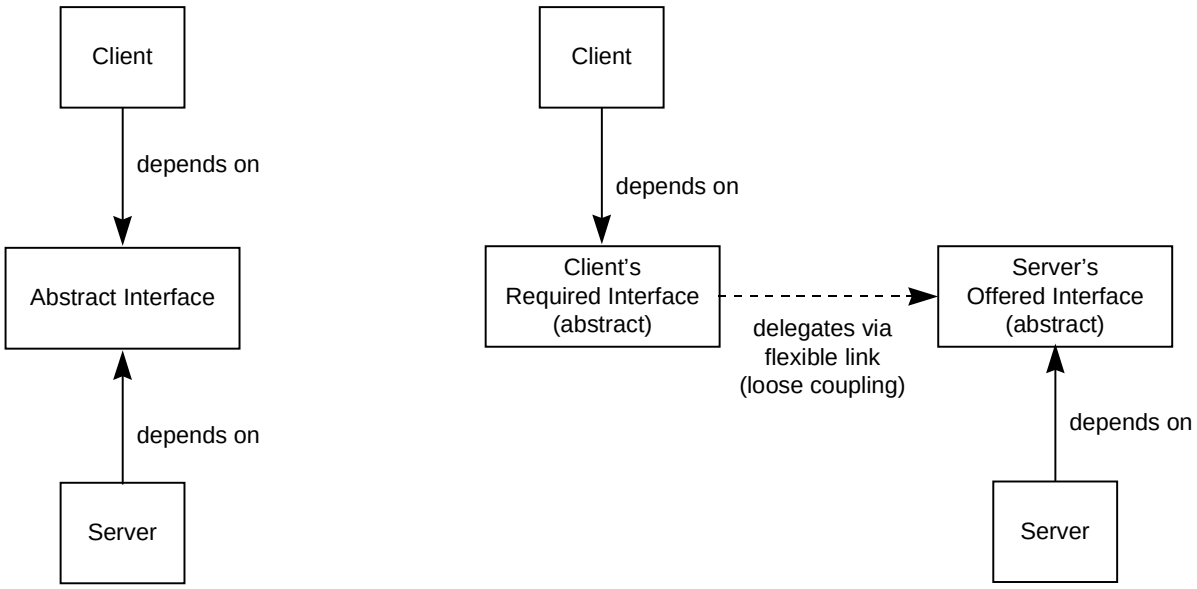

Figure 2: Left: client and server depend on the same interface and are thus not independent. Right: separation between client's required interface and server's offered interface results in truly independent building blocks

The difference between offered and required interfaces is nicely seen in plug-in interfaces: An application, such as a WWW browser, specifies how a server that shall extend its capabilities must look like to make it acceptable for the application. Beyond that integration functionality, the original application has no idea which services a new plug-in is going to offer. In case of a WWW browser, it could contain services as diverse as a viewer for an additional content format, support for script programming, or a front end for a database.

REQUIRED INTERFACE may also help in the understanding of *roles*. Roles define different views on a server's core object in the context of different collaboration patterns [REENSKAUG+96]. For example, a *person* object may take on the role of a *student* while it collaborates with a *university*, but will be an *employee* in the context of earning money at a *company*. Thus, roles may be interpreted as the required interfaces of building blocks which implement object collaboration. This is in contrast to the interpretation of some authors, e.g. [KRISØST96], where roles are primarily seen as extensions of the core objects. In our interpretation, the requirements of the collaboration come first, and extending core objects is just one implementation choice to fulfill the required interface of the collaboration.

Separating required and offered interfaces is fundamental to flexibility:

1. It gives the idea of minimal interfaces a better meaning. Since we know exactly, what the client needs to do with the server, the required interface can actually state a minimal set of requirements. This solves the interface segregation problem [MARTIN96]: a client with a minimal required interface can never depend upon functionality it does not use.
2. It allows a much finer control over access rights than traditional public/private declarations. Even if some function is publicly available in the server's offered interface, the client must not use it if it's not declared in the required interface. By accepting or rejecting the required interface, a server can control access rights on a per-client basis instead of relying on indiscriminate public/private declarations.
3. It ensures that the client depends on *stable* abstractions: The required interface only changes if the client itself is modified. Thus, the required interface shields the client from any changes in the servers' offered interfaces caused by requirements of other clients. Client and server can evolve relatively independently.

Of course, separation of offered and required interfaces also means that there is no longer a guarantee that interfaces fit together automatically. This problem will be the topic of the patterns in section "Building Block Integration".

Therefore:

Specify a building block's requirements with respect to collaborating servers (necessary capabilities, call interfaces, and pre-conditions) in a required interface. Make the required interface minimal as to avoid dependencies on unneeded server capabilities.

DESCRIPTIVE INTERFACE

A building blocks is hard to use if it is not properly described.

A classical example for a simple descriptive interfaces is the `sizeof()` command in the C language. Suppose you want to allocate an array of integers on a 16 bit machine. From the compiler documentation you learn that an `int` takes 2 bytes of storage. Thus, without `sizeof()`you have to write:

```
int * array = (int *)malloc(2*length_of_array);
```

Clearly, this is a maintenance nightmare: If on another system integers require 4 bytes, you have to search for all the hard-wired occurrences of the old (2 byte) assumption. By providing a descriptive interface which tells the size of any data structure, the compiler frees the programmer from explicitly knowing these sizes:

```
int * array = (int *)malloc(sizeof(int)*length_of_array);
```

A DESCRIPTIVE INTERFACE is defined as any kind of *machine readable documentation*, i.e. declarative information which can be used by other building blocks to simplify integration and adaptation. The descriptive interface differs from the two other interface varieties in a very important respect: a building block does not depend on its own descriptive interface. In other words, a building block can work without documentation (but it's much harder to maintain and use then).

The independence of the descriptive interface has a very nice consequence: we need not keep descriptive information in the building block itself, but can move it into dedicated *meta building blocks*. Meta building blocks allow us to extend descriptive interfaces without modifying the building blocks themselves. Consequently, when a client requests descriptive information about its collaborators, we can always add suitable meta building blocks as needed. Meta building blocks can also refer to several primary building blocks simultaneously (like, for example, the `PromoteTraits` introduced earlier). This is very useful to express constraints and other relationships *between* building blocks. Thus, meta building blocks are an important prerequisite for SELF-CONFIGURATION.

Descriptive interfaces can describe static as well as dynamic properties (the latter is often called a *reflective architecture* [BUSCH+96]). It is also possible to implement tools which extract parts of the descriptive interface automatically. For example, some compilers use the output of profilers to identify performance bottlenecks which will be optimized more carefully during the next compiler run.

Therefore:

Provide machine readable DESCRIPTIVE INTERFACES which describe the properties of building blocks and the constraints on their composition. Organize this information into meta building blocks and extend it as necessary.

Building Block Integration

Building block integration is the task of implementing larger components by composing building blocks together. According to the previous section this means that *required interfaces must be mapped onto offered interfaces*. This section describes several issues that need to be considered when doing so.

FLEXIBLE INTEGRATION

If an inadequate integration mechanism is chosen, flexibility and/or performance are lost.

When defining cooperating building blocks, we must balance two opposite forces: on the one hand, we would like building blocks to be very abstract so that they become invariant under changing collaborators. On the other hand, abstraction generates overhead, which may degrade performance so much as to make the building blocks useless in practice. The right trade-off between these forces critically depends on the scale level where the building blocks under consideration reside.

At the lowest levels, we cannot afford much overhead. Here we will thus prefer compile-time solutions. At the higher levels, run-time flexibility becomes more and more important, so that we have to apply dynamic integration techniques. Let us briefly review appropriate solutions at different scales:

At the level of **algorithms and data structures**, integration has to be done at compile-time to avoid any abstraction overhead. For a long time, code generating tools have been the only possibility to achieve compile-time flexibility. Meanwhile, code-generation capabilities have been built into important programming languages (e.g. "templates" in C++, "generic packages" in Ada, "genericity" in Eiffel). On the basis of these new features, *generic programming* has introduced novel ways to write flexible, yet efficient building blocks at the lowest scale [MUSSTEP94]. The abstract implementation of `"a+b"` presented in the pattern ABSTRACTION OF DEPENDENCIES is an example for this. The Standard Template Library (part of the Standard C++ Library, see [MUSSERSAINI95]) is currently the most advanced application of generic programming.

At the scale of **objects and micro-architecture**, we need some run-time flexibility, which is achieved by means of dynamic method dispatch. Many of the well known design patterns [GHJV94] give detailed descriptions how dynamic method dispatch is effectively applied. This is most easily achieved within dynamic languages such as SmallTalk. However, these languages introduce a relatively large run-time overhead. Compiled languages provide a more efficient mechanism by binding method dispatch to inheritance. Unfortunately, inheritance is a static relationship, which counters flexibility. Therefore, inheritance is often combined with code generation (as in many user interface builders) or generic techniques (see for example [WEIHE98]).

At the **modules and packages** level, an even higher degree of run-time flexibility is needed. Code generating techniques are therefore less suitable here. Dynamic method dispatch is successfully used, especially in conjunction with dynamic libraries. Still higher flexibility is provided by *middleware technologies* such as JavaBeans, DCOM and CORBA. At the cost of some additional overhead, their dynamic invocation capabilities even cross process and machine boundaries. This is exploited by several novel design techniques, including the promising *Component Based Development*[2] approach. Moreover, modules are often configurable so that they can be adapted to the specific needs of the client. In extreme cases, a configurable interface is specified as a programming language (like SQL for database access or PostScript for printing).

At the **architecture** level, run-time overhead is less of a consideration. So, middleware technologies are widely applied and form the basis of client-server, multi-tier and broker architectures. Sometimes, building block cooperation can be reduced to mere data exchange, for example in a pipes-and-filters architecture. Here, a common data exchange format, which is converted to the native formats of each building block dynamically, serves to integrate the system. Run-time configuration by means of integration languages or reflection and introspection also plays an important role. A comprehensive treatment can be found in [BUSCH+96].

Unfortunately, a good trade-off between flexibility and performance can not always be found. This is especially true when building blocks need to interact very tightly, like, for example, algorithms and optimizers, or data structures and concurrency control. Improved integration techniques, which can handle these cases, are currently an active area of research.

Therefore:

When integrating reusable building blocks, choose an integration technology according to the scale of the problem and the amount of run-time flexibility needed. Prefer generative technologies at lower levels and building block combination at higher levels.

---

[2] Component Based Development can be considered a special case of the design method described by the building block patterns. It applies many of the presented ideas at the module level.

INTERFACE STANDARDIZATION

If all interfaces are defined completely independently of each other, the number of interfaces will explode.

In the approach illustrated in the previous patterns, every building block will specify its offered and required interfaces somewhat independently, in the way it is most convenient for it. Thus it is quite likely that interfaces will slightly differ, even if they have essentially the same purpose. Of course, this will cause unacceptable adaptation expense. So the need for unification – standardization – arises. In this sense, INTERFACE STANDARDIZATION is not bound to the activities of some official institution, but is an integral part of every project and every organization's work.

The crucial point of this pattern is, that standardization is primarily an *abstraction* and *optimization* procedure: A standard generalizes requirements and offers to reduce the number of different interfaces needed (abstraction). It also mediates between the servers' capabilities and the clients' requests to optimally balance wishes and possibilities (optimization). This interpretation implies that standardization can only begin *after* a sufficient number of related interfaces have been collected and, more importantly, that *offered and required interfaces must be represented equally.* Without looking at both sides of the client-server relationship, standardization is impossible. Standardization is meant to respond to the requests, not to force clients to use something invented by a higher authority.

Consequently, standardization must be an iterative process. Based on the analysis of a number of interfaces, a first version of the standard is defined. Clients and servers will subscribe to the standard, if it is sufficiently close to their original requests resp. offers. The thus modified interfaces will give rise to a refinement or extension of the standard, and so on. But even when required and offered interfaces will eventually conform to the same standard they remain essentially independent.

This approach is nicely exemplified by MUSSER'S and STEPANOV'S papers on generic programming, e.g. [MUSSTEP94]. The iterator categories that have now become part of the Standard Template Library are based on an extensive analysis of what certain low-level algorithms (such as sorting and searching) would need and what popular data structures (such as input streams, lists, and vectors) could deliver. The resulting 5 iterator categories optimally balance requests and possibilities – algorithms request the simplest iterator they can afford, while containers provide the most complicated iterator they can efficiently implement.

We have made similar experiences in a project were we needed to define requirements for types that should support arithmetic operations (`+`, `-`, `*`, `/`). As long as computations were restricted to the built-in types, problems didn't arise, because all built-in types support these operations. When we extended the system to handle other types (such as vectors, matrices, RGB-triples, RGB-alpha quadruples etc.), things were less clear, because arithmetic operations on these types often have very

different semantics. Fortunately, the algorithms in question were formulated in terms of required interfaces, so we were able to analyze the their requirements in detail. This analysis resulted in two types of requirements (two interface categories which correspond to the algebraic concepts of the same names): *linear space* (where objects of a given type can be added to and subtracted from each other, and multiplied with a double) and *division algebra* (where in addition multiplication and division of objects with objects of the same type must be defined). When reviewing the algorithms according to these categories we discovered that several algorithms requesting a type belonging to the division algebra category, could in fact be reimplemented so that they only required the much weaker linear space. Other algorithms were found to need multiplication, but not division of objects, so a new category *linear algebra* was introduced to fill the gap between the other two categories. In other words, INTERFACE STANDARDIZATION lead to an iterative refinement of the requirements that were originally found during ABSTRACTION OF DEPENDENCIES of these algorithms.

Therefore:

After having collected a number of related interfaces, review them to yield abstract, standardized interface categories that generalize and balance requirements and offers. Refine the categories in an iterative process.

INTERFACE TRANSLATION

Even if you do extensive interface standardization, mismatch between interfaces will inevitably arise.

Many designers have argued that standardization is a necessary prerequisite of reusability. However, complete standardization is impossible. And even where we have standards, matching interfaces can not be guaranteed. First, according to the famous joke – "The nice thing about standards is that there are so many to pick from."– there will always be competing standards for essentially the same purpose. Second, interfaces are not completely static but evolve over time. So, at times interfaces that used to fit together will get out of sync. Third, if reuse becomes really commonplace, building blocks will frequently be used outside their original context where one cannot expect to find fitting interfaces. Reusability can not rely on standardization alone.

Since interface mismatch is unavoidable, we should no longer regard interface adaptation as a tedious and annoying work-around. Instead we should develop a proactive attitude towards this situation. We should define interfaces so that adaptation becomes as easy as possible. Explicit specification of required and offered interfaces gives us a good starting point: when the two interface types are explicit, it is no problem to insert a translator (i.e. an ADAPTER) between them if mapping is otherwise impossible. This is, however, not sufficient. Good interface specifications in the spirit of this pattern actively *support* the implementation of translators. The best way to achieve this is to treat interfaces as first class citizens: instead of defining interfaces as pure descriptions of the functionality, we should provide *explicit connectors* that actively mediate between requirements and offers.

An iterator is a typical example of this approach: there are infinitely many possibilities how linear traversals can be implemented on top of various data structures. To name just a few examples: a list allows to get to the next item, an array has an index operation taking an integer, a map provides indexing with arbitrary key types, and a graph may implement, among others, depth first and breadth first traversals. For an algorithm who wants to linearly traverse any kind of data structure, it would be extremely difficult to abstract this dependency uniformly, if a direct mapping of the required interface onto the offered interfaces of all these diverse data structures were necessary.

But when viewed through an appropriate abstract iterator, all these traversals look the same: access the current item, go to the next item, and check whether we are done. The iterator maps these basic operations onto whatever the respective data structures provide (see figure 3). Implementing the iterator as a separate object is the best way to encapsulate the (sometimes very complicated) translations these mappings might require. New iterators that modify the behavior of existing iterators or implement new, customized traversal algorithms, are easily added.

A dedicated connector gives us an explicit place (a hot-spot) which is meant to be changed when translation and adaptation becomes necessary. Thus, modification of the building blocks themselves as well as other adaptation workarounds can be avoided. In cases where translation is not necessary because required and offered interfaces fit into each other, the interface objects can be generated automatically or optimized away. Conversely, in very complicated cases (for example object-relational mapping in the database area) the interface objects can take on a live of their own and become reusable building blocks themselves.

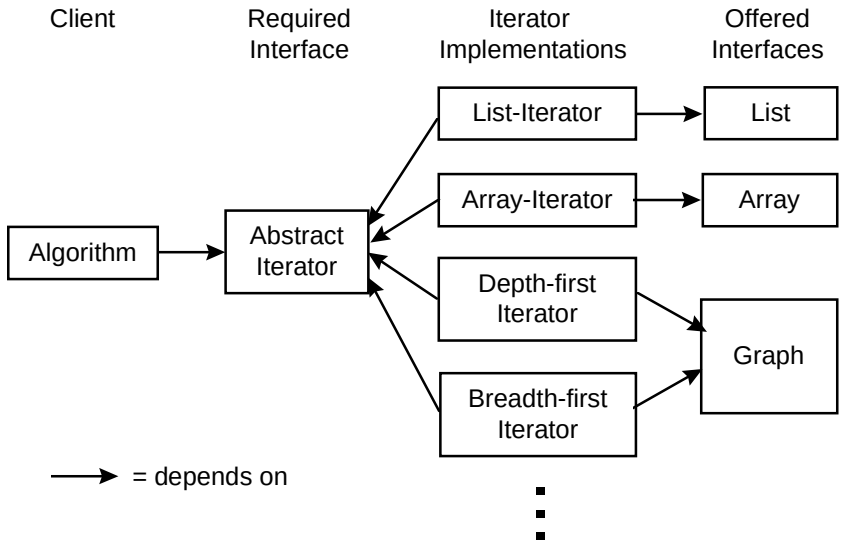

Figure 3: One algorithm works with many different data structures via iterators

Many different techniques to define and implement active interface objects are available. Examples include ADAPTER, PROXY, MEDIATOR, and CHAIN-OF-RESPONSIBILITY from [GHJV94]. In a given situation of interface mismatch, you now have a choice: When the mismatch affects many related building blocks, interface standardization might be preferable. If, however, only few building blocks don't fit, or standardization is impossible because building blocks must not be modified, you can relatively easily implement the necessary translators.

Therefore:

View interface translation as a necessary consequence of flexibility and take a proactive attitude towards it. Prefer interface definitions that use explicit connectors which serve as built-in hot-spots to encapsulate interface adaptation and transformation.

Building Block Usability

Reuse will not happen in practice when the building blocks are hard to understand and use. The following patterns describe how usability can be improved by automatic configuration and flexibility hiding.

SELF-CONFIGURATION

It is very time consuming and error prone to set up all collaborations manually. Often, selecting one collaborator implies certain choices of other, related services. If these constraints must be kept by hand, reusability will be severely reduced.

[MOWMAL97] report a nice case in point: the operating system ALTAIR 8080. It was completely configurable and could be adapted to almost any application domain. But actually doing the configurations was so hard to understand that the system was rarely deployed in praxis. Thus, it soon went out of business, while the simpler, less adaptable IBM PC experienced record sales. While good interfaces provide a high amount of adaptability, placing the entire responsibility of configuration on the programmer isn't practical.

SELF-CONFIGURATION is a way to take some of the burden away from the user. We compensate for the complications of high flexibility by introducing a *standardized meta level,* which formally describes the flexibility and serves to automate the process of setting up and controlling a network of collaborating building blocks. Meta-information allows a system to (1) identify and create appropriate building blocks, (2) control and maintain proper relationships between collaborators, and (3) configure a building block's internals according to the environment. Under the name *plug-and-play*, similar ideas have recently become very popular in the hardware context.

A common instance of the first possibility is the well-known FACTORY pattern [GHJV94]. A factory is a meta building block which creates other building blocks. By using an abstract factory, a client frees itself from the need to know the mechanisms of building block creation. According to the requirements of the current context, many different creation mechanisms can be chosen by simply exchanging or reconfiguring factories (either at compile-time or at run-time). This idea is further generalized by the TRADER pattern. A trader can be understood as a factory for factories. Building blocks pass their *required interfaces* to the trader, and it will return a factory that constructs conforming servers. A trading service is, for example, part of the OMG Object Management Architecture specification (It is, however, seldom applied because a complete formal specification of all requirements is very difficult).

The second possibility exploits descriptive interfaces which describe relationships between collaborating building blocks. C++ traits [MYERS95] (e.g. the `PromoteTraits` discussed earlier), the built-in reflection mechanisms in Smalltalk

and CLOS as well as dedicated reflection architectures can be used to automatically generate connections and connectors and to check whether a particular collaboration setup is valid. This idea is nicely exemplified by *Adaptive Programming* [LIEBERHERR96]: using a class relationship graph as meta-information, the Demeter tools automatically create adapters which route requests to the appropriate objects. Thus, clients need not know how the class graph is structured – they only see the adapter. Still more general, formal approaches to these problems, involving very sophisticated specification languages and constraint solving algorithms, are currently an active area of research, see for example [LUMPE+97].

In a different way, descriptive interfaces are used in the third possibility where building blocks adapt their internals to the properties of the environment. For example, a numeric building block could adjust the accuracy of internal computations according to the client's needs. A priority queue could select the fastest implementation after asking for the client's most likely access patterns. In practice, this is often implemented as an extension of the server's offered interface: a configurable server provides facilities to activate different modes of operation ("open implementation", see e.g. [KICZALES96]). However, in this active approach additional dependencies between client and server are generated because the client has to know which configurations exist and how they are activated. Dependencies are minimized if the server or a dedicated meta building block just read the client's descriptive interface and perform configurations without requiring the client to know about concrete options.

Therefore:

Standardize meta-information (such as descriptive interfaces and factories) so that building blocks can use it to automatically adapt themselves to their environment. Implement meta building blocks which are responsible for the configuration, creation, and validation of other building blocks within a system.

FLEXIBILITY ON DEMAND

Humans are easily overwhelmed by extensive configuration opportunities. This may result in low acceptance of otherwise well designed systems.

Automated configuration is often impossible or undesirable. Human intervention is required to set up collaborations between building blocks and configure their flexibility. Thus, we must present flexibility in a way that does not overwhelm human capabilities. That is, we would like to hide flexibility as much as possible and bring it to the surface only when needed. On the other hand, hidden flexibility must not result in lost flexibility – the gap between ease of use and full flexibility should be as small as possible. When little flexibility is needed, a building block should be a *black-box* which can be used without detailed knowledge about its inner workings. But it should also be possible to consider the building block as a *white-box* where you have many configuration options provided you know the mechanisms sufficiently well. The important point of FLEXIBILITY ON DEMAND is that the black-box should be a *thin shell* around the white-box which is easily removed when the flexibility is needed, but as easily reconstructed after the adaptations have been completed.

Several techniques have been successfully used to provide a "thin-shell black-box". The most common is certainly to provide default settings for every flexible entity. This results in a gentle learning curve, as more options can be discovered gradually, when needed. User interface toolkits such as Interviews and the Smalltalk library are nice examples of this techniques: It doesn't take more than three lines of code to get a simple, empty window on the air. In contrast, the X Window System does not follow this principle: even the most basic tasks require several dozen lines. Although providing less functionality, the X Window System is much harder to learn than the other systems.

In a known context, it is very useful to build the black box by wrapping an appropriately configured version of the building block into a physical shell (the wrapper). The flexibility is now (partially) hard-wired inside the shell and not visible from the outside. Thus, the wrapper lowers the *surface-to-volume ratio* of the original building block [FOOTEYODER97], i.e. it simplifies its external interface relative to the internal functionality and thus simplifies understanding and handling. Since the shell does not provide any additional functionality, it can be changed or even removed easily when the context changes. Technically, this is very similar to the ADAPTER pattern, but serves a different purpose.

Therefore:

Make building blocks more accessible to developers by hiding the flexibility mechanisms until they are really needed. Provide defaults and thin shells that encapsulate the flexibility, but can be overridden or removed easily on demand.

Summary and Conclusions

In this paper we have introduced a pattern language that describes important aspects of the creation of reusable, independent building blocks. It has revealed many interesting commonalties between various design techniques, but has also made clear that no single technique suffices to design building blocks for all problems. Experience with existing design patterns has shown that a compact description of a concept, along with *a unique name*, is a great way to communicate design ideas. I hope that the same will be true for the patterns presented here. In our work, the separation of required and offered interfaces and a proactive attitude towards interface adaptation has lead to much more flexible systems. The latter is especially significant since it shows that standardization is not a necessary prerequisite for flexibility.

In our organization, the presented patterns have been of great help to understand why and how a particular design technique should be used to solve a particular problem. The underlying design approach is also the basis of our VIGRA (Computer Vision with Generic Algorithms) library which supports reuse at the level of algorithms and data structures to an extend not previously possible. A detailed study of how the patterns have been applied in VIGRA will be given in a forthcoming publication.

Acknowledgments

I'd like to thank my colleagues Steffen Nowacki, Holger Diener, Klaus Hartenstein, and Uwe von Lukas for many valuable discussions on the ideas presented. I'm also very grateful to Wolfgang Keller (the shepherd), Karsten Weihe, Alistair Cockburn, and the participants of EuroPLoP '98 for comments on earlier drafts of this paper, which lead to numerous significant improvements.

References

[ALEXANDER+77] C. Alexander, S. Ishikawa, M. Silverstein: *"A Pattern Language"*, Oxford University Press, 1977

[BUSCH+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: "*Pattern-Oriented Software Architecture - A System of Patterns*", Wiley and Sons, 1996

[COCKBURN98] A. Cockburn: "*Surviving Object-Oriented Projects*", Addison-Wesley, 1998

[FOOTEYODER97] B. Foote, J. Yoder: *"The Selfish Class"*, in: R. Martin, D. Riehle, F. Buschmann: "Pattern Languages Of Program Design 3", Addison Wesley, 1997

[GARLAN+95] D. Garlan, R. Allen, L. Ockerbloom: *"Architectural Mismatch or Why it's hard to build systems out of existing parts"*, 17th Intl. Conf. on Software Engineering, 1995

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *"Design Patterns"*, Addison-Wesley, 1994

[KICZALES96] G. Kiczales: "Beyond the Black-Box: Open Implementation ", IEEE Software, vol. 13, no. 1, January 1996

[KRISØST96] B. Kristensen, K. Østerbye: "*Roles: Conceptual Abstraction Theory and Practical Language Issues*", Theory and Practice of Object Systems, 2(3), 1996

[LIEBERHERR96] K. Lieberherr: "*Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*", PWS Publishing Company, 1996

[LUMPE+97] M. Lumpe, J.-G. Schneider, O. Nierstrasz and F. Achermann: "*Towards a formal composition language*", Proc. ESEC '97 Workshop on Foundations of Component-Based Systems, September 1997, pp. 178-187

[MÄBISCH96] K.-U. Mätzel, W. Bischofsberger: *"Evolution of Object Systems"*, in: Proc. of the Ubilab Conference '96, Universitätsverlag Konstanz, 1996

[MARTIN96] R. Martin: "*The Interface Segregation Principle*", C++ Report, August 1996

[MEYER94] B. Meyer: "*Object-Oriented Software Construction*", Prentice Hall, 1994

[MOWMAL97] T. Mowbray, R. Malveau: "*CORBA Design Patterns*", Wiley and Sons, 1997

[MUSSERSAINI95] D.Musser, A. Saini: *"STL Tutorial and Reference Guide"*, Addison-Wesley, 1995

[MUSSTEP94] D.Musser, A. Stepanov: *"Algorithm Oriented Generic Libraries"*, in: Software - Practice and Experience, vol. 24, no. 7, pp. 623-642, 1994

[MYERS95] N. Myers: *"A New and Useful Template Technique: Traits"*, C++ Report, June 1995

[OMG95] Object Management Group: *"The Common Object Request Broker: Architecture and Specification"*, Revision 2.0, 1995

[REENSKAUG+96] T. Reenskaug, P. Wold, O.A. Lehne: *"Working with Objects"*, Prentice Hall, 1996

[WEIDE+96] B. Weide, S. Edwards, W. Heym, T. Long, and W. Ogden: *"Characterizing Observability and Controllability of Software Components"* , Proc. 4th International Conference on Software Reuse, 1996

[WEIHE98] K. Weihe: "*Using Templates to Improve C++ Designs*", C++ Report 10/2 (1998), 14-21